Date: September 2014

OBJECT MANAGEMENT GROUP

Ontology Definition Metamodel

Version 1.1

OMG Document Number: formal/2014-09-02
Standard Document URL: http://www.omg.org/spec/ODM/1.1/
Normative Machine Consumable Files:
http://www.omg.org/spec/ODM/20131101/ODM-metamodels.xmi
http://www.omg.org/spec/ODM/20131101/RDFProfile.xmi
http://www.omg.org/spec/ODM/20131101/OWLProfile.xmi
http://www.omg.org/spec/ODM/20131101/RDFLibrary.xmi
http://www.omg.org/spec/ODM/20131101/XSDLibrary.xmi
http://www.omg.org/spec/ODM/20131101/OWLLibrary.xmi

OMG's Issue Reporting Procedure

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

Table of Contents

Preface ix

1 Scope 1

2 Conformance 2

3 Normative References 3

4 Terms and Definitions 5

5 Symbols 8

6 Additional Information 9

6.1 Changes to Adopted OMG Specifications 9

6.2 How to Read This Specification 10

6.3 Proof of Concept 10

6.4 Acknowledgments 11

7 Usage Scenarios and Goals 13

7.1 Introduction 13

7.2 Perspectives 13

7.2.1 Model-Centric Perspectives 14

7.2.2 Application-Centric Perspectives 15

7.3 Usage Scenarios 16

7.4 Business Applications 17

7.4.1 Run Time Interoperation 17

7.4.2 Application Generation 18

7.4.3 Ontology Lifecycle 19

7.5 Analytic Applications 20

7.5.1 Emergent Property Discovery 20

7.5.2 Exchange of Complex Data Sets 20

7.6 Engineering Applications 21

7.6.1 Information Systems Development 21

7.6.2 Ontology Engineering 21

7.7 Goals for Generic Ontologies and Tools 22

8 Design Principles 25

8.1 Design Principles 25
8.2 Why Not Simply Use or Extend the UML 2.0 Metamodel? 25
8.3 Component Metamodel Selection 26
8.4 Relationships among Metamodels 27
8.4.1 The Need for Translation 27
8.4.2 UML Profiles 27
8.4.3 Mappings 28
8.4.4 Mappings Are Informative, Not Normative 28
8.5 Why Common Logic over OCL? 28
8.6 Why EMOF? 29
8.7 M1 Issues 29
9 ODM Overview 31
10 The RDF Metamodel 33
10.1 Overview 33
10.1.1 Organization 33
10.1.2 Design Considerations 33
10.2 RDF Triples 34
10.2.1 BlankNode 35
10.2.2 Node 36
10.2.3 RDFProperty 36
10.2.4 RDFSResource 37
10.2.5 RDF Triple 37
10.2.6 IRI 38
10.2.7 ReferenceNode 39
10.3 RDF Literals 39
10.3.1 RDFSResource (Augmented Definition) 40
10.3.2 Literal 40
10.4 RDF Statements 41
10.4.1 RDFProperty (Augmented Definition) 41
10.4.2 RDFSResource (Augmented Definition) 41
10.4.3 RDFStatement 42
10.4.4 Triple (Augmented Definition) 42
10.5 Classes and Utilities 42
10.5.1 RDFSClass 43
10.5.2 RDFSDatatype 44
10.5.3 RDFSResource (Augmented Definition) 45
10.6 RDF Properties 45
10.6.1 RDFProperty (Augmented Definition) 46
10.6.2 RDFSClass (Augmented Definition) 46

10.7 Containers and Collections 47
10.7.1 RDFAlt 47
10.7.2 RDFBag 48
10.7.3 RDFList 48
10.7.4 RDFSContainer 49
10.7.5 RDFSContainerMembershipProperty 49
10.7.6 RDFSeq 49
10.7.7 RDFSResource (Augmented Definition) 50
10.8 RDF Sources, Datasets, Documents and Namespaces 50
10.8.1 Document 53
10.8.2 Namespace 53
10.8.3 NamespaceDefinition 54
10.8.4 Triple 54
10.8.5 NamedGraph 55
10.8.6 Graph 55
10.8.7 Dataset 56
10.8.8 Source 56
11 The OWL Metamodel 59
11.1 Overview 59
11.1.1 Organization of the OWL Metamodel 59
11.1.2 Design Considerations 61
11.2 OWLBase Package - OWL Ontology 61
11.2.1 OWLOntology 61
11.2.2 RDFSLiteral (Augmented Definition) 63
11.3 OWLBase Package - Class Expressions 63
11.3.1 ClassExpression 65
11.3.2 ComplementClass 66
11.3.3 EnumeratedClass 66
11.3.4 Individual 67
11.3.5 OWLClass 68
11.3.6 IntersectionClass 68
11.3.7 UnionClass 68
11.3.8 Restrictions 69
11.4 OWLBase Package - Properties 75
11.4.1 OWLAnnotationProperty 76
11.4.2 OWLDatatypeProperty 77
11.4.3 OWLObjectProperty 77
11.4.4 OWLOntologyProperty 78
11.4.5 Property 79
11.5 OWLBase Package - Individuals 79
11.5.1 OWLAllDifferent 80
11.6 OWLBase Package - Datatypes and Data Ranges 81
11.6.1 DataRange 83

11.6.2 OWLDatatype 83
11.6.3 OWLDataEnumeration 83
11.6.4 11.6.4 DatatypeRestriction 84
11.6.5 UnionDatatype 85
11.6.6 IntersectionDatatype 85
11.6.7 ComplementDatatype 86

11.7 OWLBase Package - OWL Universe 86

11.7.1 OWLUniverse 87
11.7.2 OWLOntology (Augmented Definition) 87

11.8 OWLDL Package - Constraints for OWL DL Conformance 88

11.8.1 Classes in OWL DL 88
11.8.2 OWL DL Restrictions 89
11.8.3 OWL DL Property Constraints 89

11.9 OWLFull Package - Constraints For OWL Full Conformance 91

12 The Common Logic Metamodel 93

12.1 Overview 93

12.1.1 Design Considerations 93
12.1.2 Modeling Notes 94

12.2 The Phrases Diagram 94

12.2.1 Comment 94
12.2.2 ExclusionSet 95
12.2.3 Identifier 95
12.2.4 Importation 96
12.2.5 Module 97
12.2.6 Name 98
12.2.7 Phrase 99
12.2.8 Sentence 99
12.2.9 Text 100

12.3 The Terms Diagram 101

12.3.1 Argument 102
12.3.2 CommentedTerm 102
12.3.3 FunctionalTerm 103
12.3.4 SequenceMarker 103
12.3.5 Term 104

12.4 The Atoms Diagram 104

12.4.1 Atom 05
12.4.2 AtomicSentence 105
12.4.3 Equation 106

12.5 The Sentences Diagram 106

12.5.1 Biconditional 107
12.5.2 BooleanSentence 108
12.5.3 CommentedSentence 108
12.5.4 Conjunction 108

12.5.5 Disjunction 109
12.5.6 ExistentialQuantification 110
12.5.7 Implication 110
12.5.8 IrregularSentence 111
12.5.9 Negation 111
12.5.10 QuantifiedSentence 112
12.5.11 UniversalQuantification 112
12.6 The Boolean Sentences Diagram 113
12.7 The Quantified Sentences Diagram 113
12.7.1 Binding 114
12.8 Summary of CL Metamodel Elements with Interpretation 115
13 The Topic Map Metamodel 117
13.1 General 117
13.2 Topic Map Constructs 117
13.2.1 TopicMapConstruct 118
13.2.2 ReifiableConstruct 118
13.2.3 TopicMap 119
13.2.4 Topic 119
13.2.5 Association 121
13.3 Scope and Type 122
13.3.1 ScopeAble 122
13.3.2 TypeAble 123
13.3.3 AssociationRole 123
13.3.4 Occurrence 124
13.3.5 TopicName 125
13.3.6 Variant 126
13.4 Published Subjects 126
13.4.1 Type-Instance Relationship Among Topics 127
13.4.2 Subtype-Supertype Relationship Among Topics 128
13.5 Example 128
14 UML Profile for RDF and OWL 131
14.1 General 131
14.2 UML Profile for RDF 131
14.2.1 RDF Profile Package 131
14.2.2 RDF Datasets, Documents and Sources 132
14.2.3 RDF Statements 137
14.2.4 ReificationKind 144
14.2.5 Literals 145
14.2.6 Classes and Datatypes 149
14.2.7 Properties in RDF 153
14.2.8 Containers and Collections 161

14.2.9 Labeled and Named Resources 162
14.3 UML Profile for OWL 165
14.3.1 OWL Profile Package 165
14.3.2 OWL Ontology 165
14.3.3 OWL Annotation Properties 166
14.3.4 OWL Ontology Properties 167
14.3.5 Class Expressions 171
14.3.6 Properties 193
14.3.7 Individuals 200
14.3.8 Datatypes and Data Ranges 204
15 The Topic Map Profile 213
15.1 General 213
15.2 Stereotypes 213
15.2.1 Topic Map 213
15.2.2 Topic 213
15.2.3 Association 214
15.2.4 Characteristics 214
15.3 Abstract Bases 215
15.3.1 TopicMapElement 216
15.3.2 Scoped Element 216
15.3.3 TypedElement 216
15.4 Example 217
16 Mapping Topic Maps to OWL 219
16.1 Overview 219
16.2 Topic Maps to OWL Full Mapping 219
16.2.1 Overview 219
16.2.2 Packaging Construct: TopicMap 221
16.2.3 Most General Structure: TopicMapConstruct 222
16.2.4 Multiple Identifiers to SameAs 224
16.2.5 Topic to OWL Class 224
16.2.6 Subtype to Subclass 225
16.2.7 Topic to Property 226
16.2.8 Topic to Individual 226
16.2.9 Topic Subject Identifiers 227
16.2.10 Topic Subject Locators 228
16.2.11 Association to Individual 228
16.2.12 Association Role to Property 229
16.2.13 Occurrence to Property 230
16.2.14 Topic Names to Object Properties, Variants to Property Values 231
16.2.15 Scope to Property Values 233
16.3 OWL to Topic Maps 233
16.3.1 Packaging Construct: OWLOntology 234

16.3.2 Class to Topic 235
16.3.3 Class Identified by URI 235
16.3.4 Restriction to Topic 235
16.3.5 Individual to Topic 236
16.3.6 Hierarchy: RDFSsubclassOf 236
16.3.7 Object Property to Association Type 237
16.3.8 Object Property Instance Statement to Association Instance 237
16.3.9 Datatype Property to Occurrence 238
16.3.10 Datatype Property Instance Statement to Occurrence 238
16.3.11 SameAs, EquivalentClass, EquivalentProperty 239

17 Mapping RDFS and OWL to CL 241

17.1 Overview 241

17.2 RDFS to CL Mapping 241

17.2.1 RDF Triples 241
17.2.2 RDF Literals 242
17.2.3 RDF URIs and Graphs 242
17.2.4 RDF Lists 243
17.2.5 RDF Schema 243
17.2.6 RDFS Semantics 244

17.3 OWL to CL Mapping 246

17.4 RDFS to CL Mapping in MOF QVT 256

18 References (non-normative) 259

Annex A - Foundation Library (M1) for RDF and OWL 261

Annex B - Conceptual Entity Relationship Modeling 273

Annex C - A Description Logic Metamodel 279

Annex D - Mapping UML to OWL 293

Annex E - Mappings - Informative, Not Normative 341

Annex F - The Relationship of the Business Nomenclature Metamodel to the ODM 343

Annex G - MOF QVT: A Brief Tutorial 347

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Formal Specifications are available from this URL:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

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

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

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

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

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

Issues

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

1 Scope

The authors believe that this specification represents the foundation for an extremely important set of enabling capabilities for Model Driven Architecture (MDA) based software engineering, namely the formal grounding for representation, management, interoperability, and application of business semantics.

The ODM specification offers a number of benefits to potential users, including:

- Options in the level of expressivity, complexity, and form available for designing and implementing conceptual models, ranging from familiar UML and ER methodologies to formal ontologies represented in description logics or first order logic.
- Grounding in formal logic, through standards-based, model-theoretic semantics for the knowledge representation languages supported, sufficient to enable reasoning engines to understand, validate, and apply ontologies developed using the ODM.
- Profiles and mappings sufficient to support not only the exchange of models developed independently in various formalisms but to enable consistency checking and validation in ways that have not been feasible to date.
- The basis for a family of specifications that marry MDA and Semantic Web technologies to support semantic web services, ontology and policy-based communications and interoperability, and declarative, policy-based applications in general.

The specification defines a family of independent metamodels, related profiles, and mappings among the metamodels corresponding to several international standards for ontology and Topic Maps definition, as well as capabilities supporting conventional modeling paradigms for capturing conceptual knowledge, such as entity-relationship modeling.

The ODM is applicable to knowledge representation, conceptual modeling, formal taxonomy development and ontology definition, and enables the use of a variety of enterprise models as starting points for ontology development through mappings to UML and MOF. ODM-based ontologies can be used to support:

- interchange of knowledge among heterogeneous computer systems,
- representation of knowledge in ontologies and knowledge bases,
- specification of expressions that are the input to or output from inference engines.

The ODM is not intended to encompass

- specification of proof theory or inference rules,
- specification of translation and transformations between the notations used by heterogeneous computer systems,
- free logics,
- conditional logics,
- methods of providing relationships between symbols in the logical “universe” and individuals in the “real world,”
- issues related to computability using the knowledge representation formalisms represented in the ODM (e.g., optimization, efficiency, tractability, etc.).

2 Conformance

There are several compliance points distinguished for the Ontology Definition Metamodel. These include:

1. **None** or **Not Compliant**, meaning that the application in question is not compliant with a particular metamodel, as defined by the metamodel itself, the abstract syntax, well-formedness rules, semantics, and notation specified for a particular package or set of packages.
2. **Compliant**, meaning that the implementation fully complies with the abstract syntax, well-formedness rules, semantics and notation of a particular package or set of packages.
3. **Interchange**, indicating that the implementation provides compliance as specified in [2], *and* can exchange metamodel instances using ODM package conformant XMI.

There are several possible entry points for implementations that want to provide/claim minimal compliance with the ODM. These require compliance with one of the following base metamodel packages:

- RDFBase Metamodel Package (RDFBase is a sub package of the Resource Description Framework (RDF) Metamodel Package).
- Topic Maps (TM) Metamodel Package.
- Common Logic (CL) Metamodel Package.

For a given implementation to claim ODM compliance, it must be **Compliant**, as defined in 2, above, with one of these three packages.

There are several compliance options available to vendors for the RDF Metamodel Package. These include:

- RDFBase Only - as implied above, this package contains the set of elements required for core RDF support, such as is necessary to support a triple store implementation; the focus here is on the set of constructs defined in the RDF Concepts and Abstract Syntax [RDF Concepts] document.
- RDFBase + RDFWeb - provides core RDF support and fits these concepts to the World Wide Web.
- RDFBase + RDFS - moves the implementation focus from core RDF to RDF Schema, as specified in [RDF Schema].
- RDF - meaning, the implementation supports all of the concepts defined in the three sub packages, which represents RDF Schema fitted to the Web.

There are two possible compliance points for the OWL Metamodel Package. Each of these requires support for the entire RDF package, including the RDFWeb component. They include:

- OWLBase + OWLDL - focus is on a description logics application that constrains an ontology in turn for DL decidability.
- OWLBase + OWLFull - focus is on more expressive applications rather than on decidability of entailment.

The complete set of ODM compliance options is summarized in Table 2.1.

Note: The mapping clauses of the specification are informative.

Table 2.1 - Summary of Compliance Points

Compliance Point	Valid Options
RDFBase Only	None, Compliant, Interchange
RDFBase + RDFWeb	None, Compliant, Interchange
RDFBase + RDFS	None, Compliant, Interchange
RDF (Full)	None, Compliant, Interchange
OWLBase + OWLDL for the Semantic Web (requires RDF)	None, Compliant, Interchange
OWLBase + OWLFull (requires RDF)	None, Compliant, Interchange
CL Metamodel	None, Compliant, Interchange
Topic Maps Metamodel	None, Compliant, Interchange
UML Profile for RDF	None, Compliant, Interchange
UML Profile for OWL (requires UML Profile for RDF)	None, Compliant, Interchange
UML Profile for Topic Maps	None, Compliant, Interchange
Mapping from UML to OWL	None, Compliant (unidirectional, bidirectional)
Mapping from Topic Maps to OWL	None, Compliant (unidirectional, bidirectional)
Mapping from RDFS and OWL to CL	None, Compliant

3 Normative References

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

[ISO 646] ISO/IEC 646:1991, Information technology -- ISO 7-bit coded character set for information interchange.

[ISO 2382] ISO/IEC 2382-15:1999, Information technology -- Vocabulary -- Part 15: Programming languages.

[ISO 10646] ISO/IEC 10646:2003, Information technology -- Universal Multiple-Octet Coded Character Set (UCS).

[ISO 14977] ISO/IEC 14977, Information technology -- Syntactic metalanguage -- Extended BNF.

[ISO 24707] ISO/IEC FDIS 24707:2007(E) Information technology – Common Logic (Common Logic) – A framework for a family of logic-based languages. Available at http://cl.tamu.edu/.

[MOF] Meta Object Facility (MOF) Core Specification, Version 2.4.1. OMG Specification, formal/2013-06-01. Latest version is available at http://www.omg.org/spec/MOF/2.4.1/.

[MOF QVT] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0, formal/08-04-03. Available at http://www.omg.org/spec/QVT/1.0/.

[MOF XMI] MOF 2.0/XMI (XML Metadata Interchange) Mapping Specification, v2.1.1. OMG Available Specification, formal/07-12-01. Available at http://www.omg.org/docs/formal/07-12-01.pdf.

[OCL] Object Constraint Language. OMG Available Specification Version 2.0, formal/2006-05-01. Available at http://www.omg.org/docs/formal/06-05-01.pdf

[OWL S&AS] OWL Web Ontology Language Semantics and Abstract Syntax. W3C Recommendation 10 February 2004, Peter F. Patel-Schneider, Patrick Hayes, Ian Horrocks, eds. Latest version is available at http://www.w3.org/TR/owl-semantics/.

[RDF Concepts] Resource Description Framework (RDF): Concepts and Abstract Syntax. Graham Klyne and Jeremy J. Carroll, Editors. W3C Recommendation, 10 February 2004. Latest version is available at http://www.w3.org/TR/rdf-concepts/.

[RDF MIME Type] *MIME Media Types*, The Internet Assigned Numbers Authority (IANA). This document is http://www.iana.org/assignments/media-types/. The registration for application/rdf+xml is archived at http://www.w3.org/2001/sw/RDFCore/mediatype-registration.

[RDF Primer] RDF Primer. Frank Manola and Eric Miller, Editors. W3C Recommendation, 10 February 2004. Latest version is available at http://www.w3.org/TR/rdf-primer/.

[RDF Schema] RDF Vocabulary Description Language 1.0: RDF Schema. Dan Brickley and R.V. Guha, Editors. W3C Recommendation, 10 February 2004. Latest version is available at http://www.w3.org/TR/rdf-schema/.

[RDF Semantics] RDF Semantics. Patrick Hayes, Editor, W3C Recommendation, 10 February 2004. Latest version available at http://www.w3.org/TR/rdf-mt/.

[RDF Syntax] RDF/XML Syntax Specification (Revised). Dave Beckett, Editor, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/. Latest version available at http://www.w3.org/TR/rdf-syntax-grammar/.

[RFC2396] IETF (Internet Engineering Task Force) RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax, eds. T. Berners-Lee, R. Fielding, L. Masinter. August 1998.

[RFC2732] *RFC 2732 - Format for Literal IPv6 Addresses in URL's*, R. Hinden, B. Carpenter and L. Masinter, IETF, December 1999. This document is http://www.isi.edu/in-notes/rfc2732.txt.

[RFC3066] *RFC 3066 - Tags for the Identification of Languages*, H. Alvestrand, The Internet Society, January 2001. This document is http://www.isi.edu/in-notes/rfc3066.txt.

[TMDM] ISO/IEC 13250-2: Topic Maps – Data Model, 2005-12-16. Latest version is available at http://www.isotopicmaps.org/sam/sam-model/.

[UML2] [UML2] Unified Modeling Language: Superstructure, version 2.4.1. OMG Specification, formal/2011-08-06. Available at http://www.omg.org/spec/UML/2.4.1/.

[Unicode] *The Unicode Standard, Version 3*, The Unicode Consortium, Addison-Wesley, 2000. ISBN 0-201-61633-5, as updated from time to time by the publication of new versions. (See http://www.unicode.org/unicode/standard/versions/ for the latest version and additional information on versions of the standard and of the Unicode Character Database).

[XLINK] XML Linking Language (XLink) Version 1.0, W3C Recommendation 27 June 2001, http://www.w3.org/TR/xlink/.

[XML Schema Datatypes]	XML Schema Part 2: Datatypes. W3C Recommendation 02 May 2000. Latest version is available at http://www.w3.org/TR/xmlschema-2/.
[XMLNS]	Namespaces in XML; W3C Recommendation, 14 January 1999. Latest version is available at http://www.w3.org/TR/1999/REC-xml-names-19990114/.
[XTM]	ISO/IEC FCD 13250-3: Topic Maps – XML Syntax, 2006-05-02. Latest version is available at http://www.isotopicmaps.org/sam/sam-xtm/.

4 Terms and Definitions

Complete MOF (CMOF)

The CMOF, or Complete MOF, Model is the model used to specify other metamodels such as UML2. It is built from EMOF and the Core::Constructs of UML. The CMOF package does not define any classes of its own. Rather, it merges packages with its extensions that together define basic metamodeling capabilities.

Common Logic (CL)

Common Logic is a first order logic framework intended for information exchange and transmission. The framework allows for a variety of different syntactic forms, called dialects, all expressible within a common XML-based syntax and all sharing a single semantics.

Computation Independent Model (CIM)

A computation independent model is a view of a system from the computation independent viewpoint. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model, and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification. Some ontologies are essentially CIMs from a software engineering perspective.

Description Logics (DL)

Description logics are knowledge representation languages tailored for expressing knowledge about concepts and concept hierarchies, and typically represent a decidable subset of traditional first order logic. Description logic systems have been used for building a variety of applications including conceptual modeling, information integration, query mechanisms, view maintenance, software management systems, planning systems, configuration systems, and natural language understanding. The Web Ontology Language (OWL) is a member of the description logics family of knowledge representation languages.

Entity-Relationship (ER)

An ER (entity-relationship) diagram is a graphical modeling notation that illustrates the interrelationships between entities in a domain. ER diagrams often use symbols to represent three different types of information. Boxes are commonly used to represent entities. Diamonds are normally used to represent relationships and ovals are used to represent attributes.

Essential MOF (EMOF)

Essential MOF is the subset of MOF that most closely corresponds to the facilities found in object-oriented programming languages and in XML. It provides a straightforward framework for mapping MOF models to implementations such as JMI and XMI for simple metamodels. A primary goal of EMOF is to allow simple metamodels to be defined using simple concepts while supporting extensions (by the usual class extension mechanism in MOF) for more sophisticated metamodeling using CMOF.

interpretation

A relationship between individuals in a universe of discourse and the symbols and relations in a model such that the model expresses truths about the individuals.

Knowledge Interchange Format (KIF)

Knowledge Interchange Format (KIF) is a computer-oriented language for the interchange of knowledge among disparate systems. It has declarative semantics (i.e., the meaning of expressions in the representation can be understood without appeal to an interpreter for manipulating those expressions); it is logically comprehensive (i.e., it provides for the expression of arbitrary sentences in the first-order predicate calculus); it provides for the representation of knowledge about the representation of knowledge; it provides for the representation of non-monotonic reasoning rules; and it provides for the definition of objects, functions, and relations. KIF was developed in the late 1980s and early 1990s through support of the DARPA Knowledge Sharing Effort. There are several "flavors" of KIF in use today, including the best known versions: ANSI KIF (i.e., Knowledge Interchange Format dpANS, NCITS.T2/98-004, http://logic.stanford.edu/kif/dpans.html) and KIF Reference (i.e., Version 3.0 of the KIF Reference Manual, http://www-ksl.stanford.edu/knowledge-sharing/papers/kif.ps). For the purpose of this ODM specification, references to KIF should be considered references to the KIF 3.0 Reference Manual cited in the Non-normative References sub clause of this specification.

Meta-Object Facility (MOF)

The Meta Object Facility (MOF), an adopted OMG standard, provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems. Examples of these systems that use MOF include modeling and development tools, data warehouse systems, metadata repositories etc. For the purpose of this ODM specification, references to MOF should be considered references to the Meta-Object Facility 2.0 Core Specification, cited in Normative References, above.

Object Constraint Language (OCL)

The Object Constraint Language (OCL), an adopted OMG standard, is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects; i.e., their evaluation cannot alter the state of the corresponding executing system. For the purpose of this ODM specification, references to OCL should be considered references to the UML 2.0 Object Constraint Language Specification, cited in Normative References, above.

Ontology Definition Metamodel (ODM)

The Ontology Definition Metamodel (ODM), as defined in this specification, is a family of MOF metamodels, mappings between those metamodels as well as mappings to and from UML, and a set of profiles that enable ontology modeling through the use of UML-based tools. The metamodels that comprise the ODM reflect the abstract syntax of several standard knowledge representation and conceptual modeling languages that have either been recently adopted by other international standards bodies (e.g., RDF and OWL by the W3C), are in the process of being adopted (e.g., Common Logic and Topic Maps by the ISO) or are considered industry de facto standards (non-normative ER and DL appendices).

Platform Independent Model (PIM)

A *platform independent model* is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type. Examples of platforms range from virtual machines, to programming languages, to deployment platforms, to applications, depending on the perspective of the modeler and application being modeled.

Platform Specific Model (PSM)

A *platform specific model* is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.

Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a framework for representing information in the Web. RDF has an abstract syntax that reflects a simple graph-based data model, and formal semantics with a rigorously defined notion of entailment providing a basis for well founded deductions in RDF data. The vocabulary is fully extensible, being based on URIs with optional fragment identifiers (URI references, or URIrefs). For the purpose of this ODM specification, references to RDF should be considered references to the set of RDF recommendations available from the World Wide Web Consortium, and in particular, the RDF Concepts and Abstract Syntax recommendation, cited in Normative References, above.

RDF Schema (RDFS)

RDF's vocabulary description language, RDF Schema, is a semantic extension of RDF. It provides mechanisms for describing groups of related resources and the relationships between these resources. These resources are used to determine characteristics of other resources, such as the domains and ranges of properties. The RDF vocabulary description language class and property system is similar to the type systems of object-oriented programming languages such as Java. RDF differs from many such systems in that instead of defining a class in terms of the properties its instances may have, the RDF vocabulary description language describes properties in terms of the classes of resource to which they apply. For the purpose of this ODM specification, references to RDF Schema should be considered references to the set of RDF recommendations available from the World Wide Web Consortium, and in particular, the RDF Vocabulary Description Language 1.0: RDF Schema recommendation, cited in Normative References, above.

Topic Maps (TM)

Topic Maps provide a model and grammar for representing the structure of information resources used to define topics, and the associations (relationships) between topics. Names, resources, and relationships are said to be characteristics of abstract subjects, which are called topics. Topics have their characteristics within scopes: i.e., the limited contexts within which the names and resources are regarded as their name, resource, and relationship characteristics. One or more interrelated documents employing this grammar is called a “topic map.” For the purpose of this ODM specification, references to Topic Maps should be considered references to the draft ISO standard cited in Normative References, above.

traditional first order logic

The traditional algebraic (or mathematical) formulations of logic generally described by Russell, Whitehead, Peano, and Pierce, dealing with quantification, negation, and logical relations as expressed in propositions that are strictly true or false. This specifically excludes reasoning over relations and excludes using the same name as both an individual name and a relation name.

Unified Modeling Language (UML)

The Unified Modeling Language, an adopted OMG standard, is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecommunications, aerospace) and implementation platforms (e.g., J2EE, .NET). For the purpose of this ODM specification, references to UML should be considered references to the Unified Modeling Language 2.0 Infrastructure and Superstructure Specifications, cited in Normative References, above.

universe of discourse

A non-empty set over which the quantifiers of a logic language are understood to range. Sometimes called a “domain of discourse.”

Web Ontology Language (OWL)

The OWL Web Ontology Language is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL can be used to explicitly represent the meaning of terms in vocabularies and the relationships between those terms. This representation of terms and their interrelationships is called an ontology. OWL has more facilities for expressing meaning and semantics than XML, RDF, and RDF-S, and thus OWL goes beyond these languages in its ability to represent machine interpretable content on the Web. OWL has three increasingly-expressive sub-languages: OWL Lite, OWL DL, and OWL Full. For the purpose of this ODM specification, references to OWL should be considered references to the set of OWL recommendations available from the World Wide Web Consortium, and in particular, the OWL Web Ontology Language Semantics and Abstract Syntax recommendation, cited in Normative References, above.

XML Metadata Interchange (XMI)

XMI is a widely used interchange format for sharing objects using XML. Sharing objects in XML is a comprehensive solution that build on sharing data with XML. XMI is applicable to a wide variety of objects: analysis (UML), software (Java, C++), components (EJB, IDL, CORBA Component Model), and databases (CWM). For the purpose of this ODM specification, references to XMI should be considered references to the XML Metadata Interchange (XMI) 2.0 Specification, cited in Normative References, above.

eXtended Markup Language (XML)

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. RDF and OWL build on XML as a basis for representing business semantics on the Web. Relevant W3C recommendations are cited in the RDF and OWL documents as well as those cited under Normative References, above.

5 Symbols

CIM	Computation Independent Model
CL	Common Logic
DL	Description Logics
ER	Entity-Relationship
FOL	First Order Logic
IRI	Internationalized Resource Identifier
ISO/IEC	International Organization for Standardization / International Electrotechnical Commission
KIF	Knowledge Interchange Format
MDA	Model Driven Architecture
MOF	Meta-Object Facility 2.0
OCL	UML 2.0 Object Constraint Language
ODM	Ontology Definition Metamodel
OMG	Object Management Group
OWL	Web Ontology Language
OWL DL	The Description Logics dialect of OWL
OWL Full	The most expressive dialect of OWL
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query / View / Transformation
RDF	Resource Description Framework

RDFS	RDF Schema
RFP	Request for Proposal
SW	Semantic Web
TFOL	Traditional First Order Logic
TM	Topic Maps
UML	Unified Modeling Language 2.0
URI	Uniform Resource Identifier
XMI	XML Metadata Interchange
XML	eXtended Markup Language

6 Additional Information

6.1 Changes to Adopted OMG Specifications

In the UML 2.0 Specification [UML2], sub clause 9.10.1, an instance specification is explicitly defined as having one or more classifiers: "If multiple classifiers are specified, the instance is classified by all of them."

MOF 2.0 [MOF] reuses and extends certain core packages from the UML infrastructure library, including the Core::Abstractions package, wherein instance specification is defined. Essential MOF (EMOF) merges the Core::Basic package, including the definition of instance specification from the Core::Abstractions package in UML Infrastructure, with several new capabilities, including MOF Reflection. Section 12.1 in the MOF 2.0 specification explicitly reuses this definition, by stating "EMOF reuses the Basic package from UML 2.0 Infrastructure Library as is for metamodel structure without any extensions, although it does introduce some constraints." and in 12.2 "The description of the model elements is identical to that found in UML 2.0 Infrastructure and is not repeated here." The set of constraints introduced in section 12.4 makes no mention of instances or instance specification, thus, the metamodel structure reused from UML is unchanged with regard to instances and, in particular, their definition with regard to multiple classification.

There are at least three places in the MOF specification that do not support runtime instances modeled by instance specifications, as described above, where:

- The Semantic Domain model for Constructs, Figure 15.1 in the MOF specification, omits the relationship between an InstanceSpecification and its classifier(s).
- In the same figure, the classifier association end from ClassInstance to Class has a multiplicity of 1 instead of 1..*.
- The operation, Reflection::Element::getMetaClass() : Class is single-valued.

This is an issue for the specification of the set of metamodels defined herein, in particular, for the RDF and OWL metamodels. Specifics are noted in the text of Clause 10, The RDF Metamodel, and in Clause 11, The OWL Metamodel.

The authors consider this to be a problem in the MOF specification, as documented in issue #9466, and expect it to be addressed in future revisions of MOF, notably, through the emerging "SMOF" or Semantic MOF RFP. As a result, the normative metamodels contained herein presume support in MOF (SMOF) for multiple classification. Annex F includes work-arounds for the issues we have uncovered related to this problem, however, for those vendors who choose to implement the ODM before this problem is adequately addressed in a subsequent version of the MOF specification.

6.2 How to Read This Specification

The initial nine clauses of this specification are *informative*, providing discussion related to how the specification addresses the RFP requirements (this clause), a high-level summary of usage scenarios and goals (Clause 7), design rationale (Clause 8), and the overall structure of the set of metamodels that comprise the Ontology Definition Metamodel (Clause 9).

Clause 10 describes a set of MOF metamodels for developing Resource Description Framework (RDF) vocabularies *(normative)*.

Clause 11 describes a set of MOF metamodels for developing Web Ontology Language (OWL) ontologies *(normative)*.

Clause 12 describes a MOF metamodel for developing more expressive, first-order logic (FOL) ontologies in the Common Logic (CL) family of languages *(normative)*.

Clause 13 describes a MOF metamodel for developing Topic Maps *(normative)*.

Clause 14 describes UML Profiles for RDF and OWL *(normative)*.

Clause 15 describes a UML Profile for Topic Maps *(normative)*.

Clause 16 provides a mapping between Topic Maps and OWL using MOF QVT *(informative)*.

Clause 17 provides an embedding from RDF and OWL in CL *(informative)*.

Clause 18 contains non-normative references to other work.

Annex A describes model library elements (M1) necessary for use with the RDF and OWL metamodels, and related but independent library elements for use with the RDF and OWL profiles. *(normative)*.

Annex B describes extensions to UML to support Conceptual ER Modeling *(informative)*.

Annex C describes a MOF metamodel for general Description Logics *(informative)*.

Annex D provides a mapping between UML and OWL using MOF QVT *(informative)*.

Annex E discusses issues related to making mappings informative rather than normative components of this specification *(informative)*.

Annex F discusses the relationship between the ODM and Business Nomenclature (*informative*).

Annex G provides a short tutorial on the use of MOF QVT and its application in this specification (*informative*).

6.3 Proof of Concept

DSTC Pty Ltd. carried out a seven year research programme into Enterprise Distributed Systems Technology with major projects devoted to knowledge representation. DSTC Pty Ltd. had extensive experience in the standardization, implementation and use of MOF, XMI and UML. The DSTC developed MOF-based tools from 1996 until June 2005. DSTC developed the following prototypes to validate parts of this specification:

- Web-KB is a non-MOF-based implementation of many of the concepts represented in this specification. It is available for live demonstration on the Internet at www.webkb.org.
- Parts of the model presented in this specification were implemented using DSTC's dMOF product (MOF 1.3) and DSTC's TokTok product (HUTN 1.0) to validate the expressive power of the model.

IBM has developed the following tools which in part validate portions of this specification:

- IBM Semantics Toolkit is a toolkit for storage, manipulation, query, and inference of ontologies and corresponding instances. It is available for download at http://alphaworks.ibm.com/tech/semanticstk.
- EODM is a tool for manipulation of and inference over OWL ontologies and RDF vocabularies, using EMF-based Java APIs generated from the OWL and RDFS metamodels. It is available for download at http://alphaworks.ibm.com/tech/semanticstk. EODM was released as an open-source Eclipse Modeling Framework subproject (contact: xieguot@cn.ibm.com), which may be replaced by a new ODM project in the MDT family of Eclipse projects later this year.

Sandpiper Software has been developing technologies and tools to support UML-based knowledge representation since 1999. Sandpiper has developed the following products that validate parts of this specification:

- Visual Ontology Modeler (VOM) v1.5 is a UML 1.x/MOF 1.x compliant add-in to IBM Rational Rose, enabling component-based ontology modeling in UML with support for forward and reverse engineering of OWL ontologies.
- Next generation support for UML2, MOF2, and ODM compliance for RDFS/OWL and CL ontologies, and a CL constraint editor are under development, including migration to Eclipse/EMF, IBM Rational Software Architect (RSA), and integration with other UML2-compliant modeling environments such as No Magic's MagicDraw tool.
- The ODM metamodels and profiles reflected in this ODM 1.0 specification will be provided as a basis for open source development to the new Eclipse MDT ODM project later this year (2008).

6.4 Acknowledgments

The following companies initially submitted this specification to the OMG:

- IBM
- Sandpiper Software, Inc.

The following additional companies and organizations made substantial contributions to the revision of this specification:

- 88Solutions
- ACORD
- Adaptive, Inc.
- California Institute of Technology (NASA/JPL)
- Computer Sciences Corporation
- Deere & Company
- Institute for Defense Analyses
- No Magic, Inc.
- Raytheon Company
- Sparx Systems Pty Ltd
- Thematix Partners LLC
- U.S. National Institute of Standards and Technology (NIST)

The following additional companies and organizations support this specification:

- AT&T Government Solutions
- Commissariat a l'Energie Atomique (CEA)

- Consultative Committee for Space Data systems (CCSDS)
- Data Access Technologies / Model Driven Solutions
- DSTC Pty. Ltd.
- Florida Institute for Human and Machine Cognition (IHMC)
- France Telecom
- Freie Universitat Berlin
- Gentleware AG
- Hewlett-Packard Company
- Honeywell International Inc.
- Hyperion
- IKAN Group
- Institut AIFB, Universität Karlsruhe (TH)
- Mercury Computer Systems
- MetaMatrix
- MetLife
- Stanford University, Knowledge Systems Laboratory (KSL)
- Tokyo Electric Power Company
- UMTP
- University of Aberdeen

7 Usage Scenarios and Goals

7.1 Introduction

The usage scenarios presented in this clause highlight characteristics of ontologies that represent important design considerations for ontology-based applications. They also motivate some of the features and functions of the ODM and provide insight into when users can limit the expressivity of their ontologies to a description logics based approach, as well as when additional expressivity, for example from first order logic, might be needed. This set of examples is not intended to be exhaustive. Rather, the goal is to provide sufficiently broad coverage of the kinds of applications the ODM is intended to support so that ODM users can make informed decisions when choosing what parts of the ODM meet their development requirements and goals.

This analysis can be compared with a similar analysis performed by the W3C Web Ontology Working Group (W3C 2003). We believe that the six use cases and eight goals considered in W3C (2003) provide additional, and in some cases overlapping, examples, usage scenarios and goals for the ODM.

7.2 Perspectives

In order to ensure a relatively complete representation of usage scenarios and their associated example applications, we evaluated the coverage by using a set of perspectives that characterize the domain. Table 7.1 provides an overview of these perspectives.

Table 7.1 - Perspectives of Applications that Use Ontologies Considered in this Analysis

Perspective	One Extreme	Other Extreme
Level of Authoritativeness	Least authoritative, broader, shallowly defined ontologies	Most authoritative, narrower, more deeply defined ontologies
Source of Structure	Passive (Transcendent) – structure originates outside the system	Active (Immanent) – structure emerges from data or application
Degree of Formality	Informal, or primarily taxonomic	Formal, having rigorously defined types, relations, and theories or axioms
Model Dynamics	Read-only, ontologies are static	Volatile, ontologies are fluid and changing.
Instance Dynamics	Read-only, resource instances are static	Volatile, resource instances change continuously
Control / Degree of Manageability	Externally focused, public (little or no control)	Internally focused, private (full control)
Application Changeability	Static (with periodic updates)	Dynamic
Coupling	Loosely-coupled	Tightly-coupled
Integration Focus	Information integration	Application integration
Lifecycle Usage	Design Time	Run Time

An ontology is a specification of a conceptualization for some area; there may be distinct ontologies representing differing conceptualizations of the same domain. Ontologies may also differ due to the cost-benefit trade-offs associated with different specifications. The perspectives associated with the conceptualizations are called *model centric*.

An ontology can also be used in a software development process in different ways. The perspectives that reflect how an ontology participates in the software development process are called *application centric*.

7.2.1 Model-Centric Perspectives

The model centric perspectives characterize the ontologies themselves and are concerned with the structure, formalism, and dynamics of the ontologies; they are:

- Level of Authoritativeness
- Source of Structure
- Degree of Formality
- Model Dynamics
- Instance Dynamics

Level of Authoritativeness

The conceptualization from which an ontology is developed is always produced by someone. If the ontology is developed by the organization that is responsible for specifying the conceptualization, then it may be definitive, and therefore *highly authoritative*. If ontology development is distant from the organization defining the conceptualization, it *may not be very authoritative*.

Highly authoritative ontologies are typically part of the institutional fabric of the organizations that will use them. If the conceptualization is complex, it often pays to develop the specification in great depth. But if the authority of the responsible institution is limited, the specification will generally have sharp boundaries and may be relatively narrow. Ontologies that are not authoritative tend to be broad, since the creator can pick the most accessible concepts from many conceptualizations, and generally not very deep. Such ontologies may not be reliable from a user perspective, so may not attract sufficient resources to be developed in detail.

SNOMED[1] is a very large and authoritative ontology. The periodic table of the elements is very authoritative, but small. However, it can be safely used as a component of larger ontologies in physics or chemistry. Ontologies used for demonstration or pedagogic purposes, like the Wine Ontology[2], are not very authoritative. Table 7.1 can be seen as an ontology which at present is not very authoritative. Should the classifications gain wide use in the ontology community, the ontology in Table 7.1 would become more authoritative.

Source of Structure

An ontology describes a structure that may be implemented in software of some kind. In some cases, the structure represents published rules of engagement, required for interoperability, that can only be revised by authorized agents in a well-publicized manner. In other words, the ontology is developed externally to the applications that use it; and changes are made systematically, through a published revision process. Such an ontology is called *transcendent*. SNOMED is a transcendent ontology defined by the various governing bodies of medicine. E-commerce exchanges are generally supported by transcendent ontologies.

---

1. http://www.snomed.org
2. http://www.w3.org/2001/sw/WebOnt/guide-src/wine.owl

Alternatively, the structure may be defined by patterns arising from content knowledge, instantiated or inferred by applications. An ontology that emerges from content is called *immanent*. Examples include ontologies used by data mining and analysis applications, such as financial or market analysis tools that process news feeds. The set of topics extracted from such news feeds might define the structure of the ontology, and although changes in structure of specific topics may be minor, new topics can introduce radical structure change. A company may hire a new executive, but its management structure remains constant, for example. The outbreak of a war would introduce radical change, as can the introduction of a new technology like the World-Wide Web or mobile telephones. Other applications using similar capabilities include customer relationship management applications, such as that used by Amazon, search, and security applications, such as those used to detect unusual patterns of credit card activity that may indicate fraudulent use.

Degree of Formality

Degree of formality refers to the level of formality of the specification of the conceptualization, ranging from *highly informal* or taxonomic in nature, to semantic networks, that may include complex subclass/superclass relations but no formal axiom expressions, to ontologies containing *highly formal* axioms that explicitly define concepts. SNOMED is taxonomic, as is the Standard Industrial Classification system (SIC) used by the US Department of Labor Statistics, while engineering ontologies like Gruber and Olsen (1994) are highly formal.

Model Dynamics

All ontologies have structure, which likely evolves over time. If the ontology is transcendent, its maintaining organization may decide to make a change; for immanent ontologies, new patterns may emerge from the data. The question is, how often does the structure change? One extreme in the model dynamics dimension is *stable* or *read-only* ontologies that rarely change in structure. The Periodic Table is structurally stable, as are generally the rules of an engagement. SNOMED is relatively stable, as is the SIC (the SIC is slowly being replaced by the North American Industry Classification System (NAICS) after 60 years, due to changes in the American economy in that period).

The other extreme in model dynamics is ontologies whose structure is *volatile,* changing often. An ontology supporting tax accounting in Australia would be volatile at the model level, since the system of taxation institutional facts can change, sometimes quite radically, with any Budget.

Instance Dynamics

An ontology often includes a system of classes and properties (the structure), populated by instances (the extents). As with model dynamics, instance knowledge may be *stable* (*read-only)* or *volatile*. The Periodic Table is stable at the instance level (i.e., particular elements) as well as the model level (e.g., noble gasses or rare earths). New elements are possible but rarely discovered. On the other hand, an ontology supporting an e-commerce exchange is volatile at the instance level but possibly not at the model level. Z39.50 based applications are very stable in their model dynamics, but volatile in their instance dynamics. Libraries are continually turning over their collections.

7.2.2 Application-Centric Perspectives

Application centric perspectives are concerned with how applications use and manipulate the ontologies, they are:

- Control / Degree of Manageability
- Application Changeability
- Coupling
- Integration Focus
- Lifecycle Usage

Control / Degree of Manageability

This dimension considers who decides when and how much change to make to an ontology. One extreme is when the body responsible for the ontology has sole decision on change (*internally focused*). The SIC is internally focused. Change is required because the structure of the US economy has changed over the years, but the Bureau of Labor Statistics decides how and when change is introduced.

The other extreme is when changes to the ontology are mandated by outside agencies (*externally focused*). In the US, ontologies in the finance industry were required to change by the Sarbanes-Oxley Act of 2002, and changes in ontologies in many areas were mandated by the Patriot Act, passed shortly after the World Trade Center attacks in 2001. An ontology on taxation matters managed by a trade association of accountants is subject to change as the relevant taxation acts are changed by governments.

Application Changeability

An ontology may be used in applications. The applications may be developed once, as for an e-commerce exchange (*static*) with periodic updates. On the other extreme, applications may be constructed dynamically (*dynamic*), as in an application that composes web services at run time.

Coupling

This dimension describes how closely coupled applications committed to shared ontologies are to each other. The applications in an e-commerce exchange are *tightly coupled*, since they must interoperate at run time. At the other extreme, applications using the Periodic Table or the Engineering Mathematics ontology may have nothing in common at run time. They are *loosely coupled*, solely because they share a component.

Integration Focus

Some ontologies specify the structure of interoperation but not content. Z39.50 exclusive of the use attribute sets is a good example. The MPEG-21 multimedia framework is another example. It specifies the structure of multimedia objects without regard for their content. This extreme is called *application integration*, because they can be used to link programs together so that the output of one is a valid input for the other.

Other ontologies specify content structure. An ontology may specify the structure of a shared knowledge base, for example, for use by agents that exchange information about shared objects. This extreme is called *information integration*. Ontologies used for integration may be both application and information focused.

Lifecycle Usage

An ontology may be used by an application in the specification or design phases of the software life cycle, but not explicitly at run time. Use of the Periodic Table or Engineering Mathematics ontology in the specification of an engineering or scientific application is an example of *design time* usage. In a large e-commerce exchange, the exchange may check every message to see whether it conforms to the ontology and if so, what version. The message is then sent to the recipient with a certification, therefore relieving the players from having to do the checks themselves. In this case, the ontology is used at *run time*.

7.3 Usage Scenarios

As might be expected, some perspectives tend to correlate, forming application areas with similar characteristics. Our analysis, summarized in Table 7.2, identified three major clusters of application types that share perspective values:

- Business Applications have transcendent source of structure, a high degree of formality and external control relative to nearly all users.

- Analytic Applications have highly changeable and flexible ontologies, using large collections of mostly read-only instance data.
- Engineering Applications have transcendent source of structure, but users control them primarily internally and they are considered more authoritative.

Table 7.2 - Usage Scenario Perspective Values

Use Case Clusters		**Characteristic Perspective Values**									
		Model Centric					Application Centric				
	Description	Authoritative	Structure	Formality	Model Dynamics	Instance Dynamics	Control	Change	Coupling	Focus	Life Cycle
7.4	Business Applications		Transcendent	Formal			External				
7.4.1	Run-time Interoperation	Least/Broad	Transcendent	Formal	Read-Only	Volatile	External	Static	Tight	Information	Real Time
7.4.2	Application Generation	Most/Deep	Transcendent	Formal	Read-Only	Read-Only	External	Static	Loose	Application	All
7.4.3	Ontology Lifecycle	Middle/ Broad& Deep	Transcendent	Semi-Formal/ Formal	Read-Only	Read-Only	External	Static	Tight	Information	Real Time
7.5	Analytic Applications				Volatile	Read-Only		Dynamic	Flexible		
7.5.1	Emergent Property Discovery	Broad & Deep	Immanent	Informal	Volatile	Read-Only	Internal & External	Dynamic	Flexible	Information	Real Time
7.5.2	Exchange of Complex Data Sets	Broad & Deep	Immanent	Informal	Volatile	Read-Only/ Volatile	Internal & External	Dynamic	Flexible	Information	Real Time
7.6	Engineering Application	Broad & Deep	Transcendent				Internal				
7.6.1	Information System Development	Broad & Deep	Transcendent	Semi-Formal / Formal	Read-Only	Volatile	Internal	Evolving	Tight	Information	Design Time
7.6.2	Ontology Engineering	Broad & Deep	Transcendent	Semi-Formal / Formal	Volatile	Volatile	Internal	Evolving	Flexible	???	Design Time

7.4 Business Applications

7.4.1 Run Time Interoperation

Externally focused information interoperability applications are typically characterized by strong de-coupling of the components realizing the applications. They are focused specifically on information rather than application integration (and here we include some semantic web service applications, which may involve composition of vocabularies, services and processes but not necessarily APIs or database schemas). Because the community using them must agree upon the ontologies in advance, their application tends to be static in nature rather than dynamic.

Perspectives that drive characterization of these scenarios include:

- The ontology must be sufficiently authoritative to support the investment.
- Whether the control is external to the community members.
- Whether or not there is a design time component to ontology development and usage
- Whether or not the knowledge bases and information resources that implement the ontologies are modified at run time (since the source of structure remains relatively unchanged in these cases, or the ontologies are only changed in a highly controlled, limited manner).

These applications may require mediation middleware that leverages the ontologies and knowledge bases that implement them, potentially on either side of the firewall – in next generation web services and electronic commerce architectures as well as in other cross-organizational applications, for example:

- For semantically grounded information interoperability, supporting highly distributed, intra- and inter-organizational environments with dynamic participation of potential community members, (as when multiple emergency services organizations come together to address a specific crisis), with diverse and often conflicting organizational goals.
- For semantically grounded discovery and composition of information and computing resources, including Web services (applicable in business process integration and grid computing).

In electronic commerce exchange applications based on state-full protocols such as EDI or Z39.50, where there are multiple players taking roles performing acts by sending and receiving messages whose content refers to a common world.

In these cases, we envision a number of agents and/or applications interoperating with one another using fully specified ontologies. Support for query interoperation across multiple, heterogeneous databases is considered a part of this scenario.

While the requirements for ontologies to support these kinds of applications are extensive, key features include:

- the ability to represent situational concepts, such as player/actor – role – action – object – state,
- the necessity for multiple representations and/or views of the same concepts and relations, and
- separation of concerns, such as separating the vocabularies and semantics relevant to particular interfaces, protocols, processes, and services from the semantics of the domain.
- Service checking that messages commit to the ontology at run time. These communities can have thousands of autonomous players, so that no player can trust any other to send messages properly committed to the ontology.

7.4.2 Application Generation

A common worldview, universe of discourse, or domain is described by a set of ontologies, providing the context or situational environment required for use by some set of agents, services, and/or applications. These applications might be internally focused in very large organizations, such as within a specific hospital with multiple, loosely coupled clinics, but are more likely multi- or cross-organizational applications. Characteristics include:

- Authoritative environments, with tighter coupling between resources and applications than in cases that are less authoritative or involve broader domains, though likely on the “looser side” of the overall continuum.
- Ontologies shared among organizations are highly controlled from a standards perspective, but may be specialized by the individual organizations that use them within agreed parameters.

- The knowledge bases implementing the ontologies are likely to be dynamically modified, augmented at run time by new metadata, gathered or inferred by the applications using them.
- The ontologies themselves are likely to be deeper and narrower, with a high degree of formality in their definition, focused on the specific domain of interest or concepts and perspectives related to those domains.

For example:

- Dynamic regulatory compliance and policy administration applications for security, logistics, manufacturing, financial services, or other industries.
- Applications that support sharing clinical observation, test results, medical imagery, prescription and non-prescription drug information (with resolution support for interaction), relevant insurance coverage information, and so forth across clinical environments, enabling true continuity of patient care.

Requirements:

- The ontologies used by the applications may be fully specified where they interoperate with external organizations and components, but not necessarily fully specified where the interaction is internal.
- Conceptual knowledge representing priorities and precedence operations, time and temporal relevance, bulk domains where individuals don't make sense, rich manufacturing processes, and other complex notions may be required, depending on the domain and application requirements.

7.4.3 Ontology Lifecycle

In this scenario we are concerned with activity, which has as its principle objectives conceptual knowledge analysis, capture, representation, and maintenance. Ontology repositories should be able to support rich ontologies suitable for use in knowledge-based applications, intelligent agents, and semantic web services. Examples include:

- maintenance, storage and archiving of ontologies for legal, administrative and historical purposes,
- test suite generation, and
- audits and controllability analysis.

Ontological information will be included in a standard repository for management, storage, and archiving. This may be to satisfy legal or operations requirements to maintain version histories.

These types of applications require that Knowledge Engineers interact with Subject Matter Experts to collect knowledge to be captured. UML models provide a visual representation of ontologies facilitating interaction. The existence of meta-data standards, such as XMI and ODM, will support the development of tools specifically for Quality Assurance Engineers and Repository Librarians.

Requirements implications:

- Full life-cycle support will be needed to provide managed and controlled progression from analysis, through design, implementation, test and deployment, continuing on through the supported systems maintenance period.
- Part of the lifecycle of ontologies must include collaboration with development teams and their tools, specifically in this case configuration and requirements management tools. Ideally, any ontology management tool will also be ontology aware.
- It will provide an inherent quality assurance capability by providing consistency checking and validation.

- It will also provide mappings and similarity analysis support to integrate multiple internal and external ontologies into a federated web.

7.5 Analytic Applications

7.5.1 Emergent Property Discovery

By this we mean applications that analyze, observe, learn from and evolve as a result of, or manage other applications and environments. The ontologies required to support such applications include ontologies that express properties of these external applications or the resources they use. The environments may or may not be authoritative; the ontologies they use may be specific to the application or may be standard or utility ontologies used by a broader community. The knowledge bases that implement the ontologies are likely to be dynamically augmented with metadata gathered as a part of the work performed by these applications. External information resources and applications are accessed in a read-only mode.

- Semantically grounded knowledge discovery and analysis (e.g., financial, market research, intelligence operations).
- Semantics assisted search of data stored in databases or content stored on the Web (e.g., using domain ontologies to assist database search, using linguistic ontologies to assist Web content search).
- Semantically assisted systems, network, and / or applications management.
- Conflict discovery and prediction in information resources for self-service and manned support operations (e.g., technology call center operations, clinical response centers, drug interaction).

What these have in common is that the ontology is typically not directly expressed in the data of interest, but represents theories about the processes generating the data or emergent properties of the data. Requirements include representation of the objects in the ontology as rules, predicates, queries, or patterns in the underlying primary data.

7.5.2 Exchange of Complex Data Sets

Applications in this class are primarily interested in the exchange of complex (multi-media) data in scientific, engineering, or other cooperative work. The ontologies are typically used to describe the often complex multimedia containers for data, but typically not the contents or interpretation of the data, which is often either at issue or proprietary to particular players. (The OMG standards development process is an example of this kind of application.)

Here the ontology functions more like a rich type system. It would often be combined with ontologies of other kinds (for example, an ontology of radiological images might be linked to SNOMED for medical records and insurance reimbursement purposes).

Requirements include

- Representation of complex objects (aggregations of parts).
- Multiple inheritance where each semantic dimension or facet can have complex structure.
- Tools to assemble and disassemble complex sets of scientific and multi-media data.
- Facilities for mapping ontologies to create a cross reference. These do not need to be at the same level of granularity. For the purposes of information exchange, the lower levels of two ontologies may be mapped to a higher level common abstraction of a third, creating a sort of index.

7.6 Engineering Applications

The requirements for ontology development environments need to consider both externally and internally focused applications, as externally focused but authoritative environments may require collaborative ontology development.

7.6.1 Information Systems Development

The kinds of applications considered here are those that use ontologies and knowledge bases to support enterprise systems design and interoperation. They may include:

- methodology and tooling, where an application actually composes various components and/or creates software to implement a world that is described by one or more component ontologies.
- Semantic integration of heterogeneous data sources and applications (involving diverse types of data schema formats and structures, applicable in information integration, data warehousing and enterprise application integration).
- Application development for knowledge based systems, in general.

In the case of model-based applications, extent-descriptive predicates are needed to provide enough meta-information to exercise design options in the generated software (e.g., describing class size, probability of realization of optional classes). An example paradigm might reflect how an SQL query optimizer uses system catalog information to generate a query plan to satisfy the specification provided by an SQL query. Similar sorts of predicates are needed to represent quality-type meta-attributes in semantic web type applications (comprehensiveness, authoritativeness, currency).

7.6.2 Ontology Engineering

Applications in this class are intended for use by an information systems development team, for utilization in the development and exploitation of ontologies that make implicit design artifacts explicit, such as ontologies representing process or service vocabularies relevant to some set of components. Examples include:

- Tools for ontology analysis, visualization, and interface generation.
- Reverse engineering and design recovery applications.

The ontologies are used throughout the enterprise system development life cycle process to augment and enhance the target system as well as to support validation and maintenance. Such ontologies should be complementary to and augment other UML modeling artifacts developed as part of the enterprise software development process. Knowledge engineering requirements may include some ontology development for traditional domain, process, or service ontologies, but may also include:

- Generation of standard ontology descriptions (e.g., OWL) from UML models.
- Generation of UML models from standard ontology descriptions (e.g., OWL).
- Integration of standard ontology descriptions (e.g., OWL) with UML models.

Key requirements for ontology development environments supporting such activities include:

- Collaborative development.
- Concurrent access and ontology sharing capabilities, including configuration management and version control of ontologies in conjunction with other software models and artifacts at the atomic level within a given ontology, including deprecated and deleted ontology elements.
- Forward and reverse engineering of ontologies throughout all phases of the software development lifecycle.

- Ease of use, with as much transparency with respect to the knowledge engineering details as possible from the user perspective.
- Interoperation with other tools in the software development environment; integrated development environments.
- Localization support.
- Cross-language support (ontology languages as opposed to natural or software languages, such as generation of ontologies in the RDF(S)/OWL family of description logics languages, or in the Knowledge Interchange Format (KIF) where first or higher order logics are required).
- Support for ontology analysis, including deductive closure; ontology comparison, merging, alignment, and transformation.
- Support for import/reverse engineering of RDBMS schemas, XML schemas and other semi-structured resources as a basis for ontology development.

7.7 Goals for Generic Ontologies and Tools

The diversity of the usage scenarios illustrates the wide applicability of ontologies within many domains. Table 7.3 brings these requirements together. To address all of these requirements would be an enormous task, beyond the capacity of the ODM development team. The team is therefore concentrating on the most widely applicable and most readily achievable goals. The resulting ODM will be not a final solution to the problem, but will be intended as a solid start which will be refined as experience accumulates.

Table 7.3 - Summary of Requirements

Requirement	Sub clause
Structural features	
Support ontologies expressed in existing description logic, (e.g., OWL/DL) and higher order logic languages (e.g., OWL Full and KIF), as well as emerging and new formalisms.	7.4.2 7.5.1 7.6.2
Represent complex objects as aggregations of parts	7.5.2
Multiple inheritance of complex types	7.5.2
Separation of concerns	7.4.1
Full or partial specification	7.4.2
Model-based architectures require extent-descriptive predicates to provide a description of a resource in an ontology, then generating a specific instantiation of that resource.	7.6.1
Efficient mechanisms will be needed to represent large numbers of similar classes or instances.	7.4.1
Generic content	
Support physical world concepts, including time, space, bulk or mass nouns like ‘water,’ and things that do not have identifiable instances.	7.4.2

Table 7.3 - Summary of Requirements

Support object concepts that have multiple facets of representations, e.g., conceptual versus representational classes.	7.4.1
Provide a basis for describing stateful representations, such as finite state automaton to support an autonomous agent’s world representation.	7.4.1
Provide a basis for information systems process descriptions to support interoperability, including such concepts as player, role, action, and object.	7.4.1
Other generic concepts supporting particular kinds of domains	7.4.2
Run-time tools	
Tools to assemble and disassemble complex sets of scientific and multi-media data.	7.5.2
Service to check message commitment to ontology	7.4.1
Design-time tools	
Full life-cycle support	7.4.3 7.6.2
Support for collaborative teams	7.4.3 7.6.2
Ease of use, transparency with respect to details	7.6.2
Support for modules and version control.	7.4.3
Consistency checking and validation, deductive closure	7.4.3 7.6.2
Mappings and similarity analysis	7.4.3 7.5.2 7.6.2
Interoperation with other tools, forward and reverse engineering	7.6.2
Localization support	7.6.2

The table classifies the requirements into

- structural features – knowledge representation requirements
- generic content – aspects of the world common to many applications
- run-time tools – use of the ontology during interoperation
- design-time tools – needed for the design of ontologies

Associated with each requirement are the usage scenario from which it mainly arises.

8 Design Principles

8.1 Design Principles

The ODM uses the design principles, such as modularity, layering, partitioning, extensibility and reuse, that are articulated in the UML specification [UML2].

8.2 Why Not Simply Use or Extend the UML 2.0 Metamodel?

An ontology is a conceptual model, and shares characteristics with more traditional data models. The UML Class Diagram is a rich representation system, widely used, and well-supported with software tools. Why not simply use UML for representing ontologies?

OWL concepts, particularly those of OWL DL, represent an implementation of a subset of traditional first order logic called Description Logics (DL), and are largely focused on sets and mappings between sets in order to support efficient, automated inference. UML class diagrams are also based in set semantics, but these semantics are not as complete; additionally, in UML, not as much care is taken to ensure the semantics are followed sufficiently for the purposes of automatic inference. This can potentially be rectified with OCL, which is part of UML 2.0. The issues can be categorized by cases where UML is overly restrictive, not restrictive enough, or simply doesn't provide the explicit construct necessary. For example:

- UML disjointedness requires disjoint classes to have a common super-type, which is not the case in OWL (aside from the fact that all OWL classes are ultimately subclasses of owl:Thing, and similarly that all classes in RDF Schema are resources).
- To model set intersection in UML one might consider using multiple inheritance, but this still allows an instance of all the super-classes to be omitted from the subclass.
- There is no UML construct for set complement.

The lack of reliable set semantics and model theory for UML prevents the use of automated reasoners on UML models. Such a capability is important to applying Model Drive Architecture to systems integration. A reasoner can automatically determine if two models are compatible, assuming they have a rigorous semantics and axioms are defined to relate concepts in the various systems.

Another distinction is in the ability to fully specify individuals apart from classes, and for individuals to have properties independently of any class they might be an instance of in OWL. In this regard, UML shows it software heritage, in which it is not possible for an instance to exist without a class to define its structure, a characteristic that derives from classes used as abstractions of memory layout. It is not hard to work around this using singleton classes as proposed in the profile, but for methodologies that start with instances and derive classes from them, this is clutter obviously introduced from a practice in which the reverse is the norm.

In OWL Full, it is also common to reify individuals as classes. OWL Full allows classes to have instances which are themselves classes or properties; classes and properties can be the domains of other properties. Elements of an ontology frequently cross meta-levels, and may represent the equivalent of multiple meta-levels depending on the domain, application, usage model, and so forth. Ontologists frequently want to see a combination of these classes and individuals on the same diagram, and find it unnatural if they cannot. Many software languages reify classes, but UML has been only half-hearted in supporting this mechanism. One can also work around this, however, as shown in the profile. The four-layer meta level architecture that UML resides in does not restrict class reification, even though it is often confused with reification. Classes and instances can reside on a single level of the architecture, at least if UML is used to describe that layer.

While some claim that UML would need to support properties independently of classes to be used in the OWL style, this is not actually the case. In fact, independent properties in OWL are semantically equivalent to properties on owl:Thing, which is directly translatable to UML using a model library, corresponding to the one proposed in the Foundation Ontology given in Annex A. OWL does not require the use of owl:Thing for properties without defined domains, but this is really just syntactic sugar. Note that the same is true when RDF vocabularies are developed without using any OWL constructs; for the purposes of this specification, the model library should be used in either case.

The above problems could potentially be addressed in a revision of UML. However, the RFP to which this specification is responding did not call for that.

8.3 Component Metamodel Selection

A trigger for the call for development of an ODM was the development by the World-Wide Web Consortium of a set of languages that form the foundation of the Semantic Web, including the Resource Description Framework (RDF), RDF Schema, and the Web Ontology Language (OWL). In addition, there have been many other ontology language development efforts, including International Standards Organization (ISO) projects for Topic Maps and Common Logic (CL). Topic Maps is a metalanguage designed to express the “aboutness” of an information structure with key model elements *topic* and *association*. Common Logic represents a family of knowledge representation languages. Common Logic, or CL, is a first order logic, analogous to predicate calculus, and is the successor to KIF (Knowledge Interchange Format). Both Topic Maps and CL have XML serializations, and were designed to express semantics for knowledge exchanged over the World Wide Web. These languages overlap with some parts of OWL as might be expected, but are used for different purposes and have different or no requirements for automated reasoning. CL is more expressive than OWL, and is better suited to applications involving declarative representation of rules and processes, for example.

As an initial part of the ODM development process, the team determined that understanding the requirements for ontology development using ODM metamodels was essential to establishing the ODM architecture and selecting an appropriate set of languages to be incorporated in the specification. The results of this requirements analysis are summarized in Clause 7. The set of languages represented, the architecture, and potential extensions currently envisioned developed as a direct consequence of this effort. This includes the notion that organizations developing ontologies may need to leverage pre-existing data and process models represented in UML, Entity-Relationship (ER), or another modeling language, even if the development effort itself is conducted using an ODM metamodel. For some possible extensions to better support certain classes of vocabularies or ontologies, see Annex D.

A significant exception is immanent ontologies, whose structure is derived from the information being exchanged as distinguished from transcendent ontologies, whose structure is provided *a priori* by schemas and the like. News feeds, results of data mining, and intelligence applications are examples of immanent ontologies, while e-commerce exchanges, engineering applications, and controlled vocabularies generally are transcendent. Immanent ontologies are represented by at least collections of terms, but often also by some numeric representation of the relationship among terms: co-occurrence matrices, conditional probabilities of co-occurrence, and eigenvectors of co-occurrence matrices, for example. These kinds of applications have not attracted the development of standardized representation structures as have transcendent ontologies. The ODM team considered that it was outside the scope of this specification to innovate in areas such as immanent ontology development without existing standard representations.

8.4 Relationships among Metamodels

8.4.1 The Need for Translation

The various metamodels in the ODM are treated equally, in that they are generally independent of each other. It is not necessary to understand or be aware of the others to understand any one in particular. The one exception to this is that the metamodel for OWL extends the metamodel for RDF, as the OWL language itself extends the RDF language.

However, in an ontology development project it might be necessary to use several of the metamodels, and to represent a given fragment of an ontology component in more than one. For example, consider a large e-commerce exchange project. The developers might choose to represent the ontology specifying the shared world governing the exchange in OWL. But the exchange might have evolved from a single large company's electronic procurement system (as was the case for example with the General Electric Global Exchange Service [GE]). The original procurement system might have been designed using UML, so that it would be a significant saving in development cost to be able to translate the UML specification to OWL as a starting point for development of the ontology.

Once such an exchange is operating, it may have thousands of members, each of which will have its own information system performing a variety of tasks in addition to interoperating through the exchange. These systems are all autonomous, and the exchange has no interest in how they generate and interpret the messages they use to interoperate so long as they commit to the ontology. Let us assume that the various members have systems with data models in UML or dialects of the ER model. A given member will need to subscribe to at least a fragment of the ontology and make sure its internal data model conforms to the fragment. It would therefore be an advantage to be able to translate a fragment of the ontology to UML or ER to facilitate the member making any changes to its internal operations necessary for it to commit to the ontology. Alternatively, a member might have a large investment in UML and would like the development to leverage UML experience and UML tools to make at least a first approximation to alignment with the OWL model.

It is extremely important for those leveraging existing artifacts for ontology development to understand that "what makes a good object-oriented software component model" does not necessarily make a good ontology. Once a particular model has been translated to OWL, for example, care needs to be taken to ensure that the resultant model will support the desired assertions in a knowledge base. Significant restructuring is often required, in other words.

The ODM therefore needs to provide facilities for making relationships among instances of its metamodels, including UML. There are two ways to accomplish this: UML profiles and mappings.

8.4.2 UML Profiles

The goal of a UML profile from the ODM perspective is to provide a bridge between the UML and knowledge representation communities on a well-grounded, semantic basis, with a broader goal of relating software and logical approaches to representing information. Profiles facilitate implementation using common notation on existing UML tools. They support renaming and specializing UML model elements in consistent ways, so that an instance of a UML model can be seen as an extended metamodel. Profiles allow a developer to leverage UML experience and tools while moving to integrating with an ontology represented in another metamodel.

We have provided such profiles for the Topic Maps, RDFS and OWL metamodels, as one of the primary goals that emerged from our use case development work was to enable use of existing UML tools for ontology modeling. The profiles provided in Clauses 14, and 15 were designed specifically for use in UML 2.0 tools. A profile for Common Logic is under consideration as an extension to this specification through the OMG's RFC process, as potential applications for its use in business semantics and production rules applications were identified late in the specification development process.

8.4.3 Mappings

Working with multiple metamodels may require a model element by model element translation of model instances from one metamodel to another. UML profiling provides some capability for users to leverage UML as a basis for ontology development for a specific knowledge representation language, such as RDF or OWL, but not necessarily to facilitate complete transformations across the set of representation paradigms included in the ODM metamodels. We therefore need to specify mappings from one metamodel to another.

Over the course of the ODM development, a parallel RFP effort called MOF QVT (Query/View/Transform) also reached finalization, providing a standardized MOF-based platform for mapping instances of MOF metamodels from one metamodel to another [MOF QVT]. Although the QVT specification is not yet finalized, it is sufficiently mature for use in defining informative mappings in the ODM.

Translation between metamodels has the fundamental problem that there may not be a single and separate model element in the target corresponding to each model element in the source (indeed, if the metamodels are not simply syntactic variations, this would be the normal situation). We will call this situation *structure loss*. Some of the issues involved with structure loss and what to do about it using one of the earlier QVT proposals are discussed in [MSDW].

An overview of the mapping strategy used in the ODM is illustrated in Clause 9. Note that there are mappings from each metamodel to and from OWL Full, except for Common Logic (CL) for which there is only a mapping from OWL Full. A lossy, reverse mapping defined in QVT from CL to OWL, and bi-directional mappings between UML and CL are planned, and may be added through an RFP/RFC process.

8.4.4 Mappings Are Informative, Not Normative

In Clause 9, The ODM is shown as having metamodels for several languages (RDFS/OWL, Topic Maps, and Common Logic) tied together by mappings to and from OWL (including UML to and from OWL). Common Logic is the exception, with mappings from OWL to CL only.

An argument for the infeasibility of normative mappings is presented in Annex E. In a nutshell, the mappings provided in the ODM are very general. Due to the very different scope and structure of the systems metamodeled, mappings based solely on the general structure of the languages will often lead to less than ideal choices for mapping some structures. Any particular mapping project will have additional constraints arising from the structure of the particular models to be mapped and the purposes of the project, so will very likely make different mapping choices than those provided in the ODM. An industry of consultants will likely arise, adding value by exactly this activity. They can use the ODM mappings as a takeoff point, and as an aid to understanding the comparative model structure, so the ODM mappings have value as informative, but not as normative.

8.5 Why Common Logic over OCL?

Common Logic (CL) is qualitatively different from some of the other metamodels in that it represents a dialect of traditional first order logic, rather than a modeling language. UML already supports a constraint (rule) language, which includes similar logic features, OCL [OCL], so why not use it?

The short answer to that question is that the ODM does include OCL in the same way it includes UML. Unfortunately, just as UML lacks a formal model theoretic semantics, OCL also has neither a formal model theory nor a formal proof theory, and thus cannot be used for automated reasoning (today). Common Logic, on the other hand, has both, and therefore can be used either as an expression language for ontology definition or as an ontology development language in its own right.

CL represents work that has been ongoing in the knowledge representation and logic community for a number of years. It is a second-generation language intended to have an extremely concise kernel for efficient reasoning, has a surface syntax for use with Semantic Web applications, and is rooted in the Knowledge Interchange Format (KIF Reference Manual v3.0 was published in 1992) as well as in other knowledge representation formalisms. It has also reached final committee draft status (24707) in JTC 1 / SC32 of the ISO/IEC standards community, and should be finalized by the end of 2006.

Our original work with regard to the metamodel was done with active participation of the CL language authors, and sought to be true to the abstract syntax of the CL language to the extent possible. Our intent was to enable ontologies developed using the ODM to be operated on by DL and CL reasoners downstream. There are a number of such reasoners available today, including Cerebra, FaCT, Pellet, Racer, and others from the DL community, as well as KIF-based reasoners such as Stanford's Java Theorem Prover (JTP) and OntologyWorks, CLIPS (similar to KIF) based reasoners such as Jess, and so forth, which ODM users can leverage for model consistency checking, model validation, and for applications development.

Finally, given that the ODM includes mappings among the metamodels for the modeling languages, why not include mappings between OCL and CL? Such a mapping should in principle be possible, but both languages are very rich. A mapping between them must deal with concerns about issues related to unintended semantics, the ability to write complex expressions involving multiple variables that preserve quantifier scope, and so forth. These issues are very important from a reasoning perspective, and thus our approach needs to be well developed and tested using both OCL and CL reasoners if we are to go down that path. This represents a longer term activity that may be taken up in the Ontology PSIG if there is sufficient commercial interest in doing so.

8.6 Why EMOF?

MOF 2 has two flavors, EMOF (Essential MOF) and CMOF (Complete MOF), with EMOF being equivalent to a subset of CMOF. We have used EMOF for the ODM for two reasons:

1. The advantage of using EMOF is that the modeling tools available during ODM development, such as IBM Rational Rose, support EMOF (or close to it) but not CMOF. It was therefore possible to use such tools to define ODM metamodels. At present, the newness of CMOF means that CMOF facilities are not supported by most tools. Therefore use of CMOF facilities imposes a significant burden.

2. The ODM metamodels can be represented in EMOF without sacrificing major syntactic or semantic considerations.

On the other hand, some of the possible extensions discussed in Annex D do require CMOF facilities. Use of EMOF in the development of the ODM does not preclude extensions to CMOF as might be advantageous, and as the tools evolve to support it.

8.7 M1 Issues

The ODM team encountered some issues in developing MOF-based metamodels for the W3C languages RDF, RDFS and OWL, and to a lesser extent the ISO language Topic Maps. A MOF-based metamodel has a strict separation of meta-levels. The number and designation of meta-levels is changed in MOF2 from MOF 1.4, but the issue can be described in the MOF 1.4 designations:

- M3 – the MOF
- M2 – a MOF class model, specifying the classes and associations of the system being modeled, the structure of OWL for example.

- M1 – an instance of an M2 model, describing a particular instance of the system being modeled, a particular OWL ontology, for example.
- M0 – ground individuals. A population of instances of the classes in a particular OWL ontology, for example.

RDFS and OWL are defined as specializations of RDF. RDF has natively a very simple model. There are resources and properties. The entire structure of RDF, RDFS and OWL is defined in terms of instances of resources, properties, and other structures like classes, which are defined in terms of built-in resources and properties. In fact, even property is formally defined as an instance of resource, and resource (the set of resources) is itself an instance of resource. These languages are self-referential in a way that a native MOF metamodel could never be.

The same is true to a lesser degree of Topic Maps. Although the ISO standard provides a Topic Map Data Model, some important constructs like class and subclass are defined as published subjects, which are instances of topics. Topics are defined at the M2 level, so published subjects are M1 objects.

The Topic Maps metamodel in the ODM deals with the M1 problem by having an M2 structure following the published Topic Map Data Model, with a note detailing the built-in M1 published subjects, but this approach does not suit the W3C languages. In the ODM we have modeled RDF, RDF Schema, and OWL at the M2 level, following the published abstract syntax for them. Certain built-in RDF/S and OWL constructs have relevance at multiple MOF meta-levels. Some of these, such as annotation properties including rdfs:seeAlso, are included as M2 elements in the RDFS Metamodel; others, such as ontology properties including owl:priorVersion, are included as M2 elements in the OWL Metamodel.

Some important constructs, however, are not appropriate to model at all at the M2 level. These are provided in an ontology as an M1 model library (given in Annex A), and include:

- Two built-in classes - `owl:Thing` and `owl:Nothing`
- The built-in empty list - `rdf:Nil`
- The built-in rdf:value property, suggested for use with structured values but not recommended for use by ODM team members
- Instances of `rdfs:ContainerMembershipProperty` (e.g., `rdf:_1`, `rdf:_2`, etc.)
- The set of XML Schema datatypes that are supported in RDF/S and OWL - `xsd:string, xsd:boolean, xsd:decimal, xsd:float, xsd:double, xsd:dateTime, xsd:time, xsd:date, xsd:gYearMonth, xsd:gYear, xsd:gMonthDay, xsd:gDay, xsd:gMonth, xsd:hexBinary, xsd:base64Binary, xsd:anyURI, xsd:normalizedString, xsd:token, xsd:language, xsd:NMTOKEN, xsd:Name, xsd:NCName, xsd:integer, xsd:nonPositiveInteger, xsd:negativeInteger, xsd:long, xsd:int, xsd:short, xsd:byte, xsd:nonNegativeInteger, xsd:unsignedLong, xsd:unsignedInt, xsd:unsignedShort, xsd:unsignedByte` and `xsd:positiveInteger`
- Additional RDF/S and OWL constructs that may have counterparts in the M2 metamodels (e.g., annotation properties such as `rdfs:label` and `rdfs:comment`).

9 ODM Overview

As introduced briefly in the RFP [ODM RFP], ontology is a discipline rooted in philosophy and formal logic, introduced by the Artificial Intelligence community in the 1980s to describe real world concepts that are independent of specific applications. Over the past two decades, knowledge representation methodologies and technologies have subsequently been used in other branches of computing where there is a need to represent and share contextual knowledge independently of applications.

The following definition was adopted from the RFP:

> An ontology defines the common terms and concepts (meaning) used to describe and represent an area of knowledge. An ontology can range in expressivity from a Taxonomy (knowledge with minimal hierarchy or a parent/child structure), to a Thesaurus (words and synonyms), to a Conceptual Model (with more complex knowledge), to a Logical Theory (with very rich, complex, consistent, and meaningful knowledge).

This definition, and the analysis presented in Clause 7, led to the determination that the ODM would ultimately include six metamodels (four that are normative, and two that are informative). These are grouped logically together according to the nature of the representation formalism that each represents: formal first order and description logics, structural and subsumption / descriptive representations, and traditional conceptual or object-oriented software modeling.

At the core are two metamodels that represent formal logic languages: DL (Description Logics, which, although it is non-normative, is included as informative for those unfamiliar with description logics, [BCMNP]) and CL (Common Logic), a declarative first-order predicate language. While the heritage of these languages is distinct, together they cover a broad range of representations that lie on a continuum ranging from higher order, modal, probabilistic, and intentional representations to very simple taxonomic expression.

There are three metamodels that represent more structural or descriptive representations that are somewhat less expressive in nature than CL and some DLs. These include metamodels of the abstract syntax for RDFS [RDF Schema], OWL [OWL Reference]; [OWL S&AS], and TM (Topic Maps, [TMDM]). RDFS, OWL and TM are commonly used in the semantic web community for describing vocabularies, ontologies and topics, respectively.

Two additional metamodels considered essential to the ODM represent more traditional, software engineering approaches to conceptual modeling: UML2 [UML2] and ER (Entity Relationship) diagramming. UML and ER methodologies are arguably the two most widely used modeling languages in software engineering today, particularly for conceptual or logical modeling. Interoperability with and use of intellectual capital developed in these languages as a basis for ontology development and further refinement is a key goal of the ODM. Since UML2 is an adopted OMG standard, we simply reference it in the ODM, and provide an additional non-normative mechanism for handling keys from an ER perspective in Annex B. We anticipate a full metamodel for ER diagramming will be provided in the upcoming Information Modeling and Management specification, the follow-on to the current Common Warehouse Metamodel (CWM), and that there will be a mapping developed from the ODM to this new ER metamodel when it becomes available.

Three UML profiles have been identified for use with the ODM for RDF, OWL, and Topic Maps. These enable the use of UML notation (and tools) for ontology modeling and facilitate generation of corresponding ontology descriptions in RDF, OWL, and TM, respectively.

In addition, in order to support the use of legacy models as a starting point for ontology development, and to enable ODM users to make design trade-offs in expressivity based on application requirements, mappings among a number of the metamodels are provided. As discussed in 8.4.3, these mappings are expressed in the MOF QVT Relations Language. To avoid an n-squared set of mappings, the ODM includes direct mappings to and from OWL for UML and Topic Maps.

CL is an exception to this strategy. CL is much more expressive than the other metamodels, and is therefore much more difficult to map into the other metamodels. CL can be used to define constraints and predicates that cannot be expressed (or are difficult to express) in the other metamodels. Some predicates might be specified in a primary metamodel, for example, in OWL, and refined or further constrained in CL. The relevant elements of the M1 model expressed in the primary metamodel will be mapped into CL. Thus, uni-directional mappings (to CL), only, are included or planned at this time.

Figure 9.1 shows the organization of the metamodels the relationships between the RDF and OWL packages.

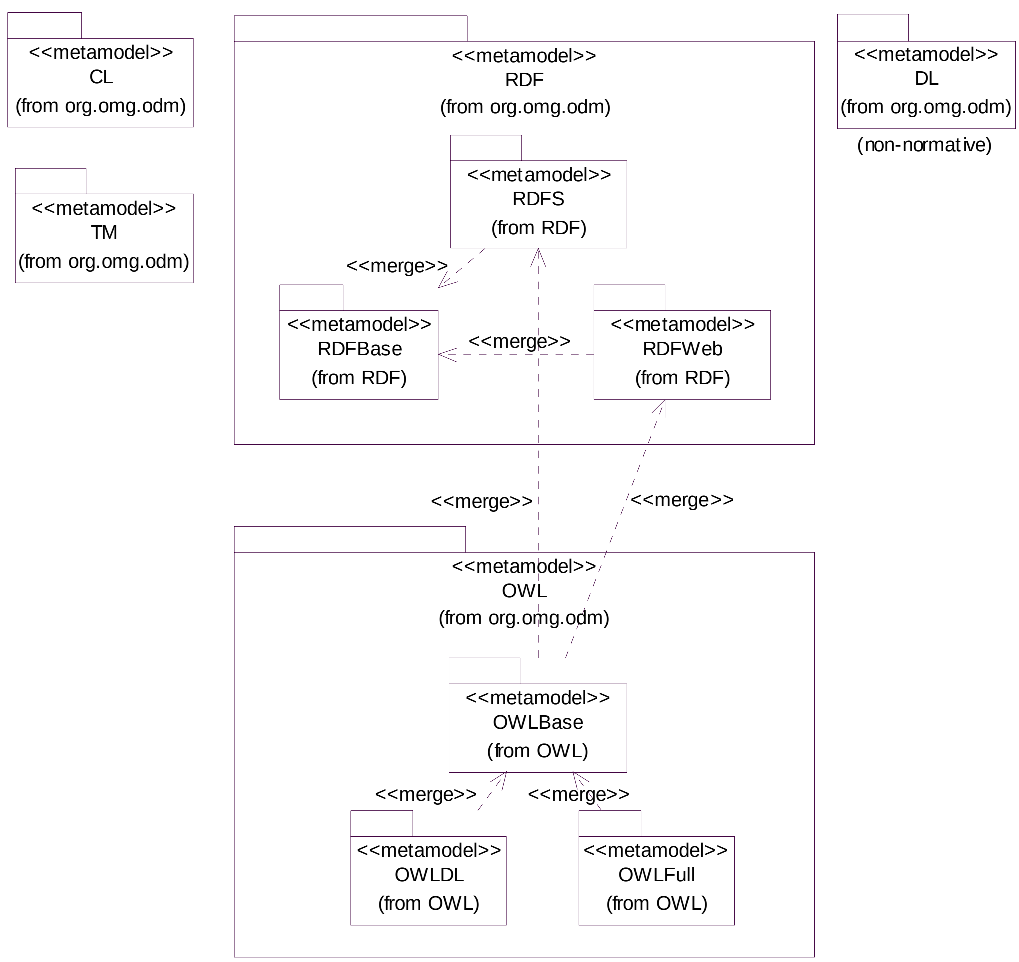

Figure 9.1 - ODM Metamodels: Package Structure

10 The RDF Metamodel

10.1 Overview

The Resource Description Framework (RDF) is a language standardized by the World Wide Web Consortium for representing information (metaknowledge) about resources in the World Wide Web. It builds on a number of existing W3C standards (including XML, URI, and related specifications), and provides a standardized way of defining vocabularies for the Web that have an underlying formal model theoretic semantics, sufficient to support automated reasoning if so desired.

10.1.1 Organization

The set of specifications that define RDF are divided into several components: basic concepts and abstract syntax, RDF Schema that provides additional vocabulary development capabilities building on RDF, and a number of concrete syntax variations, notably N3 and RDF/XML.

The RDF Concepts and RDF Schema (RDFS), metamodels, defined herein, are MOF2 compliant metamodels that allow a user to define RDF vocabularies using the terminology and concepts defined in the RDF specifications.

At the core, the RDF Concepts package reflects core concepts required by virtually all RDF applications, roughly, the set of concepts defined in the RDF Concepts and Abstract Syntax specification [RDF Concepts].

The RDFS metamodel includes the concepts defined in the RDF Concepts package, and extends them to support the vocabulary language defined in the RDF Schema specification [RDF Schema]. Figure 10.1 provides an overview of the package structure.

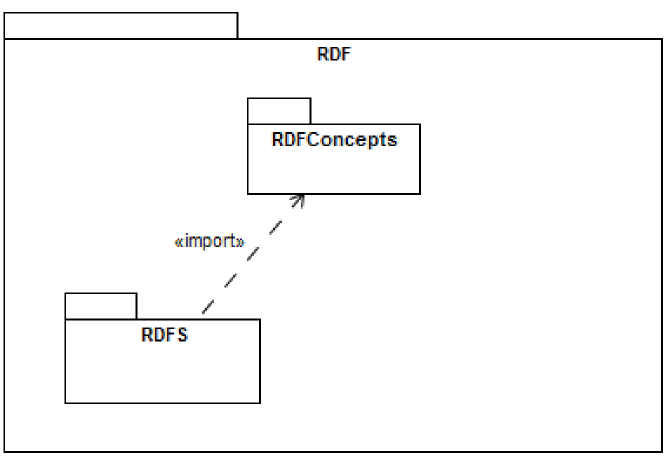

Figure 10.1 - Structure of the RDF Metamodel

10.1.2 Design Considerations

10.1.2.1 Metamodel Constructs

RDF classes are represented by MOF classes. RDF properties are represented either by MOF classes or associations, as appropriate.

RDF properties are first-class entities with unique identifiers. In addition, an RDF property can be a subproperty of another RDF property. MOF associations, on the other hand, are not first-class entities. They are defined between two MOF classes and their role names are locally scoped. In addition, in EMOF, a MOF association cannot be a sub association of another MOF association, which exemplifies an inherent impedance mismatch between RDF Schema and EMOF. Naming conventions, constraints in OCL, and textual description are used to overcome this impedance mismatch.

10.1.2.2 Naming

Classes and properties defined in the RDF metamodel(s) have prefixed names derived from the way RDF and RDFS namespaces are partitioned. One notable exception is RDFGraph, which does not have an explicit equivalent in the RDF specifications but is differentiated for the sake of clarity with respect to the OWL metamodel.

Several RDF language co-authors shared that the distinction between the RDF and RDFS namespaces has become less important and blurred over time, which has caused confusion in the Semantic Web community and created challenges for metamodeling. Distinguishing between the RDF Concepts and RDFS packages along namespace boundaries was impossible from a UML perspective for a number of reasons. `rdfs:Container` generalizes `rdf:Alt`, `rdf:Bag`, and `rdf:Seq`, for example, and there are other such cases. In order to retain the original namespace naming conventions in a set of packages that are not segregated by namespace, we have prefixed those concepts that relate directly to their RDF/S counterparts accordingly.

In addition, names of MOF classes are package qualified rather than globally scoped, as is the case with conventional XML uniform resource identifiers (URIs). In fact, `rdfs:Class`, `rdf:Property`, and other names specified in the RDF specifications are actually abbreviations for URIs using conventional namespace prefixes and concatenation. To make matters worse, names of MOF association roles are local to the MOF classes where they are defined. The prefix convention we have adopted assists in overcoming this impedance mismatch. For example, RDFSClass represents `rdfs:Class` and RDFProperty represents `rdf:Property`. Concepts that are not explicitly part of the concrete vocabulary of RDF/S are not prefixed in this manner.

10.2 RDF Triples

Figure 10.2 depicts the RDF base graph data model. For those applications that require use of the RDF graph model, as specified in [RDF Concepts], support for explicit manipulation of blank nodes may be needed. The definitions provided herein facilitate resolution of blank node semantics and finer granularity in manipulation of nodes in an RDF graph.

RDF provides a reification vocabulary for making statements about triples. This is described in 10.4.

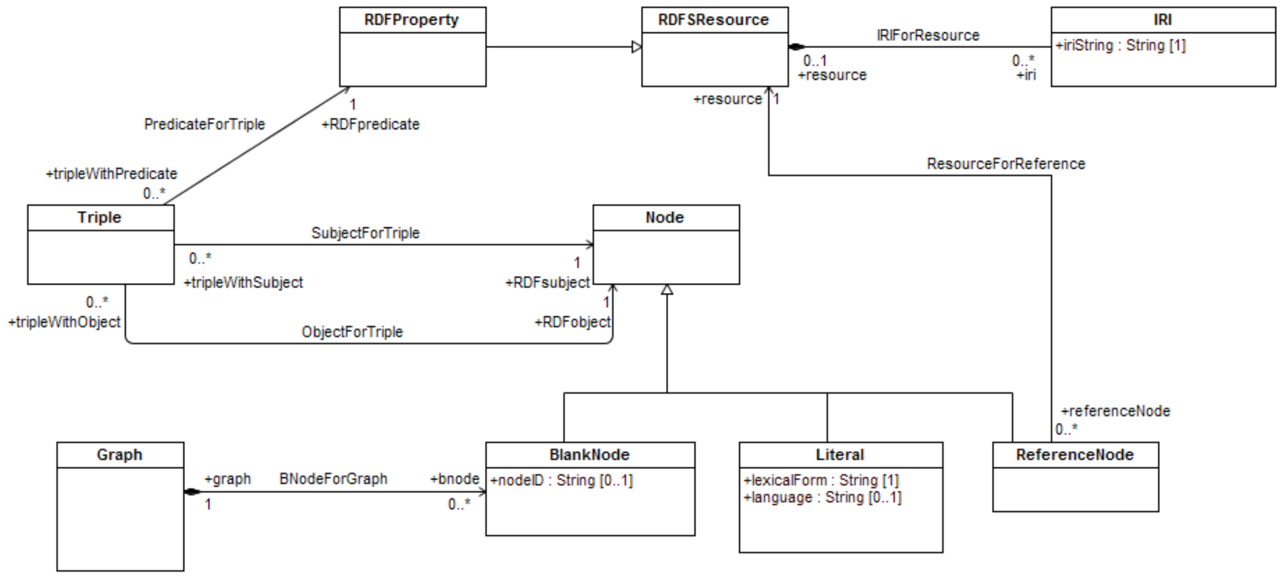

Figure 10.2 - RDF Concepts Package, The Graph Data Model

10.2.1 BlankNode

Description

A blank node is a node that is not a reference to a resource or a literal. In the RDF abstract syntax, a blank node is simply a unique node that can be used in one or more RDF statements, but has no intrinsic name.

A convention used to refer to blank nodes by some linear representations of an RDF graph is to use a blank node identifier, which is a local identifier that can be distinguished from IRIs and literals. When graphs are merged, their blank nodes must be kept distinct if meaning is to be preserved. Blank node identifiers are not part of the RDF abstract syntax, and the representation of triples containing blank nodes is dependent on the particular concrete syntax used, thus no constraints are provided here on blank node identifiers. They are optional, included strictly as a placeholder for tool vendors whose applications require them, and in particular, for interoperability among such tools.

Attributes

- nodeID: String [0..1] - the optional blank node identifier.

Associations

- Specialize Class Node.

Constraints

None.

Semantics

RDF makes no reference to the internal structure of blank nodes. The methodology for making such a determination is left to the applications that use them, for example, through reasoning about them.

Blank nodes are treated as simply indicating the existence of a thing, without using or saying anything about the name of that thing. (This is not the same as assuming that the blank node indicates an 'unknown' IRI; for example, it does not assume that there is any IRI reference that refers to the thing.) Thus, they are essentially treated as existentially quantified variables in the graph in which they occur, and have the scope of the entire graph. More on the semantics of blank nodes is given in [RDF Semantics].

10.2.2 Node

Description

The subject and object of a Triple are of type Node. ReferenceNode, BlankNode, and Literal form a complete and disjoint covering of Node.

Attributes

None.

Associations

- tripleWithSubject: Triple [0..*] in association SubjectForTriple - the triple for which this node is a subject.
- tripleWithObject: Triple [0..*] in association ObjectForTriple - the triple for which this node is an object.

Constraints

[1] The set of blank nodes, the set of all RDF IRI references (i.e., ReferenceNodes) and the set of all literals are pairwise disjoint.

[2] ReferenceNode, BlankNode, and Literal form a complete covering of Node.

```
context Node inv DisjointPartition:
    (self.resource->notEmpty implies self.oclIsTypeOf(ReferenceNode)) and
    (self.oclIsTypeOf(ReferenceNode) or self.oclIsTypeOf(BlankNode) or
     self.oclIsTypeOf(Literal)) and
     not (self.oclIsTypeOf(ReferenceNode) and self.oclIsTypeOf(BlankNode)) and
     not (self.oclIsTypeOf(BlankNode) and self.isTypeOf(Literal)) and
     not (self.oclIsTypeOf(ReferenceNode) and self.oclIsTypeOf(Literal))
```

Semantics

This type represents the nodes in RDF Graphs.

10.2.3 RDFProperty

Description

The RDF Concepts and Abstract Syntax specification [RDF Concepts] describes the concept of an RDF property as a relation between subject resources and object resources. Every property is associated with a set of instances, called the property extension. Instances of properties are pairs of RDF resources.

Attributes

None.

Associations

- tripleWithPredicate: Triple [0..*] in association PredicateForTriple -- links a triple to its predicate
- iri: IRI [0..1] in derived association IRIForResource - the IRI(s) associated with a resource
- Specialize Class RDFSResource

Constraints

[1] The predicate of an RDF triple is a URI Reference (thus, a resource that is an RDF property, when used as the predicate of a triple, must have a URI reference).

```
context RDFProperty HasURI inv:
    self.uriRef->notEmpty
```

Semantics

A property relates resources to resources or literals. A property can be declared with or without specifying its domain (i.e., classes which the property can apply to) or range (i.e., classes or datatypes that the property may have value(s) from). This type represents the arc in RDF graphs.

10.2.4 RDFSResource

Description

All things described by RDF are called resources. This is the class of everything. All other classes are subclasses of this class.

Attributes

None.

Associations

- iri: IRI [0..*] in association IRIForResource - the IRI(s) associated with a resource.

Constraints

None.

Semantics

The iri property is used to uniquely identify an RDF resource globally. Note that this property has a multiplicity of [0..*] that provides for the possibility of the absence of an identifier, as in the case of blank nodes and literals. A particular resource may be identified by more than one IRI.

10.2.5 RDF Triple

Description

An RDF triple contains three components:

- The subject, which is a resource reference or a blank node.
- The predicate, which is a resource reference and represents a relationship.
- The object, which is a resource reference, a literal, or a blank node.

An RDF triple is conventionally written in the order subject, predicate, object. The relationship represented by the predicate is also known as the property of the triple. The direction of the arc is significant: it always points toward the object.

Attributes

None.

Associations

- RDFsubject: Node [1] in association SubjectForTriple - links a triple to the node that is the subject of the triple.
- RDFpredicate: RDFProperty [1] in association PredicateForTriple - links a triple to the property that is the predicate of the triple.
- RDFobject: Node [1] in association ObjectForTriple - links a triple to the node that is the object of the triple.

Constraints

[1] The resource (node) representing an RDFsubject can be an IRI or a blank node but not a literal.

```
context Triple SubjectNotALiteral inv:
   not self.RDFsubject.oclIsKindOf(Literal)
```

[2] An RDFpredicate must be an IRI (i.e., must not be a literal or blank node).

```
context Triple PredicateNotALiteral inv:
   not self.RDFpredicate.oclIsKindOf(Literal)
context RDFStatement PredicateNotABlankNode inv:
   not self.RDFpredicate.oclIsKindOf(BlankNode)
```

Note: Both of these constraints are subject to change (may be relaxed) based on user experience in the Semantic Web community. However, in any case, the constraint that a predicate must not be a literal is likely to remain.

Semantics

Each triple represents a statement of a relationship between the things denoted by the nodes that it links. The assertion of an RDF triple says that some relationship, indicated by the predicate, holds between the things denoted by subject and object of the triple.

10.2.6 IRI

Description

An IRI (Internationalized Resource Identifier) is a Unicode string that conforms to the syntax defined in RFC 3987. IRIs are a generalization of URIs (Uniform Resource Identifiers, as defined in RFC3986), that can include a broader range of Unicode characters.

IRIs in the RDF abstract syntax must be absolute, and may contain a fragment identifier. Some concrete RDF syntaxes allow the use of relative IRIs as a shorthand notation to make it easier to develop documents independently from their final publishing location. Relative IRIs must be resolved against a base IRI to make them absolute. Therefore, the RDF graph serialized in such syntaxes is well-defined only if a base IRI can be established per RFC3986.

Attributes

- iriString: String [1] - the string representing the IRI.

Associations

None.

Constraints

IRIs must conform to the character encoding (including escape sequences and so forth) defined in [RDF Syntax] and are globally defined. This is in contrast to naming and namespace conventions in UML2, which can be limited to the package level or to a set of nested namespaces. While it may not be possible to define constraints on character strings in OCL to enforce this, tools that implement this metamodel will be expected to support the W3C standards and related RFCs in this regard.

Semantics

None.

10.2.7 ReferenceNode

Description

References to resources as subjects or objects of triples are captured as a distinct type of node: a ReferenceNode.

Attributes

None.

Associations

- resource: RDFSResource [1] in association ResourceForReference - the resource referenced by this node.
- Specialize Class Node

Constraints

None.

Semantics

No additional semantics.

10.3 RDF Literals

Figure 10.3 provides the remaining definitions included in the base package for RDF, namely, the definitions specific to RDF literals.

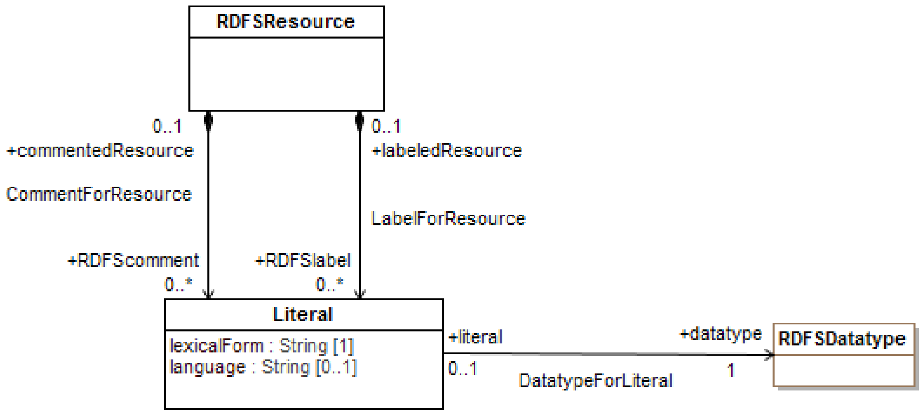

Figure 10.3 - RDF Concepts Package, The Literals Diagram

10.3.1 RDFSResource (Augmented Definition)

Associations

- RDFScomment: PlainLiteral [0..*] in association CommentForResource - links a resource to a comment, or human-readable description, about that resource.
- RDFSlabel: PlainLiteral [0..*] in association LabelForResource - links a resource to a human-readable name for that resource.

10.3.2 Literal

Description

Literals are used to identify values such as numbers and dates by means of a lexical representation. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals. Literals have a lexical form, which is a Unicode string, and a datatype URI being an RDF URI reference. If they represents strings they may also have a language.

Attributes

- lexicalForm: String [1] - represents a Unicode string in Normal Form C.
- language: String [0..1] - the optional language tag.

Associations

- datatype: RDFSDatatype[1] - the link between the typed literal and the RDFSDatatype that defines its type (of which it is an instance).
- Specialize Class Node

Constraints

[1] The language property may only have a value if the datatype represents http://www.w3.org/1999/02/22-rdf-syntax-ns#langString.

[2] The value of the language property MUST be well-formed according to section 2.2.9 of [BCP47].

Semantics

The datatype association refers to a datatype. For XML Schema built-in datatypes, this may be a proxy corresponding to URIs such as http://www.w3.org/2001/XMLSchema#int. The URI of the datatype rdf:XMLLiteral may be used. There may be other, implementation dependent, mechanisms by which proxies refer to datatypes.

The value associated with a typed literal is found by applying the lexical-to-value mapping associated with the datatype URI to the lexical form. If the lexical form is not in the lexical space of the datatype associated with the datatype URI, then no literal value can be associated with the typed literal. Such a case, while in error, is not syntactically ill formed.

10.4 RDF Statements

RDF provides a reification vocabulary with no formal semantics [RDF Schema].

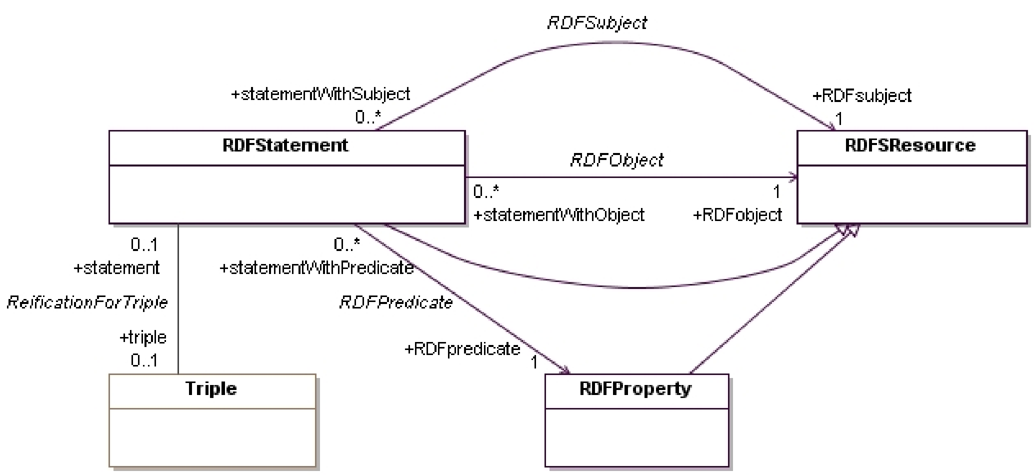

Figure 10.4 - RDF Concepts Package, the Reification Diagram

10.4.1 RDFProperty (Augmented Definition)

Associations

- statementWithPredicate: RDFStatement [0..*] in association RDFPredicate - links a statement to its predicate.

10.4.2 RDFSResource (Augmented Definition)

Associations

- statementWithObject: RDFStatement [0..*] in association RDFObject - a resource represents an object of zero or more RDF statements.

- StatementWithSubject: RDFStatement [0..*] in association RDFSubject - a resource represents a subject of zero or more RDF statements.

10.4.3 RDFStatement

Description

RDF Statement provides a way to make statements about triples or describe statements without asserting them.

Attributes

None.

Associations

- RDFobject :RDFSResource [1] in association RDFObject - links a statement to the resource that is its object.
- RDFpredicate: RDFSProperty [1] in association RDFPredicate - links a statement to a property that is its predicate.
- RDFsubject: RDFSResource [1] in association RDFSubject - links a statement to a resource that is its subject.
- triple: Triple [0..1] in association ReificationForTriple - links a statement to the triple it reifies, if such a triple exists.
- Specialize Class RDFSResource.

Constraints

None.

Semantics

None.

10.4.4 Triple (Augmented Definition)

Associations

- statement: RDFStatement [0..1] in association ReificationForTriple - the statement that reifies the triple, if such a statement exists.

10.5 Classes and Utilities

As shown in Figure 10.5, resources may be divided into groups called classes. The members of a class are known as instances of the class. Classes are themselves resources. They are often identified by IRIs and may be described using RDF properties. The rdf:type property may be used to state that a resource is an instance of a class.

RDFS distinguishes between a class and the set of its instances. Associated with each class is a set, called the class extension of the class, which is the set of the instances of the class. Two classes may have the same set of instances but be different classes. A class may be a member of its own class extension and may be an instance of itself. This feature of RDF Schema (and, as a result, of OWL) may be unintuitive for a traditional UML user, and makes distinguishing metalevels in an ontology challenging.

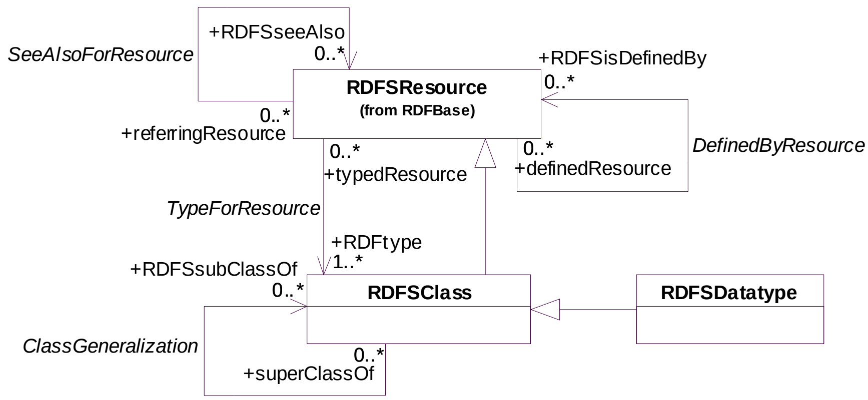

Figure 10.5 - RDFS Package, The Classes & Utilities Diagram

10.5.1 RDFSClass

Description

The group of resources that are RDF Schema classes is itself a class, called `rdfs:Class`. Classes provide an abstraction mechanism for grouping resources with similar characteristics.

If a class C is a subclass of a class C', then all instances of C will also be instances of C'. The `rdfs:subClassOf` property may be used to state that one class is a subclass of another. The term *superClassOf* is used as the inverse of `rdfs:subClassOf`. If a class C' is a superclass of a class C, then all instances of C are also instances of C'.

Attributes

None.

Associations

- RDFSsubClassOf: RDFSClass [0..*] in association ClassGeneralization - links a class to another class that generalizes it.
- superClassOf: RDFSClass [0..*] in association ClassGeneralization - links a class to another class that specializes it (note that superClassOf is not an RDF concept).
- typedResource: RDFSResource [0..*] in association TypeForResource - links a class to a resource that is an instance of the class.
- Specialize Class RDFSResource.

Constraints

None.

Semantics

A resource can be a member of multiple classes in RDF Schema.

10.5.2 RDFSDatatype

Description

Datatypes are used by RDF in the representation of values such as integers, floating point numbers, and dates. A datatype consists of a lexical space, a value space, and a lexical-to-value mapping.

RDF predefines just one datatype `rdf:XMLLiteral`, used for embedding XML in RDF. There are no built-in concepts for numbers, dates, or other common values. Rather, RDF defers to datatypes that are defined separately and identified with URI references. The predefined XML Schema Datatypes [XML Schema Datatypes] are expected to be used for this purpose. Additionally, RDF provides no mechanism for defining new datatypes. XML Schema provides a framework suitable for defining new datatypes for use in RDF.

`rdfs:Datatype` is the class of datatypes. All instances of `rdfs:Datatype` correspond to the RDF model of a datatype described in the RDF Concepts specification [RDF Concepts] `rdfs:Datatype` is both an instance of and a subclass of `rdfs:Class`. Each instance of `rdfs:Datatype` is a subclass of `rdfs:Literal`.

Attributes

None.

Associations

- resource: RDFSResource [1] in association ResourceForReference - the resource referenced by this node.
- Specialize Class RDFSClass.

Constraints

[1] RDFSDatatype classes must inherit IRIs from RDFSResource.

```
context RDFSDatatype HasIRI inv:
    self.iri->notEmpty
```

[2] Each instance of RDFSDatatype is a subclass of TypedLiteral:

```
context RDFSDatatype InstancesAreLiterals inv:
    self.instance->forall (instance | instance.oclIsKindOf(TypedLiteral ))
```

Semantics

RDF provides for the use of externally defined datatypes identified by a particular URI reference, but imposes minimal conditions on datatype definitions. It includes a single built-in datatype, `rdf:XMLLiteral`.

The semantics given for datatype definitions are minimal. RDF makes no provision for associating a datatype with a property so that it applies to all values of the property, and does not provide any way of explicitly asserting that a blank node denotes a particular datatype value. Such features may be provided in the future, for example, more elaborate datatyping conditions. Semantic extensions may also refer to other kinds of information about a datatype, such as orderings of the value space.

A datatype is an entity characterized by a set of character strings called lexical forms and a mapping from that set to a set of values. How these sets and mappings are defined is considered external to RDF.

Formally, a datatype d is defined by three items:

- a non-empty set of character strings called the lexical space of d;
- a non-empty set called the value space of d;

- a mapping from the lexical space of d to the value space of d, called the lexical-to-value mapping of d.

The set of datatypes from [XML Schema Datatypes] available for use in RDF is limited to those with well defined semantics, those that do not depend on enclosing XML documents (e.g., `xsd:QName` is excluded), those that are not used for XML document cross-reference purposes, and so forth. The set of allowable datatypes is provided in Annex A.

10.5.3 RDFSResource (Augmented Definition)

Description

Note that the multiplicity on RDFtype is [1..*], meaning that every resource must be typed. Yet, many resources in RDF are not explicitly typed, so this may seem unintuitive from an RDF perspective. In essence, this says that every resource is, at a minimum, of type `rdfs:Resource` (required from a metamodeling and mapping perspective to support representation of RDF and OWL individuals without the addition of other artificial constructs). This does not, however, necessarily mean that vendors should add the inferred triples automatically when generating RDF/S and/or OWL from a model instance. This should only be done deliberately, depending on the application.

Associations

- definedResource: RDFSResource [0..*] in association DefinedByResource - relates a particular resource to other resources that it defines.
- RDFSisDefinedBy: RDFSResource [0..*] in association DefinedByResource - relates a resource to another resource that defines it; `rdfs:isDefinedBy` is a subPropertyOf `rdfs:seeAlso`.
- RDFSseeAlso: RDFSResource [0..*] in association SeeAlsoForResource - relates a resource to another resource that may provide additional information about it.
- referringResource: RDFSResource [0..*] in association SeeAlsoForResource - relates a particular resource to other resources that it may assist in defining.
- RDFtype: RDFSClass [1..*] in association TypeForResource - relates a resource to its type (i.e., states that the resource is an instance of the class that is its type).

Constraints

[1] RDFSseeAlso and RDFSisDefinedBy must have non-empty IRIs.

[2] RDFSisDefinedBy is a subPropertyOf RDFSseeAlso.

10.6 RDF Properties

The RDF Concepts and Abstract Syntax specification [RDF Concepts] describes the concept of an RDF property as a relation between pairs of resources.

RDF Schema defines the concept of subproperty. The rdfs:subPropertyOf property may be used to state that one property is a subproperty of another. If a property P is a subproperty of property P', then all pairs of resources that are related by P are also related by P'. The term super-property is often used as the inverse of subproperty. If a property P' is a super-property of a property P, then all pairs of resources that are related by P are also related by P'.

RDF/RDFS does not define a top property that is the super-property of all properties. Such a definition may be included in a model library if vendors so desire. The properties diagram is shown in Figure 10.6.

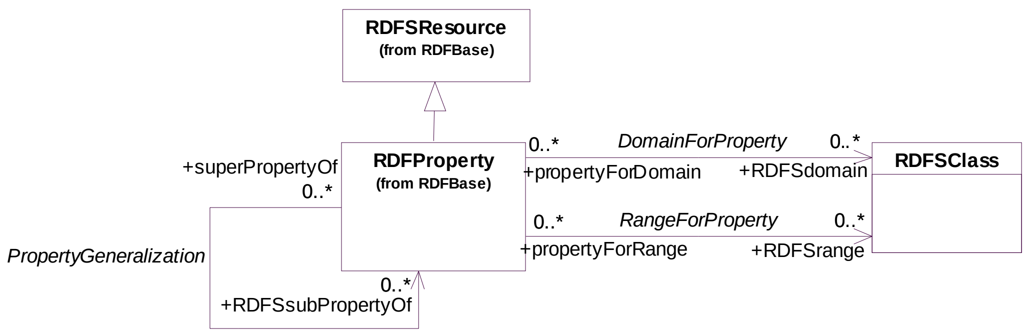

Figure 10.6 - RDFS Package, The Properties Diagram

10.6.1 RDFProperty (Augmented Definition)

Associations

- RDFSdomain: RDFSClass [0..*] in association DomainForProperty - links a property to zero or more classes representing the domain of that property. A triple of the form: `P rdfs:domain C`. states that P is an instance of the class `rdf:Property`, that C is an instance of the class `rdfs:Class` and that the resources denoted by the subjects of triples whose predicate is P are instances of the class C. Where a property P has more than one `rdfs:domain` property, then the resources denoted by subjects of triples with predicate P are instances of all the classes stated by the `rdfs:domain` properties.
- RDFSrange: RDFSClass [0..*] in association RangeForProperty - links a property to zero or more classes representing the range of that property. A triple of the form: `P rdfs:range C .` states that P is an instance of the class `rdf:Property`, that C is an instance of the class `rdfs:Class` and that the resources denoted by the objects of triples whose predicate is P are instances of the class C. Where P has more than one `rdfs:range` property, then the resources denoted by the objects of triples with predicate P are instances of all the classes stated by the `rdfs:range` properties.
- RDFSsubPropertyOf: RDFProperty [0..*] in association PropertyGeneralization - links a property to another property that generalizes it. The property `rdfs:subPropertyOf` is used to state that all resources related by one property are also related by another. A triple of the form: `P1 rdfs:subPropertyOf P2`. states that P1 is an instance of `rdf:Property`, P2 is an instance of `rdf:Property`, and P1 is a subproperty of P2. The `rdfs:subPropertyOf` property is transitive.
- superPropertyOf: RDFProperty [0..*] in association PropertyGeneralization - links a property to another property that specializes it (note that superPropertyOf is not an RDFS concept).

Semantics

Properties may be specialized. The existence of an instance of a specializing property implies the existence of an instance of the specialized property, relating the same set of resources.

10.6.2 RDFSClass (Augmented Definition)

Associations

- propertyForDomain: RDFProperty [0..*] in association DomainForProperty - links a class to a property for which it is the domain.

- propertyForRange: RDFProperty [0..*] in association RangeForProperty - links a class to a property for which it is the range.

10.7 Containers and Collections

RDF containers are resources that are used to represent groupings. The same resource may appear in a container more than once. Unlike containment in the physical world, a container may be contained in itself.

Figure 10.7 provides the metamodel elements defined to support RDF containers and collections.

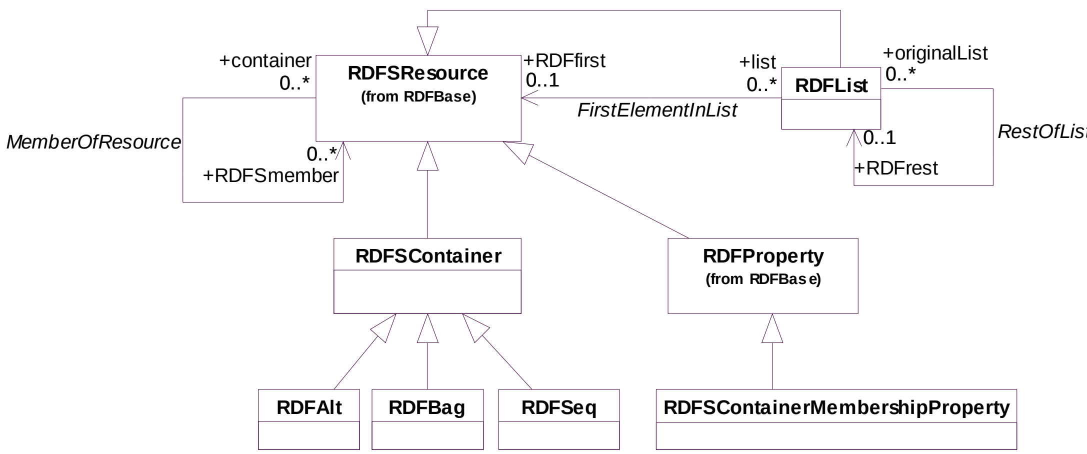

Figure 10.7 - RDFS Package, The Containers and Collections Diagram

Three different kinds of container are defined for different intended uses. An `rdf:Bag` is used to indicate that the container is intended to be unordered. An `rdf:Seq` is used to indicate that the order indicated by the numerical order of the container membership properties of the container is intended to be significant. An `rdf:Alt` container is used to indicate that typical processing of the container will be to select one of the members.

10.7.1 RDFAlt

Description

This is the class of RDF “Alternative” containers. The `rdf:Alt` class is used conventionally to indicate to a human reader that typical processing will be to select one of the members of the container. The first member of the container, i.e., the value of the `rdf:_1` property, is the default choice.

Attributes

None.

Associations

- Specialize Class RDFSContainer

Constraints

None.

Semantics

See discussion in [RDF Concepts] of container membership semantics. (Note that the blank nodes are intended to be interpreted as existentially quantified variables representing instances of URIs in `rdf:Property`.)

10.7.2 RDFBag

Description

This is the class of RDF “Bag” containers. It is used conventionally to indicate that the container is intended to be unordered.

Attributes

None.

Associations

- Specialize Class RDFSContainer

Constraints

None.

Semantics

See discussion in [RDF Concepts] of container membership semantics.

10.7.3 RDFList

Description

This class represents descriptions of RDF collections, conventionally called lists, and other list-like structures.

Attributes

None.

Associations

- originalList: RDFList [0..*] in association RestOfList - the original list for rdf:rest.
- RDFfirst: RDFSResource [0..1] in association FirstElementInList - links a list to its first element.
- RDFrest: RDFList [0..1] in association RestOfList - links a list to its sublist excluding its first element.
- Specialize Class RDFSResource.

Constraints

None.

Semantics

`rdf:Nil` is a predefined instance of `rdf:List` that explicitly denotes the termination of an `rdf:List`. Since `rdf:Nil` is at the model level, it is not explicitly represented, outside of the model library provided in Annex A.

10.7.4 RDFSContainer

Description

This is a super-class of RDF container classes.

Attributes

None.

Associations

- Specialize Class RDFSResource

Constraints

None.

Semantics

The same resource may appear in a container more than once. A property of a container is not necessarily a property of all of its members.

10.7.5 RDFSContainerMembershipProperty

Description

The `rdfs:ContainerMembershipProperty` class has as instances the properties `rdf:_1`, `rdf:_2`, `rdf:_3` ... that are used to state that a resource is a member of a container. Each instance of this class is an `rdfs:subPropertyOf` the `rdfs:memberOf` property.

Attributes

None.

Associations

- Specialize Class RDFProperty

Constraints

None.

Semantics

Container membership properties may be applied to resources other than containers. (Note that the blank nodes are intended to be interpreted as existentially quantified variables representing instances of URIs in `rdf:Property`.) The instances that make up this class are provided in the model library given in Annex A.

10.7.6 RDFSeq

Description

This is the class of RDF "Sequence" containers. It is used conventionally to indicate that the numerical ordering of the container membership properties of the container is intended to be significant.

Attributes

None.

Associations

- Specialize Class RDFSContainer

Constraints

None.

Semantics

See discussion in [RDF Concepts] of container membership semantics.

10.7.7 RDFSResource (Augmented Definition)

Associations

- container: RDFSResource [0..*] in association MemberOfResource - relates a particular resource to other resources that are its members.
- list: RDFList [0..*] in association FirstElementInList - relates a particular resource to the list(s) for which it is the initial element.
- RDFSmember: RDFSResource [0..*] in association MemberOfResource - relates a resource to another resource of which it is a member (i.e., a resource that contains it).

10.8 RDF Sources, Datasets, Documents and Namespaces

Figure 10.8 specifies several concepts that link an RDF document to the names and statements it contains. While both documents and graphs may have sets of statements associated with them, namespace definitions, and the mappings between namespace prefixes and URIs are associated with RDF documents (in this simplified view of XML Schema - in actuality, they are associated with XML elements), not with RDF graphs.

Note that the model supports multiple graphs within a document, and the notion that a particular graph may cover multiple documents. While in common practice there can be a one to one correspondence between a document and a graph, examples of both kinds of exceptions are included in the set of RDF specifications defining the language and in related W3C documents.

Single graph covering multiple documents. The ability to refer to definitions that are external to a particular document (e.g., XML Schema Datatypes) and in OWL, the ability to directly import such definitions, naturally extends a graph beyond the boundaries of a single document. Additionally, in [RDF Primer], there is a discussion of the use of XML Base, such that relative URIs may be defined based on a base URI other than that of the document in which they occur. This may be appropriate, for example, when there are mirror sites that share common definitions and extend them at the mirror site, but where it is not necessary to duplicate all definitions at every such site. In such cases, a graph can span multiple documents, and the URI of the mirror site document is distinct from that of its base. As a result, the metamodel provides for the optional definition of an `xml:base` distinct from the URI of the document.

Multiple graphs in the same document. It is common practice in ontology development to have multiple “main nodes” in the same document - for example, multiple concepts whose parent class is simply `owl:Thing`, or classes without a defined “parent class” in RDF. Some explicit examples are provided in the discussion of Named Graphs (see http://www.w3.org/2004/03/trix/, particularly those given on the TriG Homepage, at http://www.wiwiss.fu-berlin.de/suhl/bizer/

TriG/). One can imagine others such as when defining SKOS-based concept schemes, or thesauri, and managing multiple versions of such schemes (see the SKOS Core Guide, http://www.w3.org/TR/swbp-skos-core-guide, and http://www.w3.org/TR/swbp-thesaurus-pubguide, for more information). The ability to name a graph provides a means by which multiple component graphs defined in the same document can be referenced externally as a unit, enabling graph mapping and alignment, for example. Thus, the NamedGraph class can be used to support naming graphs for those applications that require this feature.

Bounding an RDF vocabulary. The notion of scope is somewhat opaque in the current set of recommendations that together define RDF and its vocabulary language, RDF Schema. This is, in part, due to the fact that URIs have global scope in RDF. Yet, we need a way of talking about and modeling the set of resources that describe a particular vocabulary. Each document is associated with a resource whose URI reference is the primary URL where the document is published. It is good practice to include this URL in the serialized form of an RDF XML document, as the value of an `xml:base` on its root element. The bounds of a particular RDF vocabulary is the collection of statements (triples) sharing a base URI, or, in the absence of such a URI, a graph whose base URI is, by default, that of the document that contains it.

Qualified Names and Transformations. Instructions regarding how QNames and `rdf:ID` attribute values can be transformed into RDF URI references are defined in [RDF Syntax]. Additionally, RDF/XML allows further abbreviating RDF URI references through the use of the XML Infoset mechanism for setting a base URI that is used to resolve relative RDF URI references (`xml:base`), or by considering the base URI to be that of the document. The base URI applies to all RDF/XML attributes that deal with RDF URI references, including `rdf:about`, `rdf:resource`, `rdf:ID`, and `rdf:datatype`. (See http://www.w3.org/TR/xmlbase/ for more on XML Base.)

Secondly, the `rdf:ID` attribute on a node element (not property element) can be used instead of `rdf:about` and gives a relative RDF URI reference equivalent to # concatenated with the `rdf:ID` attribute value. So for example if `rdf:ID="`*`name`*`"`, that would be equivalent to `rdf:about="#`*`name`*`"`. rdf:ID provides an additional check since the same name can only appear once in the scope of an `xml:base` value (or document, if none is given), so is useful for defining a set of distinct, related terms relative to the same RDF URI reference.

Both forms require a base URI to be known, either from an in-scope `xml:base`, or, in the case of a reference to a definition outside of the current document, from the URI of the RDF/XML document in which the target definition is specified.

Note that, regardless of the fact that various XML techniques may be used to make the RDF/XML more concise and readable, this metamodel represents only the resultant full IRI regardless of how it was represented in the serialized document.

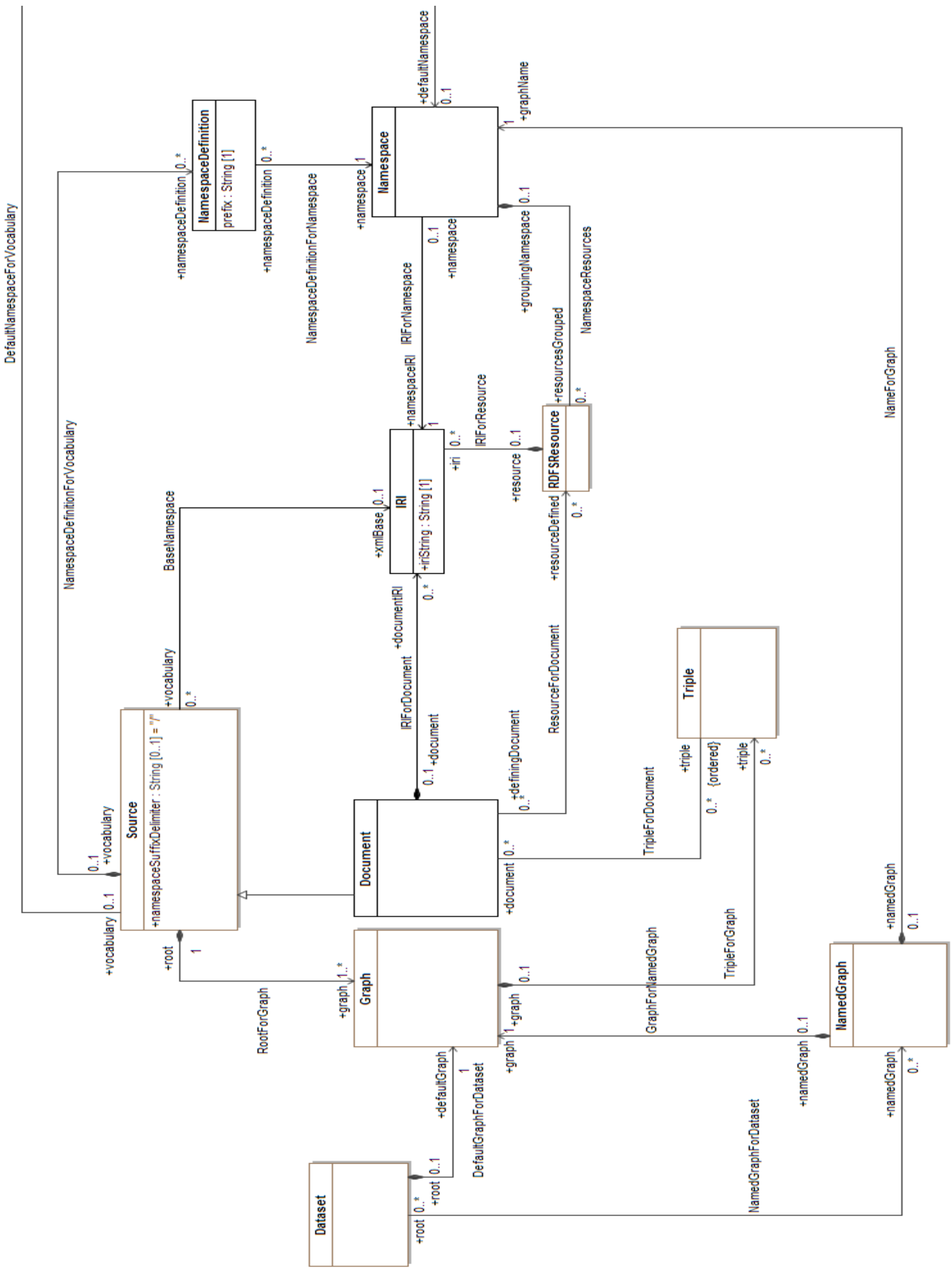

Figure 10.8 - RDF Sources, Datasets, Documents, and Namespaces Diagram

10.8.1 Document

Description

RDF's conceptual model is a graph. RDF also provides an XML syntax for writing down and exchanging RDF graphs, called RDF/XML. An RDF document is a serialization of an RDF graph into a concrete syntax, as specified in [RDF Syntax], which provides the container for the graph, and conventionally also contains declarations of the XML namespaces referenced by the statements in the document.

RDF refers to a set of IRIs, intended for use in an RDF Graph, as a vocabulary. Often, the IRIs in such vocabularies are organized so that they can be represented as sets of QNames using common prefixes. IRIs that are contained in the vocabulary are formed by appending individual local names to the relevant prefix. This practice is also commonly used in OWL ontology development for improved readability. While the metamodel does not explicitly support QNames, the elements required to enable such support in vendor implementations are provided.

Attributes

None.

Associations

- triple: Triple[1..*] in association TripleForDocument - links a document to the set of triples it contains.
- xmlBase: Namespace [0..*] in association DefaultNamespaceForDocument - links a document to one or more default namespaces (xml:base namespaces) associated with the statements in the document.
- resourceDefined:RDFSResource[0..*] - the resources defined in this document.
- Specialize Class Source.

Constraints

None.

Semantics

An RDF/XML document is only required to be well-formed XML; it is not intended to be validated against an XML DTD (or an XML Schema).

10.8.2 Namespace

Description

An XML namespace is a collection of names, identified by a IRI, which are used in XML documents as element types and attribute names.

Attributes

None.

Associations

- namespaceDefinition: NamespaceDefinition [0..*] in association NamespaceForNamespaceDefinition - links a namespace definition to the namespace it describes (resolves to).
- namespaceIRI: IRI [1] in association IRINamespace - links a namespace to the corresponding IRI.

- resourcesGrouped:RDFSResource [0..*] - the resources that are grouped by this namespace, i.e., that are based on its IRI.

Constraints

[1] Namespaces should conform to the specification given in "[XMLNS]" on page 4. While it may not be possible to define constraints on character strings in OCL to enforce this (and while the namespace recommendation may not explicitly require enforcement), tools that implement this metamodel will be expected to support the W3C standards and related RFCs to the extent possible.

Semantics

None.

10.8.3 NamespaceDefinition

Description

A namespace is declared using a family of reserved attributes. These attributes, like any other XML attributes, may be provided directly or by default. Some names in XML documents (constructs corresponding to the non-terminal Name) may be given as qualified names. The prefix provides the namespace prefix part of the qualified name, and must be associated with a namespace IRI in a namespace declaration.

Namespace definitions are used in RDF and OWL for referencing and/or importing externally specified terms, vocabularies, or ontologies.

Attributes

- namespacePrefix: String [1] - the string representing the namespace prefix.

Associations

- vocabulary: Source [1] - the document or other source containing the namespace definition.
- namespace: Namespace [1] in association NamespaceDefinitionForNamespace - indicates that a namespace definition, if it exists, resolves to exactly one namespace.

Constraints

[1] Namespace definitions should conform to the specification given in [XMLNS].

Semantics

None.

10.8.4 Triple

Associations

- document: Document [1..*] in association TripleForDocument - the document(s) containing the triple.
- graph: Graph [0..1] - the logical graph containing the triple.

10.8.5 NamedGraph

Description

A named graph is an IRI and RDF graph pair. It effectively provides a way to name an RDF graph and thus refer to the graph.

Attributes

None.

Associations

- graphForNG: Graph [1] the graph that is being named.
- graphname: Namespace[1] - the name for the graph (as a Namespace).
- Specialize class RDFResource.

Constraints

[1] The multiplicity on the derived ResourceForReference association on the iri role must be 1 for NamedGraphs.

Semantics

A named graph is a first class object that represents an RDF graph. It is named with an IRI. Two relationship types are predefined for relationships among named graphs (EquivalentGraph and SubGraphOf). These assert equivalence and subset relationships respectively among the RDF graphs (in the graphForNG role) that correspond to the named graphs linked by these relationships.

10.8.6 Graph

Description

An RDF graph is a set of RDF triples. The set of nodes of an RDF graph is the set of subjects and objects of triples in the graph.

A number of classes in the metamodel, including Graph, RDFStatement, Document, etc., are included (1) for the sake of completeness, and (2) are provided for vendors to use, as needed from an application perspective. They may not be necessary for all tools, and may not necessarily be accessible to end users, again, depending on the application requirements.

Attributes

None.

Associations

- namedGraph: NamedGraph [0..*] in association GraphForNamedGraph - links an RDF graph with named graphs that may represent it.
- triple: Triple [0..*] in association TripleForGraph - links a graph to the triples it contains.

Constraints

None.

Semantics

As described in [RDF Semantics], RDF is an assertional language, intended for use in defining formal vocabularies and using them to state facts and axioms about some domain.

An RDF graph is defined as a set of RDF triples.

The assertion of an RDF triple says that some relationship, indicated by the predicate, holds between the things denoted by subject and object of the triple. The assertion of an RDF graph amounts to asserting all the triples in it, so the meaning of an RDF graph is the conjunction (logical AND) of the statements corresponding to all the triples it contains.

10.8.7 Dataset

Description

A RDF dataset is a collection of RDF graphs. Blank nodes may be shared between graphs in an RDF dataset.

Attributes

None.

Associations

- defaultGraph: Graph [1] the default graph for the dataset, which does not have a name and may be empty.
- namedGraph: namedGraph[0..*] - zero or more named graphs; each named graph is a pair consisting of the graph name and an RDF graph.

Constraints

None.

10.8.8 Source

Description

An RDF source is a persistent yet mutable source or container of RDF graphs. It is a resource that may be said to have a state that can change over time. A snapshot of the state can be expressed as an RDF graph. For example, any web document that has an RDF-bearing representation may be considered an RDF source. Like all resources, RDF sources may be named with IRIs and therefore described in other RDF graphs.

Attributes

- namespaceSuffixDelimiter: String [0..1] - the default is “/” and should not need to be specified; used to indicate when “#” is preferred.

Associations

- defaultNamespace: NameSpace [0..1] - an optional default namespace for the source.
- namespaceDefinition: NamespaceDefinition [0..*] – zero or more namespace definitions for the source, relating namespaces to preferred namespace prefixes, including an optional namespace prefix for the source as well as others used in the context of the source.
- xmlBase: IRI [0..1] – an optional xmlbase allowing concise identification of elements in the source and, informally, linking them together in a vocabulary.

- graph: graph [1..*] – one or more graphs associated with the source.

Constraints

[1] Either a defaultNamespace or at least one xmlBase must be specified.

[2] The string value for any defaultNamespace property must conform to the character encoding defined in RFC 3986.

[3] The string value for any xmlBase property must conform to the character encoding defined in RFC 3986.

11 The OWL Metamodel

11.1 Overview

The Web Ontology Language (OWL) is a semantic markup language for publishing and sharing ontologies on the World Wide Web. Where earlier knowledge representation languages have been used to develop tools and ontologies for specific user communities (particularly in the sciences and in company-specific e-commerce applications), they were not defined to be compatible with the architecture of the World Wide Web in general, and the Semantic Web in particular.

OWL uses both URIs for naming and the description framework for the Web provided by RDF to add the following capabilities to ontologies:

- Ability to be distributed across many systems
- Scalability to Web needs
- Compatibility with Web standards for accessibility and internationalization
- Openness and extensibility

OWL builds on RDF and RDF Schema and augments the RDFS vocabulary for describing properties and classes: among others, relations between classes (e.g., disjointedness), cardinality (e.g., "exactly one"), equality, richer typing of properties, characteristics of properties (e.g., symmetry), and enumerated classes.

The OWL Metamodel is a MOF2 compliant metamodel that allows a user to specify ontologies using the terminology and underlying model theoretic semantics of OWL [OWL S&AS]. The OWL Metamodel extends the set of metamodels defined herein for RDFBase, RDFS (RDF Schema), and RDFWeb.

OWL provides three increasingly expressive sublanguages designed for use by specific communities of users and implementors:

- OWL Lite - which supports users primarily needing a classification hierarchy and simple constraints.
- OWL DL - which supports users who want maximum expressiveness without losing computational completeness and decidability of reasoning systems.
- OWL Full - which is intended for users who want maximum expressiveness and the syntactic freedom of RDF without computational guarantees.

Based on requirements derived from the usage scenarios described in Clause 7, Usage Scenarios and Goals, the ODM was designed to enable ontology development using either OWL DL or OWL Full, which essentially share abstract syntax constructs and differ primarily in terms of constraints. We have not explicitly covered OWL Lite, but all constructs and many relevant constraints are provided in the base OWL and OWL DL packages. Vendors who are interested in supporting OWL Lite can simply use the relevant constructs from the base package and tighten constraints from the OWL DL package, as required.

11.1.1 Organization of the OWL Metamodel

The primary OWLBase package contains the metamodel constructs common to both OWL DL and OWL Full - corresponding to the abstract syntax elements of the Web Ontology Language. Two additional sub packages contain constraints required to distinguish the two dialects (OWLDL and OWLFull) from one another. From a compliance perspective, vendors can elect to support the primary package and either or both of the subordinate packages in order to have complete coverage of either or both dialects of OWL. The package structure for the OWL metamodel and its dependencies on the RDF metamodel are shown in Figure 11.1.

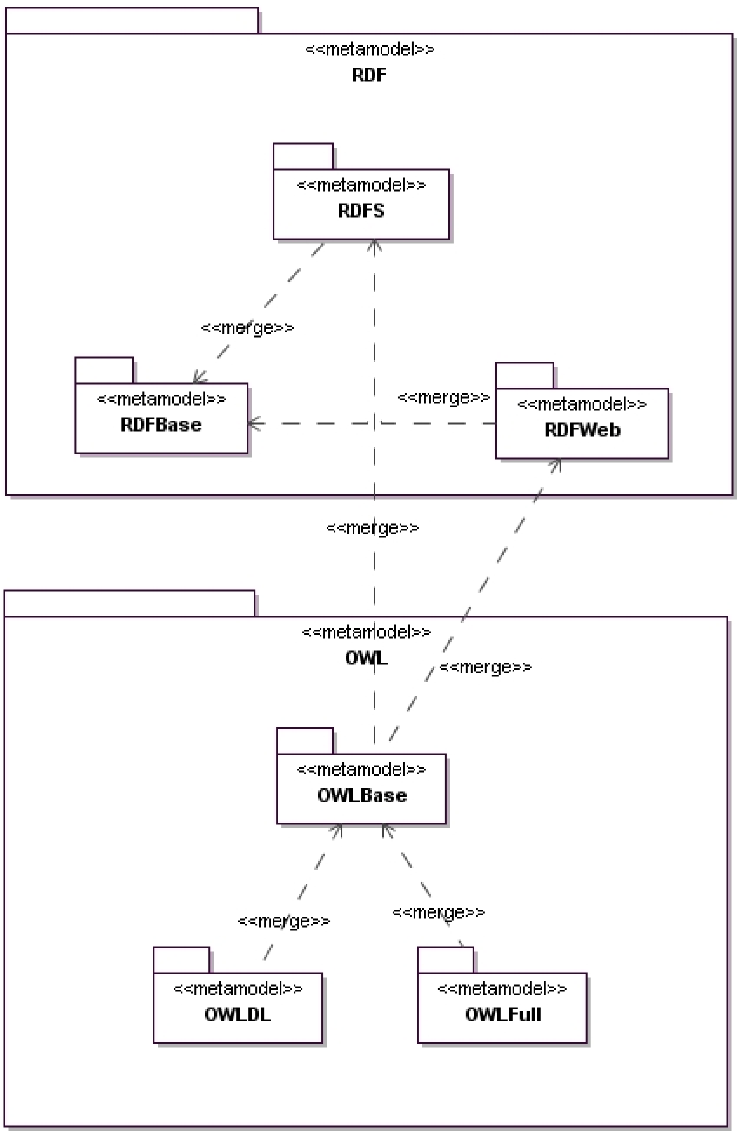

Figure 11.1 - The OWL Metamodel Package Structure

11.1.2 Design Considerations

11.1.2.1 Naming

As in the RDF metamodels, prefixes are used in naming MOF classes and MOF properties that directly represent OWL classes and OWL properties, respectively. For example, OWLClass represents `owl:Class` and OWLimports represents `owl:imports`. Individual, which does not have a prefix, represents something that is not explicitly defined in the RDF/XML serialization of OWL. Exceptions to this convention include OWLUniverse and OWLStatement, included for vendor use in mapping RDF graphs and/or statements to OWL, for mapping to other metamodels, and so forth.

11.2 OWLBase Package - OWL Ontology

As shown in Figure 11.2, an OWL ontology consists of a collection of facts, axioms, and annotations defined in terms of RDF graphs and triples. The ontologyID (in the form of the URI reference it has by virtue of being a resource) allows us to make statements about a particular ontology - including annotations such as the relationship between a particular ontology and other ontologies, version information, and so forth.

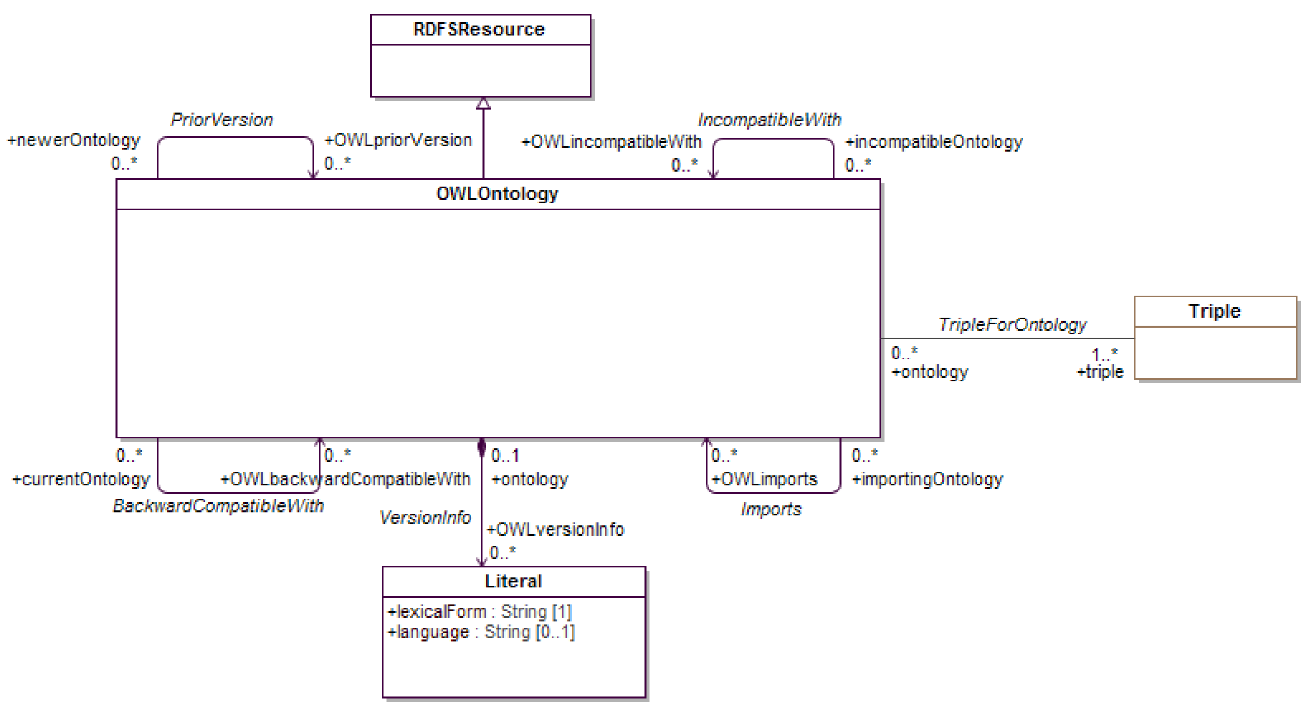

Figure 11.2 - The Ontology Diagram

11.2.1 OWLOntology

Description

An OWL ontology contains a sequence of annotations, axioms, and facts. Annotations on OWL ontologies can be used to record authorship and other information associated with an ontology, including imports references to other ontologies. The main content of OWLOntology is carried in its axioms and facts, which provide information about classes, properties, and individuals in the ontology.

Names of ontologies are used in the abstract syntax to carry the meaning associated with publishing an ontology on the Web. The intent is that the name of an ontology in the abstract syntax is the URI where it can be found, although this is not part of the formal meaning of OWL. Imports annotations, in effect, are directives to retrieve a Web document and treat it as an OWL ontology.

Attributes

None.

Associations

- currentOntology: OWLOntology [0..*] in association BackwardCompatibleWith - links an ontology to zero or more other ontologies it has backwards compatibility with.
- OWLbackwardCompatibleWith: OWLOntology [0..*] in association BackwardCompatibleWith - links an ontology to zero or more other ontologies it has backwards compatibility with.
- importingOntology: OWLOntology [0..*] in association Imports - links an ontology to zero or more other ontologies it imports.
- OWLimports: OWLOntology [0..*] in association Imports - links an ontology to zero or more other ontologies it imports.
- incompatibleOntology: OWLOntology [0..*] in association IncompatibleWith - links an ontology to zero or more other ontologies it is not compatible with (typically used to say that a newer version of a particular ontology introduces destructive changes from a prior version).
- OWLincompatibleWith: OWLOntology [0..*] in association IncompatibleWith - links an ontology to zero or more other ontologies it is not compatible with.
- newerOntology: OWLOntology [0..*] in association PriorVersion - links an ontology to zero or more other ontologies that are earlier versions of the current ontology.
- OWLpriorVersion: OWLOntology [0..*] in association PriorVersion - links an ontology to zero or more other ontologies that are earlier versions of the current ontology.
- OWLversionInfo: RDFSLiteral [0..*] in association VersionInfo - links an ontology to an annotation providing version information.
- triple: Triple [1..*] in association TripleForOntology - links an ontology to one or more triples it contains.
- Specialize Class RDFSResource.

Constraints

None.

Semantics

The semantics of OWL ontology are described in [OWL S&AS].

An `owl:imports` statement references another OWL ontology containing definitions, whose meaning is considered to be part of the meaning of the importing ontology. Each reference consists of a URI specifying from where the ontology is to be imported. Syntactically, `owl:imports` is a property with the class `owl:Ontology` as its domain and range.

The `owl:imports` statements are transitive, that is, if ontology A imports B, and B imports C, then A imports both B and C. Importing an ontology into itself is considered a null action, so if ontology A imports B and B imports A, then they are considered to be equivalent.

An `owl:versionInfo` statement generally has as its object a string giving information about this version, for example RCS/CVS keywords. This statement does not contribute to the logical meaning of the ontology other than that given by the RDF(S) model theory. Although this property is typically used to make statements about ontologies, it may be applied to any OWL construct.

An `owl:priorVersion` statement contains a reference to another ontology. This identifies the specified ontology as a prior version of the containing ontology. This has no meaning in the model-theoretic semantics other than that given by the RDF(S) model theory. However, it may be used by software to organize ontologies by versions.

An `owl:backwardCompatibleWith` statement contains a reference to another ontology. This identifies the specified ontology as a prior version of the containing ontology, and further indicates that it is backward compatible with it. In particular, this indicates that all identifiers from the previous version have the same intended interpretations in the new version. Thus, it is a hint to document authors that they can safely change their documents to commit to the new version (by simply updating namespace declarations and `owl:imports` statements to refer to the URL of the new version). If `owl:backwardCompatibleWith` is not declared for two versions, then compatibility should not be assumed.

An `owl:incompatibleWith` statement contains a reference to another ontology. This indicates that the containing ontology is a later version of the referenced ontology, but is not backward compatible with it. Essentially, this is for use by ontology authors who want to be explicit that documents cannot upgrade to use the new version without checking whether changes are required.

11.2.2 RDFSLiteral (Augmented Definition)

Associations

- dataRange: OWLDataRange [0..*] in association ElementsForDataRange - links one or more literals in an enumerated list to zero or more OWL DataRanges.
- ontology: OWLOntology [1] in association VersionInfo - links an `owl:versionInfo` annotation to the ontology it describes.
- restrictionClass: HasValueRestriction [0..*] in association HasLiteralValue - optionally links one literal to a has value property restriction.

11.3 OWLBase Package - Class Expressions

As described in [OWL Reference], classes provide an abstraction mechanism for grouping resources with similar characteristics. Like RDF classes, every OWL class can be associated with a set of individuals, called the *class extension*. The individuals in the class extension are called the instances of the class. A class has an intensional meaning (the underlying concept) that is related but not equal to its class extension. Thus, two classes may have the same class extension, but still be different classes. OWL classes are described through “class expressions,” which can be combined into “class axioms.”

A class expression, as shown in Figure 11.3, describes an OWL class, either by a class name or by specifying the class extension of an unnamed anonymous class. OWL distinguishes six types of class expressions:

1. a class identifier (an IRI)
2. an exhaustive enumeration of individuals

3. a property restriction
4. the intersection of class expressions
5. the union of class expressions
6. the complement of a class expression

The first type is special in the sense that it describes a class through a class name (syntactically represented as a URI reference). The other five types of class expressions describe an anonymous class by placing constraints on the class extension.

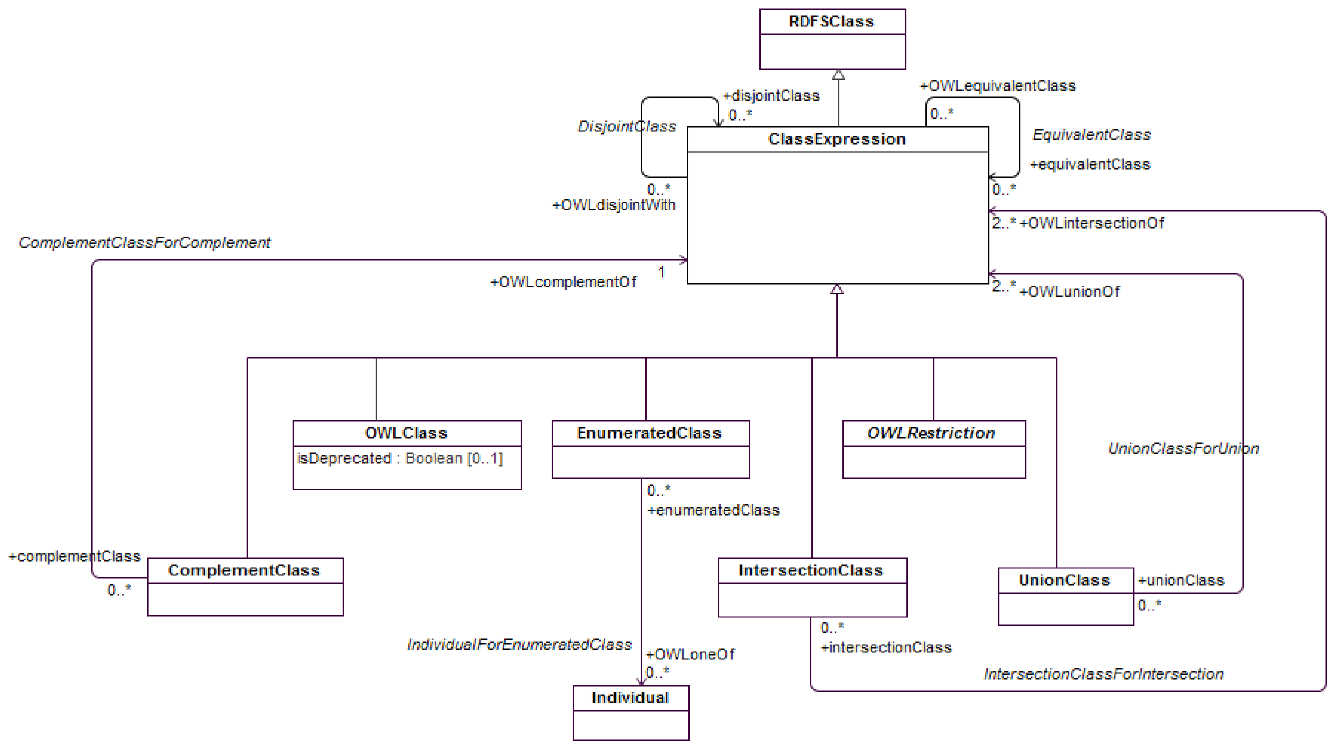

Figure 11.3 - The OWL Class Expressions Diagram

OWL property restrictions describe special kinds of class expressions, that may (or may not) be anonymous classes, consisting of all individuals that satisfy the restriction, as shown in Figure 11.4.

OWL distinguishes two kinds of property restrictions: value constraints and cardinality constraints. OWL value constraints are used to constrain the range of a property when applied to the particular class expression, which is distinct from the concept of an `rdfs:range` property (which is essentially global and applies to all cases where the property is used). OWL cardinality constraints are similar to UML multiplicities, and constrain the number of values a property can have, again in the context of the particular class expression it is applied to.

The three types of advanced class constructors that are used in Description Logic can be viewed as representing the AND, OR, and NOT operators on classes. The corresponding operators have the standard set-operator names: intersection, union, and complement. These language constructs also share the characteristic that they can contain nested class expressions, either one (complement) or more (union, intersection).

11.3.1 ClassExpression

Description

A class expression describes an OWL class, either by a class name or by specifying the class extension of an unnamed anonymous class.

Attributes

None.

Associations

- complementClass: ComplementClass [0..*] in association ComplementClassForComplement - links a class to another class defined as its set complement.
- disjointClass: OWLClass [0..*] in association DisjointWith - links a class to zero or more classes that it is disjoint with.
- OWLdisjointWith: OWLClass [0..*] in association DisjointWith - links a class to zero or more classes that it is disjoint with.
- equivalentClass: OWLClass [0..*] in association EquivalentClass - links a class to zero or more classes that it is considered equivalent to.
- OWLequivalentClass: OWLClass [0..*] in association EquivalentClass - links a class to zero or more classes that it is considered equivalent to.
- intersectionClass: IntersectionClass [0..*] in association IntersectionClassForIntersection - links a class to zero or more intersections that it participates in.
- restrictionClass: AllValuesFromRestriction [0..*] in association AllValuesFromClass - links a class to an owl:allValuesFrom restriction for which it provides the range (or set of values).
- restrictionClass: SomeValuesFromRestriction [0..*] in association SomeValuesFromClass - links a class to an owl:someValuesFrom restriction for which it provides the range (or set of values).
- unionClass: UnionClass [0..*] in association UnionClassForUnion - links a class to zero or more unions that it participates in.
- Specialize Class RDFSClass.

Constraints

Except for OWLClass, ClassExpressions must be anonymous: specifically the property IRIForResource must be empty.

Semantics

See the formal [OWL S&AS] for additional semantics.

11.3.2 ComplementClass

Description

An `owl:complementOf` statement describes a class for which the class extension contains exactly those individuals that do not belong to the class extension of the class expression that is the object of the statement. It is analogous to logical negation: the class extension consists of those individuals that are NOT members of the class extension of the complement class.

Attributes

None.

Associations

- OWLcomplementOf: ClassExpression [1] in association ComplementClassForComplement - links a class to its set complement.
- Specialize Class ClassExpression

Constraints

None.

Semantics

See the formal [OWL S&AS] for additional semantics.

11.3.3 EnumeratedClass

Description

A class expression of the "enumeration" kind is defined with the `owl:oneOf` property. The value of this built-in OWL property must be a list of individuals that are the instances of the class. This enables a class to be described by exhaustively enumerating its instances. The class extension of a class described with `owl:oneOf` contains exactly the enumerated individuals, no more, no less. The list of individuals is typically represented with the help of the RDF construct `rdf:parseType="Collection"` that provides a convenient shorthand for writing down a set of list elements.

Attributes

None.

Associations

- OWLoneOf: Individual [0..*] in association EnumeratedClassForIndividual - links a class to the list of individuals that are its instances.
- Specialize Class ClassExpression

Constraints

[1] The set of individuals specified represents a complete definition of the class extension.

Semantics

Note that use of an enumeration presumes that the class extension is complete. Also, the elements of an enumerated class are not necessarily unique and no unique names assumption applies.

11.3.4 Individual

Description

Individuals are defined with individual axioms (also called "facts"). Two types of facts are supported in OWL: (1) Facts about class membership and property values of individuals, and (2) Facts about individual identity. Many facts are statements that define class membership of individuals and property values of individuals; these can also refer to anonymous individuals.

Attributes

None.

Associations

- allDifferent: OWLAllDifferent [0..*] in association DistinctIndividuals - links an individual to an idiomatic class, OWLAllDifferent, of which it is a member, indicating that it is pairwise disjoint with (unique from) the other members of the class.
- enumeratedClass: EnumeratedClass [0..*] in association IndividualForEnumeratedClass - links an individual to zero or more enumerated classes of which it is a member.
- differentIndividual: Individual [0..*] in association DifferentIndividual - links an individual to another individual that it is different from (pairwise disjoint with).
- OWLdifferentFrom: Individual [0..*] in association DifferentIndividual - links an individual to another individual that it is different from (pairwise disjoint with).
- sameIndividual: Individual [0..*] in association SameIndividual - links an individual to another individual that it is equal to (the same as).
- OWLsameAs: Individual [0..*] in association SameIndividual - links an individual to another individual that it is equal to (the same as).
- restrictionClass: HasValueRestriction [0..*] in association HasIndividualValue - links an individual to a restriction class for which it represents the value.
- Specialize Class RDFSResource.

Constraints

No additional constraints.

Semantics

Note that individuals in OWL have a "default type" (i.e., `owl:Thing)`, and can have zero or more explicit "types" (i.e., can be members of zero or more classes in addition to `owl:Thing`). What this means is that one can say that an individual exists in an OWL ontology without necessarily saying anything about its class membership, other properties it may have, or the values for any properties it may have. This is common in ontology development, unlike more traditional UML modeling. Multiple inheritance is also supported. See the formal [OWL S&AS] for additional semantics.

11.3.5 OWLClass

Description

An OWLClass is a class that is a named resource.

Attributes

- isDeprecated: Boolean [0..1] - indicates that use of this class expression is deprecated.

Associations

- Specialize Class ClassExpression

Constraints

No additional constraints.

Semantics

No additional semantics.

11.3.6 IntersectionClass

Description

The `owl:intersectionOf` property links a class to a list of class expressions. An `owl:intersectionOf` statement describes a class for which the class extension contains precisely those individuals that are members of the class extension of all class expressions in the list.

Attributes

None.

Associations

- OWLintersectionOf: ClassExpression [2..*] in association IntersectionClassForIntersection - links an intersection class to the classes participating in the intersection.
- Specialize Class ClassExpression

Constraints

No additional constraints.

Semantics

`owl:intersectionOf` can be viewed as being analogous to logical conjunction.

11.3.7 UnionClass

Description

The `owl:unionOf` property links a class to a list of class expressions. An `owl:unionOf` statement describes an anonymous class for which the class extension contains those individuals that occur in at least one of the class extensions of the class expressions in the list.

Attributes

None.

Associations

- OWLunionOf: ClassExpression [2..*] in association UnionClassForUnion - links a union class to the class expressions that participate in the union.
- Specialize class ClassExpression.

Constraints

No additional constraints.

Semantics

`owl:unionOf` is analogous to logical disjunction.

11.3.8 Restrictions

The OWL Restrictions diagram is shown below.

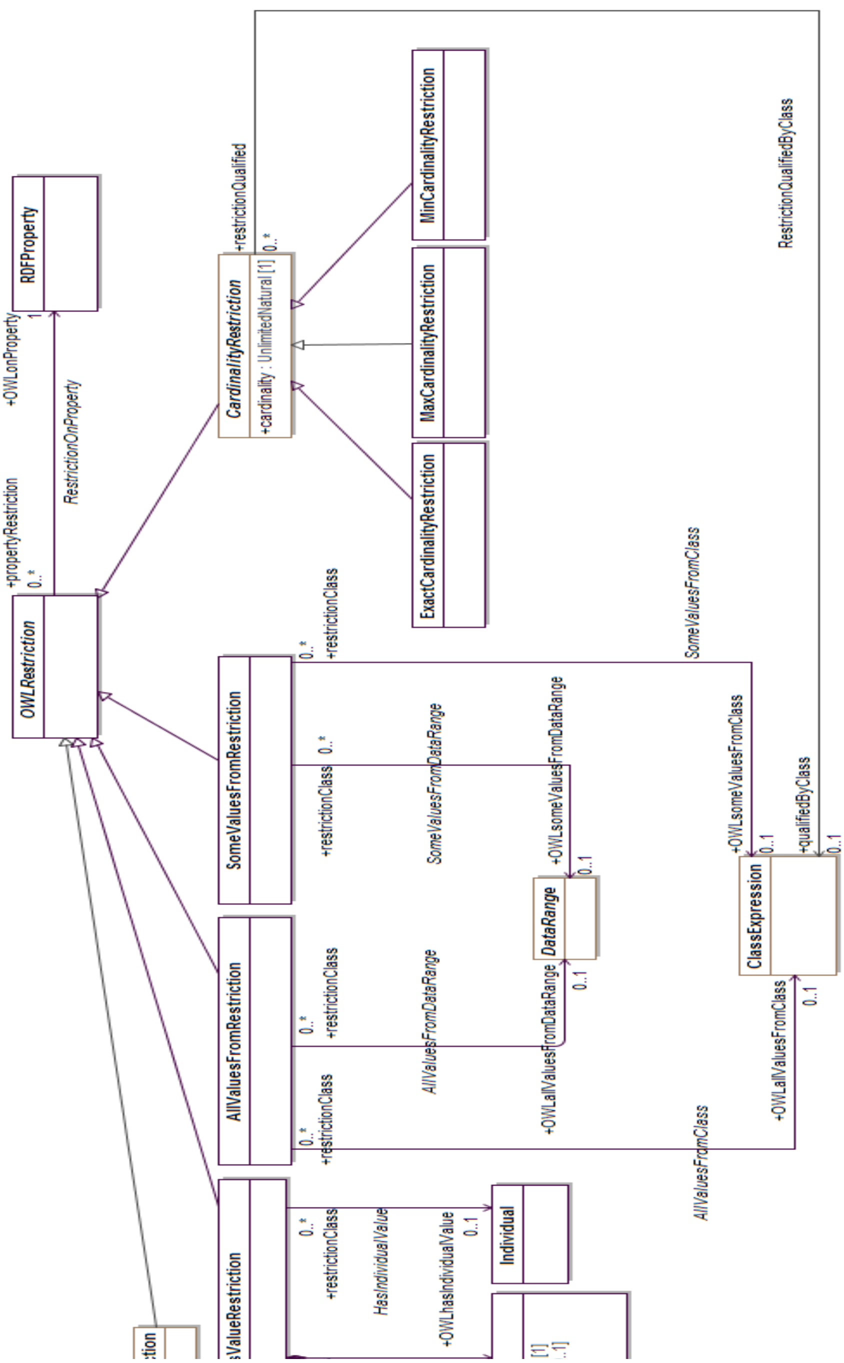

Figure 11.4 - The OWL Restrictions Diagram

11.3.8.1 OWLRestriction

Description

The class `owl:Restriction` is defined as a subclass of `owl:Class`. A restriction class should have exactly one triple linking the restriction to a particular property, using the `owl:onProperty` property. The restriction class should also have exactly one triple that represents the value or cardinality constraint on the property under consideration, e.g., that the cardinality of the property is exactly 1. The restriction may optionally have a triple that qualifies the value of the restricted property to have individuals of a specified class

Property restrictions can be applied both to datatype properties (properties for which the value is a data literal) and object properties (properties for which the value is an individual).

Attributes

None.

Associations

- OWLonProperty: RDFProperty [1] in association RestrictionOnProperty - links an OWL restriction class to the property that constrains it.
- Specialize Class ClassExpression

Constraints

[1] A restriction class must have exactly one number or value constraint.

Semantics

See the formal [OWL S&AS] for additional semantics.

11.3.8.2 CardinalityRestriction

Description

This abstract class represents properties common to all cardinality restrictions.

Attributes

- cardinality:UnlimitedNatural [1] – represents the cardinality used for the restriction. The interpretation varies by the subclass for the specific kind of cardinality restriction.

Associations

- qualifiedByClass:ClassExpression [0..1] – represents an optional qualification of the restriction by class membership – if not empty then the property must satisfy not only the cardinality but the class membership.
- Specialize Class OWLRestriction.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.3.8.3 ExactCardinalityRestriction

Description

The cardinality constraint `owl:cardinality` is a built-in OWL property that links a restriction class to a data value belonging to the range of the XML Schema datatype `xsd:nonNegativeInteger`. A restriction containing an `owl:cardinality` constraint describes a class of all individuals that have exactly N semantically distinct values (individuals or data values) for the property concerned, where N is the value of the cardinality constraint. Syntactically, the cardinality constraint is represented as an RDF property element with the corresponding `rdf:datatype` attribute.

Attributes

None.

Associations

- Specialize Class CardinalityRestriction.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.3.8.4 MaxCardinalityRestriction

Description

The cardinality constraint `owl:maxCardinality` is a built-in OWL property that links a restriction class to a data value belonging to the value space of the XML Schema datatype `xsd:nonNegativeInteger`. A restriction containing an `owl:maxCardinality` constraint describes a class of all individuals that have *at most* N semantically distinct values (individuals or data values) for the property concerned, where N is the value of the cardinality constraint. Syntactically, the cardinality constraint is represented as an RDF property element with the corresponding `rdf:datatype` attribute.

Attributes

None.

Associations

- Specialize Class CardinalityRestriction.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.3.8.5 MinCardinalityRestriction

Description

The cardinality constraint `owl:minCardinality` is a built-in OWL property that links a restriction class to a data value belonging to the value space of the XML Schema datatype `xsd:nonNegativeInteger`. A restriction containing an `owl:minCardinality` constraint describes a class of all individuals that have *at least* N semantically distinct values (individuals or data values) for the property concerned, where N is the value of the cardinality constraint. Syntactically, the cardinality constraint is represented as an RDF property element with the corresponding `rdf:datatype` attribute.

Attributes

None.

Associations

- Specialize Class CardinalityRestriction.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.3.8.6 AllValuesFromRestriction

Description

An AllValuesFromRestriction describes a class for which all values of the property under consideration are either members of the class extension of the class expression or are data values within the specified data range. In other words, it defines a class of individuals x for which holds that if the pair (x, y) is an instance of P (the property concerned), then y should be an instance of the class expression or a value in the data range, respectively.

Attributes

None.

Associations

- OWLallValuesFromClass: ClassExpression [0..1] in association AllValuesFromClass - links the restriction class to the class expression containing all of the individuals in its range.
- OWLallValuesFromDataRange: DataRange [0..1] in association AllValuesFromDataRange - links the restriction class to the data range containing all of the data values in its range.
- Specialize Class OWLRestriction.

Constraints

[1] An AllValuesFromRestriction identifies either one ClassExpression or one DataRange through either the AllValuesFromClass association or the AllValuesFromDataRange association, respectively.

Semantics

An `owl:allValuesFrom` constraint is analogous to the universal (for-all) quantifier of Predicate logic - for each instance of the class that is being described, every value for P must fulfill the constraint.

11.3.8.7 HasValueRestriction

Description

A HasValueRestriction describes a class of all individuals for which the property concerned has at least one value semantically equal to V (it may have other values as well).

Attributes

None.

Associations

- OWLhasIndividualValue: Individual [0..1] in association HasIndividualValue - links the restriction class to the class expression containing the individual that fills its value role.
- OWLhasLiteralValue: RDFSLiteral [0..1] in association HasLiteralValue - links the restriction class to the literal that fills its value role.
- Specialize Class OWLRestriction.

Constraints

[1] A HasValueRestriction links to only one value, either an individual through OWLhasIndividualValue or a literal through OWLhasLiteralValue.

Semantics

No additional semantics.

11.3.8.8 SomeValuesFromRestriction

Description

A SomeValuesFromRestriction describes a class for which *at least one value* of the property under consideration is either a member of the class extension of the class expression or is a data value within the specified data range. In other words, it defines a class of individuals x for which there is at least one y (either an instance of the class expression or value in the data range) such that the pair (x, y) is an instance of P (the property concerned). This does not exclude that there are other instances (x, y') of P for which y' does not belong to the class expression or data range.

Attributes

None.

Associations

- OWLsomeValuesFromClass: ClassExpression [0..1] in association SomeValuesFromClass - links the restriction class to a class expression containing at least one of the values in its range.
- OWLsomeValuesFromDataRange: DataRange [0..1] in association SomeValuesFromDataRange - links the restriction class to a data range containing at least one of the data values in its range.

- Specialize Class OWLRestriction.

Constraints

[1] A SomeValuesFromRestriction identifies either one ClassExpression or one DataRange through either the SomeValuesFromClass association or the SomeValuesFromDataRange association, respectively.

Semantics

An `owl:someValuesFrom` constraint is analogous to the existential (there-exists) quantifier of Predicate logic - for each instance of the class that is being described, at least one value for P must fulfill the constraint.

11.3.8.9 HasSelfRestriction

Description

A HasSelfRestriction describes a class of all individuals for which the property concerned has a value semantically equal to the subject (it may have other values as well).

Attributes

None.

Associations

None.

Constraints

The onProperty domain and range must be the same.

Semantics

No additional semantics.

11.4 OWLBase Package - Properties

As shown in Figure 11.5, OWL refines the notion of an RDF property to support two main categories of properties as well as annotation properties that may be useful for ontology documentation:

- Object properties - which relate individuals to other individuals.
- Datatype properties - which relate individuals to data values.
- Annotation properties - which allow us to annotate various constructs in an ontology.
- Ontology properties - which allow us to say things about ontologies themselves.

The distinction made between kinds of annotation properties (i.e., annotation vs. ontology properties) are needed to support OWL DL semantics. In addition, a number of property axioms are provided for property characterization.

Note: Certain information regarding OWL property inheritance, for example whether or not a particular object or datatype property is also functional, may not be accessible to some applications due to issue #9466 regarding multiple classification in MOF. See Annex F for details on how to work around this until the MOF issue is adequately addressed and MOF tool support for multiple classification is available.

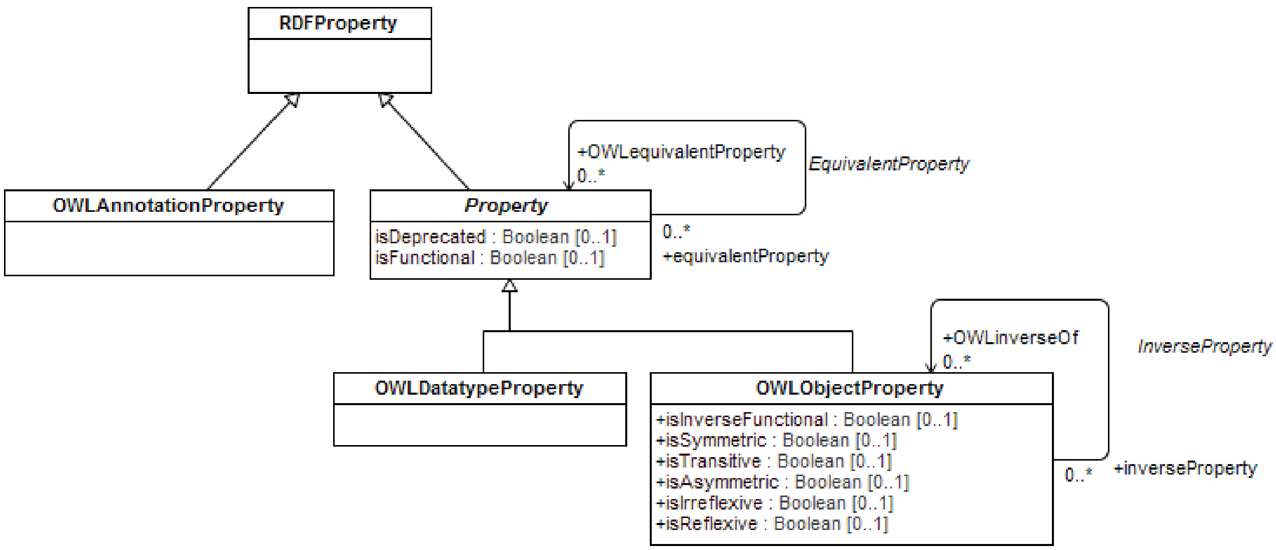

Figure 11.5 - The OWL Properties Diagram

11.4.1 OWLAnnotationProperty

Description

OWL Full does not put any constraints on annotations in an ontology. OWL DL allows annotations on classes, properties, individuals and ontology headers, as outlined in 11.8.1.

Five annotation properties are predefined by OWL, namely:

- `owl:versionInfo`
- `rdfs:label`
- `rdfs:comment`
- `rdfs:seeAlso`
- `rdfs:isDefinedBy`

In addition to the associations given in the metamodel representing these properties, they are defined in the model library provided in Annex A.

Attributes

None.

Associations

- Specialize Class RDFProperty.

Constraints

[1] The object of an annotation property must be RDFSLiteral, Individual, or URIReference.

Semantics

No additional semantics.

11.4.2 OWLDatatypeProperty

Description

Datatype properties are used to link individuals to data values. A datatype property is defined as an instance of the built-in OWL class `owl:DatatypeProperty`.

Attributes

None.

Associations

- Specialize Class *Property.*

Constraints

[1] The range of an OWLDatatypeProperty is restricted to the set of data values, i.e., a member of the class extension of RDFSLiteral or an instance of OWLDataRange.

context OWLDatatypeProperty RangeIsLiteral inv:

self.RDFSrange.oclIsKindOf (RDFSLiteral) or self.RDFSrange.oclIsKindOf (OWLDataRange)

Semantics

See the formal [OWL S&AS] for additional semantics.

11.4.3 OWLObjectProperty

Description

An object property relates an individual to other individuals. An object property is defined as an instance of the built-in OWL class `owl:ObjectProperty`.

Attributes

- isInverseFunctional: Boolean [0..1] - If a property is declared to be inverse-functional, then the object of a property statement uniquely determines the subject (some individual). More formally, if we state that isInverseFunctional = true for property P, then (x1, P, y) and (x2, P, y) are asserted, then x1 is the same as x2 (they cannot be distinct).

  Inverse-functional properties resemble the notion of a key in databases.

  Note that isInverseFunctional specifies global cardinality constraints. That is, no matter which class the property is applied to, the cardinality constraints must hold. This is different from the cardinality constraints contained in property restrictions. The latter are class expressions and are only enforced on the property when applied to that class.

- isTransitive: Boolean [0..1] - A transitive property is a property for which the triples (x, P, y) and (y, P, z) imply (x, P, z).
- isSymmetric: Boolean [0..1] - A symmetric property is a property for which the triple (x, P, y) implies (y, P, x).
- isAsymmetric: Boolean [0..1] – An asymmetric property is a property for which the triple (x, P, y) implies NOT(y, P, x).
- isReflexive: Boolean [0..1] – A reflexive property is a property for which the triple (x, P, x) is always true.
- isIrreflexive: Boolean [0..1] – An irreflexive property is a property for which the triple (x, P, x) is always false.

Associations

- inverseProperty: OWLObjectProperty [0..*] in association InverseProperty.
- OWLinverseOf: OWLObjectProperty [0..*] in association InverseProperty.
- Specialize Class *Property.*

Constraints

[1] The range of an OWLObjectProperty is restricted to the set of individuals, i.e., a member of the class extension of OWLClass.

context OWLObjectPropertyRangeIsOWLClass inv:

(self.RDFSrange.oclIsKindOf (OWLClass))

[2] If isSymmetric = true or isTransitive = true, then the domain must be the same as the range.

[3] if isReflexive = true, then the domain must be the same as the range.

[4] isSymmetric and isAsymmetric may not both be true.

[5] isReflexive and isIrreflexive may not both be true.

Semantics

See the formal [OWL S&AS] for additional semantics.

11.4.4 OWLOntologyProperty

Description

A document describing an ontology typically contains information about the ontology itself. An ontology is a resource, so it may be described using properties from the OWL and other namespaces. An ontology property is essentially an annotation property that allows us to say things about the current and other ontologies, such as indicating that a particular ontology is a prior version of the current ontology.

Several ontology properties are predefined by OWL, namely:

- `owl:imports`
- `owl:priorVersion`
- `owl:backwardCompatibleWith`
- `owl:incompatibleWith`

Attributes

None.

Associations

- Specialize Class RDFProperty

Constraints

[1] Instances of `owl:OntologyProperty` must have the class `owl:Ontology` as their domain and range.

context OWLOntologyPropertyDomainRangeIsOWLOntology inv:

(self.RDFSdomain.oclIsKindOf(OWLOntology) and

self.RDFSrange.oclIsKindOf(OWLOntology))

Semantics

No additional semantics.

11.4.5 Property

Description

Property is an abstract class that simplifies representation of property equivalence and deprecation, simplifies constraints for OWL DL and OWL Full, and facilitates mappings with other metamodels.

Attributes

- isDeprecated: Boolean [0..1] - indicates that use of this property is deprecated.
- isFunctional: Boolean [0..1] – A functional property is a property that can have only one (unique) value y for each instance x, i.e., if the triples (x, P, y1) and (x, P, y2) are asserted then y1 must be the same as y2 (they cannot be distinct).

Associations

- equivalentProperty: Property [0..*] in association EquivalentProperty - links a property to zero or more properties that it is considered equivalent to.
- OWLequivalentProperty: Property [0..*] in association EquivalentProperty - links a property to zero or more properties that it is considered equivalent to.
- Specialize Class RDFProperty.

Constraints

No additional constraints. Note that in OWL as in RDF, properties are required to have IRIs, which in this case are inherited from RDFProperty.

Semantics

No additional semantics.

11.5 OWLBase Package - Individuals

Individuals in OWL are defined through individual axioms (also called “facts”). Two types of facts are available for use in ontology development:

- Facts about class membership and property values of individuals.
- Facts about individual identity.

Many languages have a so-called “unique names” assumption: different names refer to different things in the world. On the web, such an assumption is not possible. For example, the same person could be referred to in many different ways (i.e., with different URI references). For this reason OWL does not make this assumption. Unless an explicit statement is being made that two URI references refer to the same or to different individuals, OWL tools should in principle assume either situation is possible. Figure 11.6 depicts the set of constructs available to state facts about individual identity in OWL.

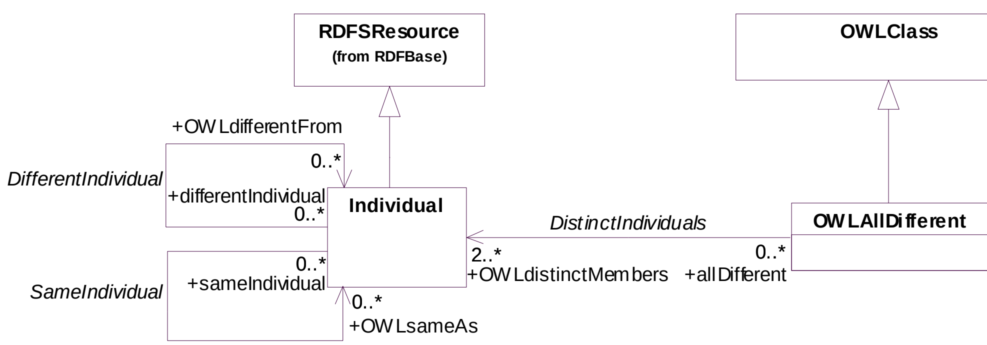

Figure 11.6 - The OWL Individuals Diagram

11.5.1 OWLAllDifferent

Description

For ontologies in which the unique-names assumption holds, the use of `owl:differentFrom` is likely to lead to a large number of statements, as all individuals have to be declared pairwise disjoint. For such situations OWL provides a special idiom in the form of the construct `owl:AllDifferent`. `owl:AllDifferent` is a special built-in OWL class, for which the property `owl:distinctMembers` is defined, which links an instance of `owl:AllDifferent` to a list of individuals. The intended meaning of such a statement is that all individuals in the list are all different from each other.

Note that instances of `owl:AllDifferent` are blank nodes.

Attributes

No additional attributes.

Associations

- OWLdistinctMembers: Individual [2..*] in association DistinctIndividuals - specifies that a particular set of individuals are distinct from one another.

- Specialize Class OWLClass.

Constraints

[1] All members of a particular instance of the class `owl:AllDifferent` are pairwise disjoint from each other.

Semantics

No additional semantics.

11.6 OWLBase Package - Datatypes and Data Ranges

OWL makes use of the RDF datatyping scheme, which provides a mechanism for referring to XML Schema datatypes [XML Schema Datatypes]. Note that only a subset of the XML Schema datatypes are recommended for use in RDF and OWL, as discussed in Clause 10.

In place of the RDF datatypes, OWL provides several constructs for defining a range of data values, (1) it identifies a specific datatype class from the RDF datatypes (e.g., xsd:integer) that a value in the data range must reflect; (2) a restriction on an RDF datatype, which makes use of XML Schema and RDF facets; (3) an enumerated datatype, which is an enumerated list of literals; (4) a logical expression applied to other datatypes.

The Datatypes diagram is provided in Figure 11.7.

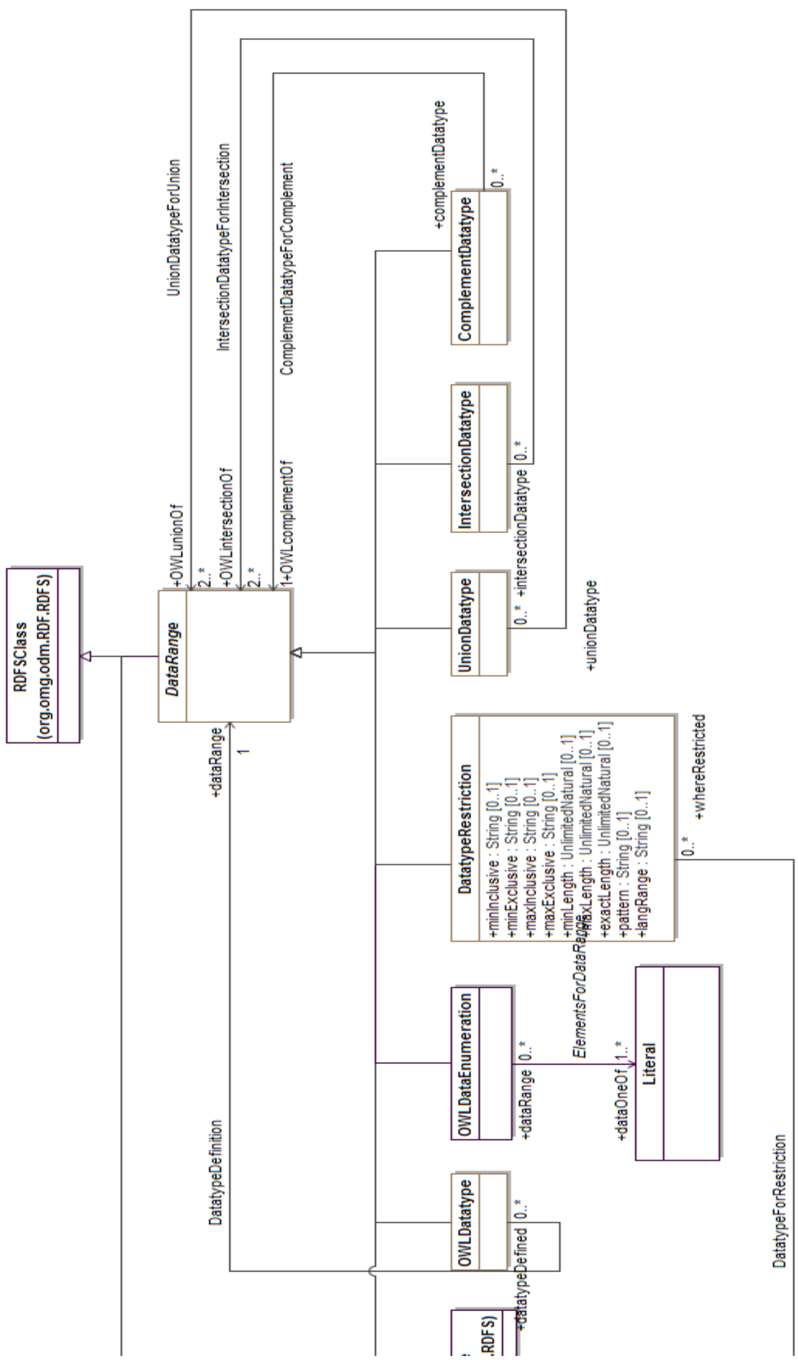

Figure 11.7 - The OWL Datatypes Diagram

11.6.1 DataRange

Description

DataRange is an abstract class used for structuring the datatype expressions. It represents an anonymous type, but can also be used to define an OWLDatatype which is a named resource. Note, however, that these datatypes cannot themselves have DatatypeRestrictions nor can they be used to classify Literals.

Attributes

None.

Associations

- Specialize Class RDFSClass.

Constraints

[1] Except for OWLDatatype, ClassExpressions must be anonymous: specifically the property IRIForResource must be empty.

Semantics

No additional semantics.

11.6.2 OWLDatatype

Description

This represents a named Datatype which is a resource and is defined using a DataRange.

Attributes

None.

Associations

- dataRange:DataRange[1] - The DataRange defining this datatype.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.6.3 OWLDataEnumeration

Description

This represents a DataRange which is defined through listing its complete set of values.

Attributes

None.

Associations

- dataOneOf: Literal[1..*] - links a data range to the enumerated list of literals that comprise its values.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.6.4 11.6.4 DatatypeRestriction

Description

This represents a DataRange which is a restriction of another datatype (which must be a RDFSDatatype).

Attributes

- minInclusive: String [0..1] – minimum inclusive value (on numeric datatypes).
- minExclusive: String [0..1] – minimum exclusive value (on numeric datatypes).
- maxInclusive: String [0..1] – maximum inclusive value (on numeric datatypes).
- maxExclusive: String [0..1] – maximum exclusive value (on numeric datatypes).
- minLength: UnlimitedNatural [0..1] – minimum string length (on string datatypes).
- maxLength: UnlimitedNatural [0..1] – maximum string length (on string datatypes).
- exactLength: UnlimitedNatural [0..1] – exact string length (on string datatypes).
- pattern: String [0..1] – pattern expressions constraining string datatypes. The syntax and usage is defined in 4.3.4 of [XML Schema Datatypes].
- langRange: String [0..1] – restricts the natural language permitted for string datatypes. This is a Basic Language Range as specified in 4.3 of the OWL 2 specification.

Associations

- datatypeRestricted:RDFSDataType[1] - The source datatype that the restriction is applied to.

Constraints

[1] At most one of the following properties may have a value: minInclusive, minExclusive.

[2] At most one of the following properties may have a value: maxInclusive, maxExclusive.

[3] At most one of the following properties may have a value: minLength, exactLength.

[4] At most one of the following properties may have a value: maxLength, exactLength.

[5] The properties used must be compatible with the type referenced by datatypeRestricted as per the description for each property.

Semantics

No additional semantics.

11.6.5 UnionDatatype

Description

An owl:unionOf statement describes a datatype for which the extension contains exactly those values that belong to the extension of at least one of the data ranges that are the object of the statement. It is analogous to logical disjunction.

Attributes

None.

Associations

- OWLunionOf:DataRange[2..*] - The DataRanges unioned to form this datatype.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.6.6 IntersectionDatatype

Description

An owl:intersectionOf statement describes a datatype for which the extension contains exactly those values that belong to the extension of each of the data ranges that are the object of the statement. It is analogous to logical conjunction.

Attributes

None.

Associations

- OWLintersectionOf:DataRange[2..*] - The DataRanges intersectioned to form this datatype.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.6.7 ComplementDatatype

Description

An owl:complementOf statement describes a datatype for which the extension contains exactly those values that do not belong to the extension of the data range that is the object of the statement. It is analogous to logical negation: the datatype extension consists of those values that are NOT members of the extension of the complement data range.

Attributes

None.

Associations

- OWLcomplementOf:DataRange[1] - The DataRange complemented to define this datatype.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.7 OWLBase Package - OWL Universe

One of the more difficult OWL concepts for UML users to grasp is that an ontology can be a very large graph spanning multiple documents, with additional definitions that are distributed over even more documents "somewhere out in the wild, wild Web." Yet, we want to be able to represent such notions using UML tools, and to map other kinds of models to this metamodel as a starting point for ontology development. While it is true that one can determine the contents of a particular ontology by "walking the graph" to determine the set of statements it contains, this approach can be awkward from a mapping perspective in particular.

Additionally, we want to be able to define the set of constraints that will allow us to differentiate between an ontology that conforms to OWL DL and one that is OWL Full compliant. In Figure 11.8, we provide the notion of an abstract *OWLUniverse* class, which facilitates ontology traversal for mapping purposes as well as utility in defining constraints for distinguishing these two dialects of OWL.

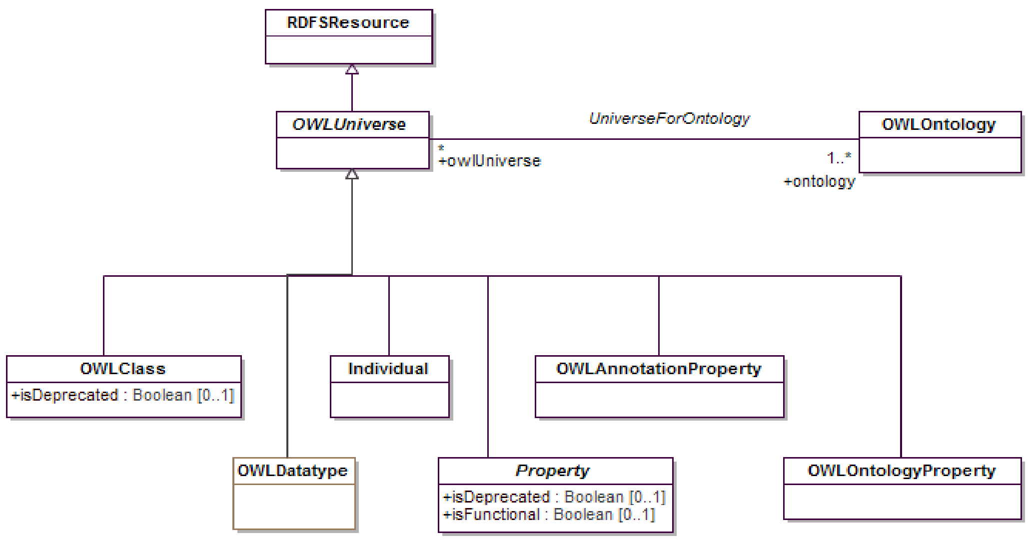

Figure 11.8 - The OWL Universe Diagram

11.7.1 OWLUniverse

Description

This class is intended to simplify packaging / mapping requirements for cases where the ability to determine the set of classes, individuals, and properties that together comprise a particular OWL ontology is required.

Attributes

None.

Associations

- ontology: OWLOntology [1..*] in association UniverseForOntology - specifies one or more OWLOntology that members of this universe are associated with/describe.
- Specialize Class RDFSResource.

Constraints

No additional constraints.

Semantics

No additional semantics.

11.7.2 OWLOntology (Augmented Definition)

Associations

- owlUniverse: OWLUniverse [*] in association UniverseForOntology - specifies an OWL universe(s) for this ontology.

11.8 OWLDL Package - Constraints for OWL DL Conformance

The RDF-Compatible Model-Theoretic Semantics for OWL [OWL S&AS] defines the OWL DL universe as being the subset of the RDF universe that contains the set of OWL classes, individuals, and properties, as shown in Figure 11.8. In that context, the set of classes, datatypes, datatype properties, object properties, annotation properties, ontology properties, individuals, data values, and other built-in vocabulary are pairwise disjoint.

```
context OWLUniverse inv OWLDLDisjointPartition:
-- subclasses exhaust OWLUniverse
(self.oclIsKindOf(OWLClass) or self.oclIsKindOf(Individual) or self.oclIsKindOf(Property) or
 self.oclIsKindOf(OWLDataRange) or self.oclIsKindOf(OWLAnnotationProperty) or
 self.oclIsKindOf(OWLOntologyProperty)) and

-- subclasses are pairwise disjoint
not (self.oclIsKindOf(OWLClass) and self.oclIsKindOf(Individual)) and
not (self.oclIsKindOf(OWLClass) and self.oclIsKindOf(Property)) and
not (self.oclIsKindOf(OWLClass) and self.oclIsKindOf(OWLDataRange)) and
not (self.oclIsKindOf(OWLClass) and self.oclIsKindOf(OWLAnnotationProperty)) and
not (self.oclIsKindOf(OWLClass) and self.oclIsKindOf(OWLOntologyProperty)) and

not (self.oclIsKindOf(Individual) and self.oclIsKindOf(Property)) and
not (self.oclIsKindOf(Individual) and self.oclIsKindOf(OWLDataRange)) and
not (self.oclIsKindOf(Individual) and self.oclIsKindOf(OWLAnnotationProperty)) and
not (self.oclIsKindOf(Individual) and self.oclIsKindOf(OWLOntologyProperty)) and

not (self.oclIsKindOf(Property) and self.oclIsKindOf(OWLDataRange)) and
not (self.oclIsKindOf(Property) and self.oclIsKindOf(OWLAnnotationProperty)) and
not (self.oclIsKindOf(Property) and self.oclIsKindOf(OWLOntologyProperty)) and

not (self.oclIsKindOf(OWLDataRange) and self.oclIsKindOf(OWLAnnotationProperty)) and
not (self.oclIsKindOf(OWLDataRange) and self.oclIsKindOf(OWLOntologyProperty)) and

not (self.oclIsKindOf(OWLAnnotationProperty) and self.oclIsKindOf(OWLOntologyProperty))
```

Several additional constraints must be applied in general in OWL DL:

- All classes and properties must be explicitly typed.

- Axioms about individual equality and difference must be about named individuals only (a consequence of category separation).
- There are severe limitations on the use of RDF vocabulary in OWL DL [OWL S&AS].
- OWL, RDF, and RDFS vocabularies cannot be modified by statements in OWL DL.

11.8.1 Classes in OWL DL

In OWL DL, OWLClass is defined as a proper subset of RDFSClass.

11.8.2 OWL DL Restrictions

Additional restrictions apply to OWL DL value restrictions, as follows.

11.8.2.1 AllValuesFromRestriction

Constraints

[1] If the property linked to the AllValuesFromRestriction is an OWLDatatypeProperty, then the restriction is linked to exactly 1 OWLDataRange and 0 OWLClass.

[2] If the property linked to the AllValuesFromRestriction is an OWLObjectProperty, the restriction is linked to exactly 1 OWLCLass and 0 OWLDataRange.

11.8.2.2 HasValueRestriction

Constraints

[1] If the property linked to the HasValueRestriction is an OWLDatatypeProperty, then the restriction is linked to exactly 1 RDFSLiteral and 0 Individual.

[2] If the property linked to the HasValueRestriction is an OWLObjectProperty, then the restriction is linked to exactly 1 Individual and 0 RDFSLiteral.

11.8.2.3 SomeValuesFromRestriction

Constraints

[1] If the property linked to the SomeValuesFromRestriction is an OWLDatatypeProperty, then the restriction is linked to exactly 1 DataRange and 0 Class.

[2] If the property linked to the SomeValuesFromRestriction is an OWLObjectProperty, then the restriction is linked to exactly 1 Class and 0 DataRange.

11.8.3 OWL DL Property Constraints

Pairwise separation between datatype, object, annotation, and ontology properties must be strictly maintained in OWL DL.

context RDFProperty inv OWLDLDisjointPartition:

-- subclasses exhaust RDFProperty

(self.oclIsKindOf(OWLAnnotationProperty) or self.oclIsKindOf(OWLDatatypeProperty) or

self.oclIsKindOf(OWLObjectProperty) or self.oclIsKindOf(OWLOntologyProperty)) and

-- subclasses are pairwise disjoint

not (self.oclIsKindOf(OWLAnnotationProperty) and self.oclIsKindOf(OWLDatatypeProperty)) and

not (self.oclIsKindOf(OWLAnnotationProperty) and self.oclIsKindOf(OWLObjectProperty)) and

not (self.oclIsKindOf(OWLAnnotationProperty) and self.oclIsKindOf(OWLOntologyProperty)) and

not (self.oclIsKindOf(OWLDatatypeProperty) and self.oclIsKindOf(OWLObjectProperty)) and

not (self.oclIsKindOf(OWLDatatypeProperty) and self.oclIsKindOf(OWLOntologyProperty)) and

not (self.oclIsKindOf(OWLObjectProperty) and self.oclIsKindOf(OWLOntologyProperty))

11.8.3.1 OWLAnnotationProperty

Description

The following additional restrictions on the use of annotation properties apply:

- Annotation properties must have an explicit typing triple of the form:

  AnnotationPropertyID `rdf:type owl:AnnotationProperty`.
- Annotation properties must not be used in property axioms. Thus, in OWL DL one cannot define subproperties or domain/range constraints for annotation properties.
- The object of an annotation property must be either a data literal, a URI reference, or an individual.

Constraints

[1] The association RDFSrange cannot be used with an OWLAnnotationProperty.

[2] The association RDFSdomain cannot be used with an OWLAnnotationProperty.

[3] Hierarchies of annotation properties are disallowed: the association RDFSsubPropertyOf cannot be used with an OWLAnnotationProperty.

11.8.3.2 OWLDatatypeProperty

Description

The following additional restrictions apply to the use of datatype properties in OWL DL:

- The range of a datatype property must be either a data range or literal.
- Property equivalence only holds among datatype properties.
- Property characteristics including inverse, inverse functional, symmetric, and transitive cannot be applied to datatype properties.

Constraints

[1] If the association OWLequivalentProperty is defined on an OWLDatatypeProperty, the Property on the other end of that equivalence must also be of type OWLDatatypeProperty.

[2] If the association RDFSsubPropertyOf is defined on an OWLDatatypeProperty, the RDFProperty on the other end of the generalization must also be of type OWLDatatypeProperty.

[3] The range of OWLDatatypeProperty (association RDFSrange on superclass RDFProperty) is limited to OWLDataRange.

11.8.3.3 OWLObjectProperty

Description

The following additional restrictions apply to the use of object properties in OWL DL:

- The sets of object properties, datatype properties, annotation properties, and ontology properties must be mutually disjoint.
- Cardinality constraints (local or global) cannot be applied to transitive properties, to their inverses, or to any of their super properties.
- Inverse functional, symmetric and transitive properties must be object properties.
- A property that is the subject or object of inverseOf must be an object property.

Constraints

[1] If the association OWLEquivalentProperty is defined on an OWLObjectProperty, the Property on the other end of the equivalence must also be of type OWLObjectProperty.

[2] If the association RDFSsubPropertyOf is defined on an OWLObjectProperty, the RDFProperty on the other end of the generalization must also be of type OWLObjectProperty.

[3] The range of OWLObjectProperty (association RDFSrange on superclass RDFProperty) is limited to OWLClass.

11.8.3.4 OWLOntologyProperty

Description

Similar restrictions to those on the use of annotation properties apply to ontology properties:

- Ontology properties must have an explicit typing triple of the form:

  OntologyPropertyID `rdf:type owl:OntologyProperty .`
- Ontology properties must not be used in property axioms. Thus, in OWL DL one cannot define subproperties or domain/range constraints for ontology properties.
- The subject and object of an ontology property must be an ontology.

Constraints

[1] The association RDFSrange cannot be used with an OWLOntologyProperty.

[2] The association RDFSdomain cannot be used with an OWLOntologyProperty.

[3] Hierarchies of ontology properties are disallowed: RDFSsubPropertyOf cannot be used with an OWLOntologyProperty.

11.8.3.5 TransitiveProperty

Constraints

[1] No local or global cardinality constraints can be declared on a transitive property or on any of its super properties, nor on the inverse of the property or any of the inverse's superProperty.

11.9 OWLFull Package - Constraints For OWL Full Conformance

There are several key distinctions between OWL DL and OWL Full. These include:

- Unconstrained use of the RDF vocabulary.
- Lack of disjointedness between classes and individuals which allows for variation in the role that a particular concept plays given different perspectives within the same or a group of ontologies.
- Equivalence between `rdfs:class` and `owl:class` in OWL Full (whereas in OWL DL, OWLClass is a proper subset of RDFSClass meaning that not all RDF classes are OWL DL classes).
- Data values are not disjoint from individuals in OWL Full, thus the distinction between datatype properties and object properties is relaxed: (1) `owl:Thing` is equivalent to `rdfs:Resource`, (2) `owl:ObjectProperty` is equivalent to `rdf:Property`, (3) and thus effectively, `owl:DatatypeProperty` is a subclass of `owl:ObjectProperty`.

12 The Common Logic Metamodel

12.1 Overview

Common Logic (CL) is a first-order logical language intended for information exchange and transmission over an open network [ISO 24707]. It allows for a variety of different syntactic forms, called dialects, all expressible within a common XML-based syntax and all sharing a single semantics. The language has declarative semantics, which means that it is possible to understand the meaning of expressions written in CL without requiring an interpreter to manipulate those expressions. CL is logically comprehensive – at its most general, it provides for the expression of arbitrary logical expressions. CL has a purely first-order semantics, and satisfies all the usual semantic criteria for a first-order language, such as compactness and the downward Skolem-Löwenheim property.

Motivation for its consideration as an integral component of the Ontology Definition Metamodel (ODM) includes:

- The potential need by ontologists using the ODM to be able to represent constraints and rules with expressiveness beyond that supported by description logics (e.g., for composition of semantic web services), as highlighted in Clause 7, Usage Scenarios and Goals.
- The availability of normative mappings from CL to syntactic forms for several commonly used knowledge representation standards, defined in [ISO 24707], including the Knowledge Interchange Format [KIF] and Conceptual Graphs [CGS].
- The availability of a normative XML-based surface syntax for CL, called XCL (also defined in [ISO 24707], which dramatically increases its potential for use in web-based applications.
- The availability of a direct mapping from the Web Ontology Language (OWL) [OWL S&AS] to CL, such that CL reasoners can leverage both the ontologies expressed in OWL and constraints written in CL to solve a wider range of problems than can be addressed by OWL alone (see Clause 18, Mapping RDFS and OWL to CL).

In general, first order logic provides the basis for most commonly used knowledge representation languages, including relational databases; more application domains have been formalized using first order logic than any other formalism – its meta-mathematical properties are thoroughly understood. CL in particular provides a modern form of first order logic that takes advantage of recent insights in some of these application areas including the semantic web.

First order logic can also provide the formal grounding for business semantics. Although work on the OMG's Business Semantics For Business Rules (BSBR) RFP was initially done in parallel with the ODM, there has been significant effort to leverage CL as the first order logic basis for the Semantics of Business Vocabulary and Business Rules (SBVR) specification. Common Logic (and thus ODM) now supports irregular sentences, a recent addition to the abstract syntax of CL required for the SBVR modality representations, for example. Subsequent versions of both specifications will be amended to accommodate additional interoperability requirements to the extent possible.

12.1.1 Design Considerations

The CL Metamodel is defined per [ISO 24707], and was developed with the help of the CL language authors to be a comprehensive and accurate representation of the abstract syntax of CL. As indicated in Clause 8, Design Rationale, a decision was made not to depend on the OCL 2.0 Metamodel specifically because such a dependency would introduce unnecessary complexity and semantics that may be inconsistent with the simplicity, efficiency, and formal semantics of CL. Inconsistencies in the semantics can have unintended consequences for downstream reasoning, limiting the utility of an ODM-based application that leverages the CL metamodel. A mapping between CL and OCL may be considered in an add-on to the ODM (through OMG's RFC process). Such a mapping would require validation through the use of CL and OCL-based reasoning engines, which will likely not be available prior to finalization of this specification.

To date, although a number of proposals have been put forth to the W3C for a rule language for OWL, there is no formal recommendation available from the W3C today. Such a standard may be considered for integration with, or as an additional candidate for mapping to, the CL metamodel through a subsequent RFP/RFC.

The complete syntax and formal semantics for CL are documented in [ISO 24707] and are considered essential to understanding the expressions that might be imported, managed, manipulated, or generated by any ODM/CL-compliant tool.

12.1.2 Modeling Notes

All of the OCL constraints documented below have been validated using OCL tools.

12.2 The Phrases Diagram

Phrases provide mechanisms for grouping and scoping the elements that constitute an ontology (or set of constraints associated with an OWL ontology), authored in Common Logic or any of its syntactic variants. An overview of the top-level elements of the CL metamodel is provided in Figure 12.1.

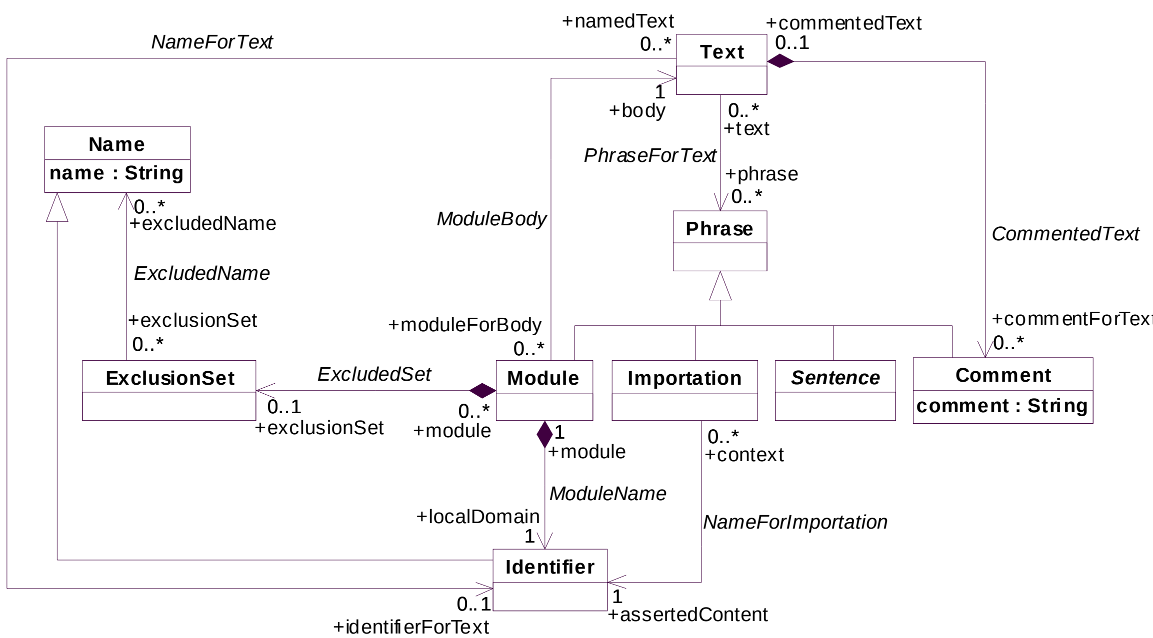

Figure 12.1 - Phrases

12.2.1 Comment

Description

A Comment is a “piece of data” that provides the facility for commenting on a particular phrase or set of sentences. Common Logic places no restrictions on the nature of comments.

Attributes

- comment: String [1] – the character string that is the comment on the phrase.

Associations

- commentedText: Text [0..1] in association CommentedText – the text to which the comment applies.
- Specialize Class Phrase.

Constraints

None.

Semantics

None.

12.2.2 ExclusionSet

Description

A module may optionally have an exclusion list of names whose denotations might be excluded from the local domain of discourse. These are called non-discourse names.

Attributes

None

Associations

- excludedName: Name [0..*] in association ExcludedName – the names that are members of the ExclusionSet.
- module: Module [0..*] in association ExcludedSet – the module(s) that excludes this set of names.

Constraints

None.

Semantics

An ExclusionSet essentially represents some set of non-discourse names as they relate to a particular domain of discourse. See [ISO 24707] for additional detail.

12.2.3 Identifier

Description

An identifier is a name explicitly used to identify a module or piece of common logic text.

Attributes

None.

Associations

- context: Importation [0..*] in association NameForImportation – links an identifier to an importation that references it.

- module: Module [1] in association ModuleName – links an identifier to the module it names.
- namedText: Text [0..1] in association NameForText – links an identifier to the text it names.
- Specialize Class Name.

Constraints

None.

Semantics

Names used to name texts on a network are understood to be *rigid* and to be *global* in scope, so that the name can be used to identify the thing named – in this case, the Common Logic text or module – across the entire communication network. (See [RFC2396] for further discussion.) A name which is globally attached to its denotation in this way is an *identifier*, and is typically associated with a system of conventions and protocols which govern the use of such names to identify, locate, and transmit pieces of information across the network on which the dialect is used. While the details of such conventions are beyond the scope of this specification, we can summarize their effect by saying that the act of publishing a named Common Logic text (or module) is intended to establish the name as a rigid identifier of the text, and Common Logic acknowledges this by requiring that *all* interpretations shall conform to such conventions when they apply to the network situation in which the publication takes place.

Note that in the case of an importation, the name serves to identify the module, which is accomplished through a double interpretation in the semantics. The 'import' condition is that (import x) is true in I just when I(I(x)) is true. In other words, interpreting an identifier gets what it denotes. If the name happens to be a CL ontology (I(x) is an ontology), then interpreting it again I(I(x)) returns a truth-value; thus, (import x) says that x is an ontology which *this* ontology (the one doing the importing) asserts to be true.

12.2.4 Importation

Description

An importation contains a name. The intention is that the name is an identifier of a piece of Common Logic content represented externally to the text, and the importation re-asserts that content in the text.

Attributes

None.

Associations

- assertedContent: Identifier [1] in association NameForImportation – the name of the module to be imported; the name argument of an importation will usually be a URI.
- Specialize Class Phrase.

Constraints

None.

Semantics

An import construction requires that we assume the existence of a global module-naming convention, and that module names refer to entities in formal interpretations. Common Logic uses the same semantic web conventions used in RDF and OWL, based on W3C recommendation for representing namespaces in XML (see "[XMLNS]" on page 4). The meaning of an importation phrase is that the name it contains shall be understood to identify some Common Logic content, and the importation is true just when that content is true. Thus, an importation amounts to a virtual 'copying' of some Common Logic content from one 'place' to another. This idea of 'place' and 'copying' can be understood only in the context of deploying logical content on a communication network. A *communication network*, or simply a *network*, is a system of agents which can store, publish, or process common logic text, and can transmit common logic text to one another by means of information transfer protocols associated with the network.

12.2.5 Module

Description

A module consists of a name, an optional set of names called an *exclusion set*, and a text called the *body text*. The module name indicates the "local universe of discourse" in which the text is understood; the exclusion list indicates any names in the text which are excluded from the local domain (i.e., variables whose scope is external to the local domain).

Attributes

None.

Associations

- body: Text [1] in association ModuleBody – the body, or set of phrases, that are contained in the module.
- exclusionSet: ExclusionSet [0..1] in association ExcludedSet – the optional set of names, or exclusion list, associated with a given module.
- localDomain: Identifier [1] in association ModuleName – the logical name associated with a module (for most applications, particularly those that are web based, module names must be unique).
- Specialize Class Phrase.

Constraints

In cases where CL is used to define ontologies for the Web, module names take the form of Uniform Resource Identifiers [RDF Syntax] or URI references, and are global (thus must be unique).

Semantics

A module provides the scoping mechanism for a CL ontology, corresponding to an RDF graph [RDF Primer] or document, or to an OWL ontology. The name of a module should be the name of the corresponding RDF document in cases where CL constraints are associated with an RDFS/OWL ontology, and has the same URI or URI reference (i.e., that of the RDFS/OWL ontology).

The CL syntax provides for modules to state an intended domain of discourse, to relate modules explicitly to other domains of discourse, and to express intended restrictions on the syntactic roles of symbols. This feature is critical to component-based ontology (or micro-theory) construction, and therefore relevant to any MDA-based authoring environment.

12.2.6 Name

Description

A *name* is any lexical token, or character string, which is understood to refer to something in the universe. Part of the design philosophy of CL is to avoid syntactic distinctions between name types, allowing ontologies freedom to use names without requiring mechanisms for syntactic alignment. Names are primitive referring elements in CL, and refer to elements of a particular ontology, such as module names, role names, relations, or numbers.

Dialects intended for use on the Web should allow Universal Resource Identifiers and URI references [RDF Syntax] to be used as names. Common Logic dialects should define names in terms of Unicode [ISO 10646] conventions.

Attributes

- name: String [1] – the character string symbolizing the name.

Associations

- exclusionSet: ExclusionSet [0..*] in association ExcludedName – the optional exclusion list referring to the name.
- binding: Binding [1] in association BoundName – associates a name (variable) with the related binding (i.e., the name becomes a binding) in quantified sentences.
- Specialize Class Term.

Constraints

[1] The lexical syntax for several CL dialects identifies a number of rules for specifying valid names that cannot be expressed in OCL, and are thus delegated to CL parsers (such as identification of special characters that cannot be embedded in names, the requirement for conformance to Unicode conventions, additional constraints on logical names that are URIs or URI references, and so forth).

[2] Names and sequence markers are disjoint syntax categories, and each is disjoint from all other syntax categories.

Semantics

The only undefined terms in the CL abstract syntax are *name* and *sequence marker*. The only required constraint on these is that they must be exclusive. Common Logic does not require names to be distinguished from variables, nor does it require names to be partitioned into distinct classes such as relation, function or individual names, or impose sortal restrictions on names. Particular Common Logic dialects may make these or other distinctions between subclasses of names, and impose extra restrictions on the occurrence of types of names or terms in expressions - for example, by requiring that bindings be written with a special variable prefix, as in KIF, or with a particular style, as in Prolog; or by requiring that operators be in a distinguished category of relation names, as in conventional first-order syntax.

A dialect may impose particular semantic conditions on some categories of names, and apply syntactic constraints to limit where such names occur in expressions. For example, the core syntax treats numbers as having a fixed denotation, and prohibits their use as identifiers. A dialect may require some names to be non-discourse names. This requirement may be imposed by, for example, partitioning the vocabulary, or by requiring names which occur in certain syntactic positions to be non-denoting. A dialect with non-discourse names is called segregated.

12.2.7 Phrase

Description

A phrase is a syntactic unit of text. A phrase is either a comment, or a module, or a sentence, or an importation, or a phrase with an attached comment.

Attributes

None.

Associations

- text: Text [0..*] in association PhraseForText – the text(s) in which the phrase occurs.

Constraints

[1] Module, Importation, Sentence, and Comment are specializations of Phrase and form a disjoint partition, as follows:

```
context Phrase inv XOR:
   (self.oclIsKindOf(Module) xor self.oclIsKindOf(Importation) xor
    self.oclIsKindOf(Sentence) xor self.oclIsKindOf(Comment))
```

Semantics

No additional semantics.

12.2.8 Sentence

Description

CL provides facilities for expressing several kinds of sentences, including atomic sentences as well as compound sentences built up from atomic sentences or terms with a set of logical constructors. A sentence is either a quantified sentence or a Boolean sentence or an atom, or a sentence with an attached comment, or an irregular sentence. CL sentences can be classified (or used) as phrases, as stated above and as shown in Figure 12.1.

The convention used in CL for expressing sentences differs from the approach taken in the informative DL metamodel. In the DL case, constructors are uniquely defined, whereas in CL the constructors are an integral part of the sentence, named for the kind of construction used in the sentence.

Attributes

None.

Associations

- biconditional: Biconditional [0..1] in association LvalueForBiconditional – associates a sentence as the “lvalue” (or left-hand side) of a Biconditional or biconditional relation.
- biconditional: Biconditional [0..1] in association RvalueForBiconditional – associates a sentence as the “rvalue” (or right-hand side) of a Biconditional or biconditional relation.
- comment: CommentedSentence [0..1] in association CommentForSentence – provides the facility for commenting any given CL sentence.

- conjunction: Conjunction [0..1] in association Conjunction – associates a sentence to its conjuncts in a conjunction.
- disjunction: Disjunction [0..1] in association Disjunction – associates a sentence to its disjuncts in a disjunction.
- implication: Implication [0..1] in association AntecedentForImplication – associates a sentence as the antecedent of an implication.
- implication: Implication [0..1] in association ConsequentForImplication – associates a sentence as the consequent of an implication.
- negation: Negation [0..1] in association NegatedSentence – associates a sentence with a negation.
- quantification: QuantifiedSentence [0..1] in association QuantificationForSentence – associates a sentence (body) with a quantifier and optional bindings.
- Specialize Class Phrase.

Constraints

The partition formed by the subclasses of Sentence is disjoint:

```
context Sentence inv DisjointPartition:
    (self.oclIsKindOf(Conjunction) xor self.oclIsKindOf(Disjunction) xor
     self.oclIsKindOf(Negation) xor self.oclIsKindOf(Implication) xor
     self.oclIsKindOf(Biconditional) xor self.oclIsKindOf(UniversalQuantification) xor
     self.oclIsKindOf(ExistentialQuantification) xor self.oclIsKindOf(Atom) xor
     self.oclIsKindOf(CommentedSentence))
```

Semantics

No additional semantics.

12.2.9 Text

Description

Text is a collection of phrases (set, sequence, or bag, optionally specified by a CL dialect), optionally identified by a name. A module is a named piece of text with an optional exclusion set containing names considered to be outside the domain of discourse for the module.

Attributes

None.

Associations

- commentForText: Comment [0..*] in association CommentedText – optional comment(s) associated with the text.
- identifierForText: Identifier [0..1] in association NameForText – links a text with an identifier in a named text.
- phrase: Phrase [0..*] in association PhraseForText – the phrase(s) or sentence(s) that comprise the text.
- moduleForBody: Module [0..*] in association ModuleBody – the module(s) owning the text.

Constraints

None.

Semantics

The semantics of Common Logic is defined in terms of a *satisfaction* relation between CL text and structures called *interpretations*. All dialects **must** apply these semantic conditions to all common logic expressions, that is, to any of the forms given in the abstract syntax. They **may** in addition apply further semantic conditions to subclasses of common logic expressions, or to other expressions.

A vocabulary is a set of names and sequence markers. The vocabulary of a Common Logic text is the set of names and sequence markers which occur in the text. In a segregated dialect, vocabularies are partitioned into denoting names and non-discourse names.

An interpretation I of a vocabulary V is a set U_I, the *universe*, with a distinguished non-empty subset D_I, the domain of discourse, or simply *domain*, and four mappings: rel_I from U_I to subsets of D_I*, fun_I from U_I to FunctionalTerms D_I*->D_I, (which we will also consider to be the set D_I* x D_I), int_I from names in V to U_I, and seq_I from sequence markers in V to D_I*. If the dialect is segregated, then $int_I(x)$ is in D_I if and only if x is a denoting name. If the dialect recognizes irregular sentences, then they are treated as names of propositions, and int_I also includes a mapping from the irregular sentences of a text to the truth values {true, false}.

Intuitively, D_I is the domain of discourse containing all the individual things the interpretation is 'about' and over which the quantifiers range. U_I is a potentially larger set of things which might also contain entities which are not in the universe of discourse. All names are interpreted in the same way, whether or not they are understood to denote something in the domain of discourse; this is why there is only a single interpretation mapping applying to all names regardless of their syntactic role. In particular, $rel_I(x)$ is in D_I* even when x is not in D. When considering only segregated dialects, the universe outside the domain may be considered to contain names and can be ignored; when considering only unsegregated dialects, the distinction between universe and domain is unnecessary. The distinction is required in order to give a uniform treatment of all dialects. Irregular sentences are treated as though they were arbitrary propositional variables.

A discussion of the semantics regarding text interpretation is given in "[ISO 24707]" on page 4, including distinctions in quantifier scope, features enabling structured relationships among modules, closed-world and unique naming issues, and so forth.

12.3 The Terms Diagram

The Terms Diagram, shown in Figure 12.2 provides additional insight into the core syntactic elements of Common Logic. These include names, commented terms, and term sequences (FunctionalTerms).

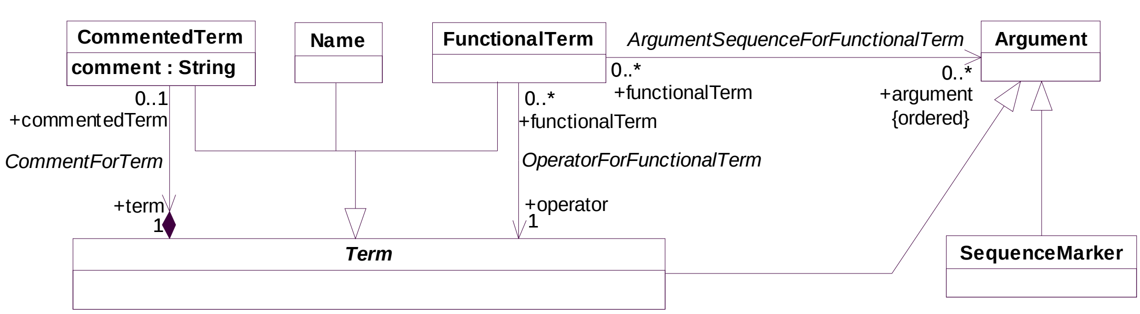

Figure 12.2 - Valid Terms in CL

12.3.1 Argument

Description

An argument sequence is a finite ordered sequence of bindings, which are terms or sequence markers. A sequence may be empty or may consist of a single sequence marker.

Attributes

None.

Associations

- atomicSentence: AtomicSentence [0..*] in association ArgumentSequenceForAtomicSentence – links an argument sequence to an atomic sentence.
- functionalTerm: FunctionalTerm [0..*] in association ArgumentSequenceForFunctionalTerm – links an argument sequence to a functional term.

Constraints

[1] The terms in an argument sequence are ordered.

Semantics

An argument sequence represents a sequence of argument terms. A sequence marker stands for a sequence of terms, considered to be inserted into the sequence in place of the marker.

12.3.2 CommentedTerm

Description

Terms may have an attached comment.

Attributes

- comment: String [1] – supports comments on individual terms (or names).

Associations

- term: Term [1] in association CommentForTerm – links the comment to the term.
- Specialize Class Term.

Constraints

None.

Semantics

None.

12.3.3 FunctionalTerm

Description

A FunctionalTerm consists of a term, called the *operator*, and a term sequence called the *argument sequence*, containing terms called *arguments*.

Attributes

None.

Associations

- argument: Argument [0..*] in association ArgumentSequenceForFunctionalTerm – links zero or more additional terms (i.e., arguments) to a functional term.
- operator: Term [1] in association OperatorForTerm – links an operator to a functional term.
- Specialize Class Term.

Constraints

[1] The argument sequence of a functional term is ordered.

Semantics

See additional discussion of the semantics of functional term in CL in [ISO 24707].

12.3.4 SequenceMarker

Description

An argument sequence is a finite ordered sequence of bindings, which are terms or sequence markers. A sequence may be empty or may consist of a single sequence marker. Atomic sentences consist of an application of one term, denoting a relation, to a finite sequence of other terms. Such argument sequences may be empty, but they must be present in the syntax, as an application of a relation term to an empty sequence does not have the same meaning as the relation term alone.

Attributes

None.

Associations

- binding: Binding [1] in association BoundSequenceMarker – associates a sequence marker with the related binding in quantified sentences.
- Specialize Class Argument.

Constraints

[1] Names and sequence markers are disjoint syntax categories, and each is disjoint from all other syntax categories.

Semantics

Sequence markers take Common Logic beyond first-order expressiveness. A sequence marker stands for an arbitrary sequence of arguments. Sequence markers can be universally quantified, and a sentence containing such a quantifier has the same semantic import as the *infinite* conjunction of all the expressions obtained by replacing the sequence marker by a finite sequence of names, all universally quantified.

A dialect which does not provide for sequence markers, but is otherwise fully conformant, is a *compact* dialect, and may be described as a *fully conformant compact dialect* if it provides for all other constructions in the abstract syntax. Additional discussion on the semantics and use of sequence markers in CL is provided in the [ISO 24707] Common Logic specification.

12.3.5 Term

Description

A term is either a name or a functional term, or a term with an attached comment.

Attributes

None

Associations

- atomicSentence: AtomicSentence [0..*] in association PredicateForAtomicSentence – links a predicate (term) to the relation in which it participates.
- commentedTerm: CommentedTerm [0..1] in association CommentForTerm – provides the facility for commenting any CL term.
- function: FunctionalTerm [0..*] in association OperatorForFunction – links an operator to the FunctionalTerm that it is a part of.
- equation: Equation [0..*] in association LvalueForIdentity – links the term representing the ‘lvalue’ to an equation.
- equation: Equation [0..*] in association RvalueForIdentity – links the term representing the ‘rvalue’ to an equation.

Constraints

The Name / CommentedTerm / FunctionalTerm partition is disjoint.

```
context Term inv DisjointPartion:
  (self.oclIsKindOf(Name) xor self.oclIsKindOf(CommentedTerm) xor
    self.oclIsKindOf(FunctionalTerm))
```

Semantics

See additional discussion of the semantics of terms in CL in [ISO 24707].

12.4 The Atoms Diagram

Atomic sentences are similar in structure to terms, as shown in Figure 12.3. Equations are considered to be atomic sentences. Equations are distinguished as a special category because of their special semantic role and special handling by many applications.

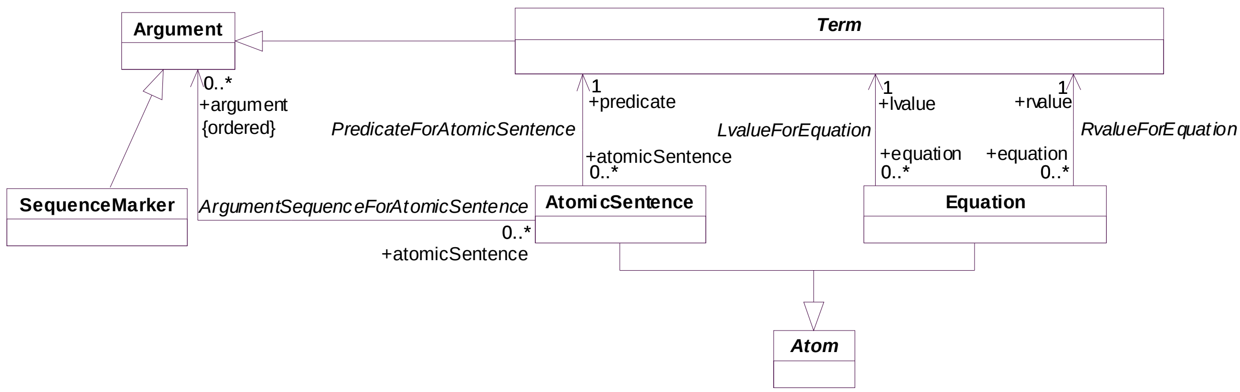

Figure 12.3 - Atomic Sentences

12.4.1 Atom

Description

An atom is either an *equation* containing two *arguments* which are terms, or consists of a term, called the *predicate*, and term sequence called the *argument sequence*, containing terms called *arguments* of the atom.

Attributes

None.

Associations

- Specialize Class Sentence

Constraints

[1] The AtomicSentence/Equation partition is disjoint.

```
context Atom inv DisjointPartition:
        (self.oclIsKindOf(AtomicSentence) xor self.oclIsKindOf(Equation))
```

Semantics

An atom, or atomic sentence, asserts that a relation holds between arguments. Its general syntactic form is that of a relation term applied to an argument sequence.

12.4.2 AtomicSentence

Description

An atomic sentence consists of a relation term (predicate) applied to an argument sequence.

Attributes

None.

Associations

- argument: Argument [0..*] in association ArgumentSequenceForAtomicSentence – links an argument sequence to the relation that the argument(s) participate in.
- predicate: Term [1] in association PredicateForAtomicSentence – links a predicate to the relation (atomic sentence) it participates in.
- Specialize Class Atom.

Constraints

None.

Semantics

See additional discussion of the semantics of relations in CL in [ISO 24707].

12.4.3 Equation

Description

An equation asserts that its arguments are equal and consists of exactly two terms.

Attributes

None.

Associations

- lvalue: Term [1] in association LvalueForIdentity – associates a term as the “lvalue” of the equation (identity relation).
- rvalue: Term [1] in association RvalueForIdentity – associates a term as the “rvalue” of the equation (identity relation).
- Specialize Class Atom.

Constraints

None.

Semantics

Equations are distinguished as a special category because of their special semantic role and special handling by many applications. See additional discussion of the semantics of equations in CL in [ISO 24707].

12.5 The Sentences Diagram

As shown in Figure 12.4, a sentence is either an atom, a boolean or quantified sentence, an irregular sentence, or a sentence with an attached comment, or an irregular sentence. *The current specification does not recognize any irregular sentence forms. They are included in the abstract syntax to accommodate future extensions to Common Logic, such as modalities for SBVR.*

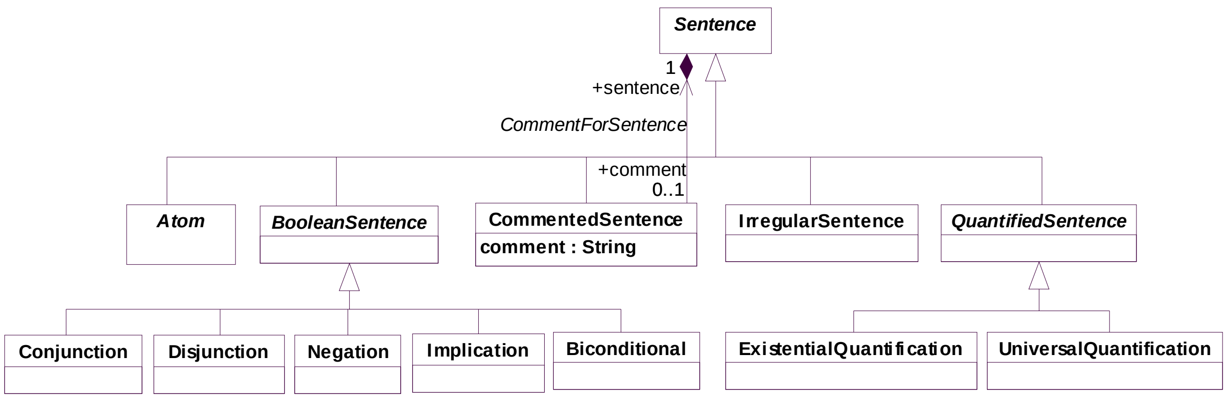

Figure 12.4 - Sentences

12.5.1 Biconditional

Description

A Biconditional (or equivalence), consisting of `(iff s1 s2)`, asserts that it is.

```
true if I(s1) = I(s2), otherwise false.
```

Attributes

None.

Associations

- lvalue: Sentence [1] in association LvalueForBiconditional – associates exactly one sentence as the 'lvalue' of the expression.
- rvalue: Sentence [1] in association RvalueForBiconditional – associates exactly one sentence as the 'rvalue' of the expression.
- Specialize Class BooleanSentence.

Constraints

None.

Semantics

This asserts that two sentences have the same truth value. See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.5.2 BooleanSentence

Description

BooleanSentence is an abstract class representing boolean sentences. A Boolean sentence has a type, called a *connective*, and a number of sentences called the *components* of the Boolean sentence. The number depends on the particular type. Every Common Logic dialect shall distinguish the *conjunction*, *disjunction*, *negation*, *implication* and *biconditional* types with respectively any number, any number, one, two and two components.

Attributes

None.

Associations

- Specialize Class Sentence.

Constraints

None.

Semantics

See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.5.3 CommentedSentence

Description

This feature enables annotation of sentences.

Attributes

- comment: String [1] – represents the comment about the sentence.

Associations

- sentence: Sentence [1] in association CommentForSentence – associates exactly one sentence as the argument of the expression.
- Specialize Class Sentence.

Constraints

None.

Semantics

No additional sentences.

12.5.4 Conjunction

Description

A conjunction, consisting of a set of conjuncts, `(and` s_1 `...` s_n`)`, asserts that it is

$false\ if\ I(s_i) = false\ \ for\ some\ i\ in\ 1 \ldots n,\ otherwise\ true.$

Essentially, a conjunction means that all its components are true. Note that *true* is defined as the empty case of a conjunction – there are no explicit definitions of true and false in CL. These definitions are conventional in formal logic and knowledge representation work.

Attributes

None.

Associations

- conjunct: Sentence [0..*] in association Conjunction – associates zero or more sentences as conjuncts of the expression.
- Specialize Class BooleanSentence.

Constraints

None.

Semantics

See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.5.5 Disjunction

Description

A disjunction, consisting of a set of disjuncts, $(or\ s_1\ \ldots\ s_n)$, asserts that it is

$true\ if\ I(s_i) = true\ for\ some\ i\ in\ 1 \ldots n,\ otherwise\ false.$

Essentially, a disjunction means that at least one of its components is true. Note that *false* is defined as the empty case of a disjunction.

Attributes

None.

Associations

- disjunct: Sentence [0..*] in association Disjunction – associates zero or more sentences as disjuncts of the expression.
- Specialize Class BooleanSentence.

Constraints

None.

Semantics

See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.5.6 ExistentialQuantification

Description

An existentially quantified sentence, consisting of `(exists (var) body)`, asserts that some things exist in the universe of discourse which satisfy the description in the body. An existentially quantified sentence means that its body is *true for some re-interpretation of its bindings*. Bindings may be names or sequence markers, which are re-interpreted respectively as referring to things or sequences of things, in the universe of discourse.

Attributes

None.

Associations

- Specialize Class QuantifiedSentence.

Constraints

None.

Semantics

See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.5.7 Implication

Description

An implication, consisting of `(implies s1 s2)`, asserts that it is

```
false if I(s1) = true and I(s2) = false, otherwise true.
```

Essentially, this means that the *antecedent* implies the *consequent*.

Attributes

None.

Associations

- antecedent: Sentence [1] in association AntecedentForImplication – associates exactly one sentence as the antecedent of the expression.
- consequent: Sentence [1] in association ConsequentForImplication – associates exactly one sentence as the consequent of the expression.
- Specialize Class BooleanSentence.

Constraints

None.

Semantics

See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.5.8 IrregularSentence

Description

Provides the placeholder for irregular sentences in the metamodel, potentially for use with modal sentence requirements for the Semantics for Business Vocabularies and Rules (SBVR) specification.

Attributes

None.

Associations

- Specialize Class Sentence.

Constraints

None.

Semantics

None.

12.5.9 Negation

Description

A negation, consisting of `(not s)`, asserts that it is `true if I(s) = false, otherwise false`. Essentially, a negation means that its inner sentence is false.

Attributes

None.

Associations

- sentence: Sentence [1] in association NegatedSentence – associates exactly one sentence as the argument of the expression.
- Specialize Class BooleanSentence.

Constraints

None.

Semantics

See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.5.10 QuantifiedSentence

Description

QuantifiedSentence is an abstract class representing quantified sentences. A quantified sentence has a type, called a *quantifier*, a finite sequence of names or sequence markers called *bindings*, and a sentence called the *body* of the quantified sentence. Every Common Logic dialect shall distinguish the *universal* and the *existential* types of quantified sentence.

Quantifiers may bind any number of variables; bindings may be restricted to a named category.

Attributes

None.

Associations

- body: Sentence [1] in association QuantificationForSentence – associates exactly one sentence (body) with the expression.
- binding: Binding [0..*] in association BindingSequenceForQuantifiedSentence – associates zero or more ordered bindings with the expression.
- Specialize Class Sentence.

Constraints

None.

Semantics

See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.5.11 UniversalQuantification

Description

A universally quantified sentence, consisting of `(forall (var) body)`, asserts that everything that exists in the universe of discourse satisfies the description in the body. A universally quantified sentence means that its body is true for any interpretation of its bindings. It consists of a sequence of bindings and a body that is a sentence.

Attributes

None.

Associations

- Specialize Class QuantifiedSentence.

Constraints

None.

Semantics

See additional discussion of the semantics of sentences in CL in [ISO 24707].

12.6 The Boolean Sentences Diagram

A Boolean sentence has a type, called a *connective*, and a number of sentences called the *components* of the Boolean sentence, as shown in Figure 12.5. The number depends on the particular type. Every common logic dialect **must** distinguish the *conjunction*, *disjunction*, *negation*, *implication*, and *biconditional* types with respectively any number, any number, one, two and two components.

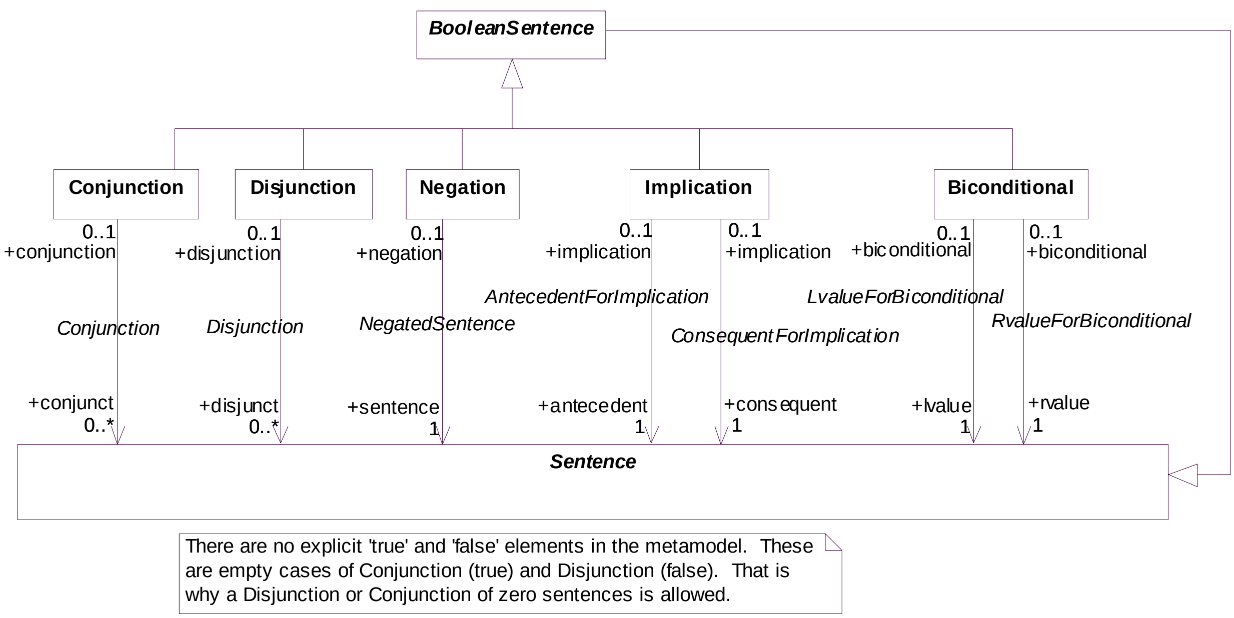

Figure 12.5 - Boolean Sentences

12.7 The Quantified Sentences Diagram

A quantified sentence has a type, called a *quantifier*, and a set of names called the *bindings*, and a sentence called the *body* of the quantified sentence, as shown in Figure 12.6. Every common logic dialect **must** distinguish the *universal* and the *existential* types of quantified sentence.

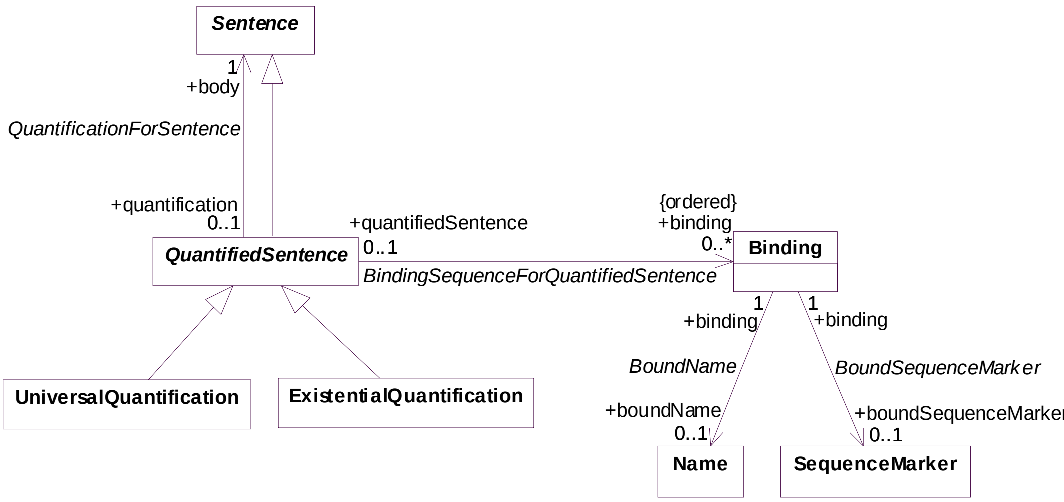

Figure 12.6 - Quantified Sentences

12.7.1 Binding

Description

A quantified sentence has (i) a type, called a *quantifier*, (ii) a finite, non-repeating sequence of names and sequence markers called the *binding sequence*, each element of which is called a *binding* of the quantified sentence, and (iii) a sentence called the *body* of the quantified sentence. A name or sequence marker which occurs in the binding sequence is said to be *bound* in the body. Any name or sequence marker which is not bound in the body is said to be *free* in the body.

Attributes

None.

Associations

- quantifiedSentence: QuantifiedSentence [0..1] in association BindingSequenceForQuantifiedSentence - associates an optional sentence with a binding.
- boundName: Name [0..1] in association BoundName - associates an optional name with a particular binding.
- boundSequenceMarker: SequenceMarker [0..1] in association BoundSequenceMarker - associates an optional sequence marker with a particular binding.

Constraints

[1] Name and SequenceMarker form a complete covering of Binding.

```
context Binding inv DisjointPartition:
    (self.oclIsTypeOf(Name) xor self.oclIsTypeOf(SequenceMarker))
```

Semantics

No additional semantics.

12.8 Summary of CL Metamodel Elements with Interpretation

Table 12.1 presents a summary of the elements in the metamodel (not exhaustive) with the corresponding elements of the core abstract syntax and their interpretation, derived from the summary given in 6.5 of the [ISO 24707] Common Logic Specification.

Table 12.1 - CL Metamodel Summary with Interpretation

CL Metamodel Element(s)	CL Core Syntax	Interpretation
	If E is an expression of the form	then I(E) =
Name	name N	int_I(N)
SequenceMarker	sequence marker s	seq_I(s)
Argument	term sequence t1...tn: [t1]...[tn]	$<I(t_1)...I(t_n)>$
Argument	term sequence $t_1...t_n$ with sequence marker s: $[t_1]...[t_n][s]$	$<I(t_1)...I(t_n)>;I(s)$
FunctionalTerm	term with operator o and term sequence s: ([o][s])	fun_I(I(o))(I(s)), i.e., the x such that $<(I(s),x)>$ is in (I(o))
Equation	atom which is an equation containing terms t_1, t_2	true if $I[t_1] = I[t_2]$, otherwise false
Atom, AtomicSentence	atomic sentence with predicate p and term sequence s	true if I(s) is in rel_I(I(p)), otherwise false
BooleanSentence, Negation	boolean sentence of type negation and component c	true if I(c) = false, otherwise false
BooleanSentence, Conjunction	boolean sentence of type conjunction and components $c_1...c_n$	true if $I(c_1) = ... = I(c_n)$ = true, otherwise false
BooleanSentence, Disjunction	boolean sentence of type disjunction and components $c_1...c_n$	false if $I(c_1) = ... = I(c_n)$ = false, otherwise true
BooleanSentence, Implication	boolean sentence of type implication and components c_1, c_2	false if $I(c_1)$ = true and $I(c_2)$ = false, otherwise true
BooleanSentence, Biconditional	boolean sentence of type biconditional and components c_1, c_2	true if $I(c_1) = I(c_2)$, otherwise false
QuantifiedSentence, UniversalQuantification	quantified sentence of type universal with bindings N and body B	true if for every N-variant J of I, J(B) is true; otherwise false

Table 12.1 - CL Metamodel Summary with Interpretation

CL Metamodel Element(s)	CL Core Syntax	Interpretation
QuantifiedSentence, ExistentialQuantification	`quantified sentence of type existential with bindings N and body B`	`true if for some N-variant J of I, J(B) is true; otherwise false`
Sentence, IrregularSentence	`irregular sentence [S]`	int_I(S)
Phrase, Sentence	`phrase which is a sentence: [S]`	*I*(S)
Phrase, Importation	`phrase which is an importation containing name N`	true if I(*text*(I(N))) = true, otherwise false.
Module, ExclusionSet, Text	`module with name N, exclusion set L and body text B`	true if [*I*<L](B) = true and rel_I(*I*(N)) = $UD_{[I<L]^*}$, otherwise false
Text	text containing phrases $S_1...S_n$	true if *I*(S_1) = … = *I*(S_n) = true, otherwise false.
Text	`a text T with a name N`	UR_I contains a named text value t with *text*(t) = T and *name*(t) = N

13 The Topic Map Metamodel

13.1 General

The Topic Maps Meta-Model is defined based primarily upon ISO 13250-2 Data Model [TMDM] and to a lesser degree ISO 13250-3 XML Syntax. The TMDM provides the most authoritative definition of the abstract syntax for Topic Maps. The following discussion assumes a basic understanding of Topic Maps.

The TMDM includes UML diagrams illustrating its data structures and their relationships. However, the normative specification is textual. The UML diagrams are informative only. The ODM includes a MOF 2 metamodel for Topic Maps to provide such a normative metamodel, and because one of the objectives coming out of the usage requirements analysis was to enable interoperability between UML, RDF, OWL CL, and Topic Maps. The latter requirement, in turn, requires a TM metamodel to support mappings, XMI, Java API generation, and interoperability with other MOF-compliant tools. The ODM metamodel for Topic Maps is similar to that specified in [TMDM], with named meta-associations, UML/MOF compliant naming conventions, and a few additional abstract classes.

13.2 Topic Map Constructs

Some of the primary elements in the TM meta-model are shown in Figure 13.1. Topic Maps are composed of a set of Topics and a set of Associations defining multi-way relations among those Topics.

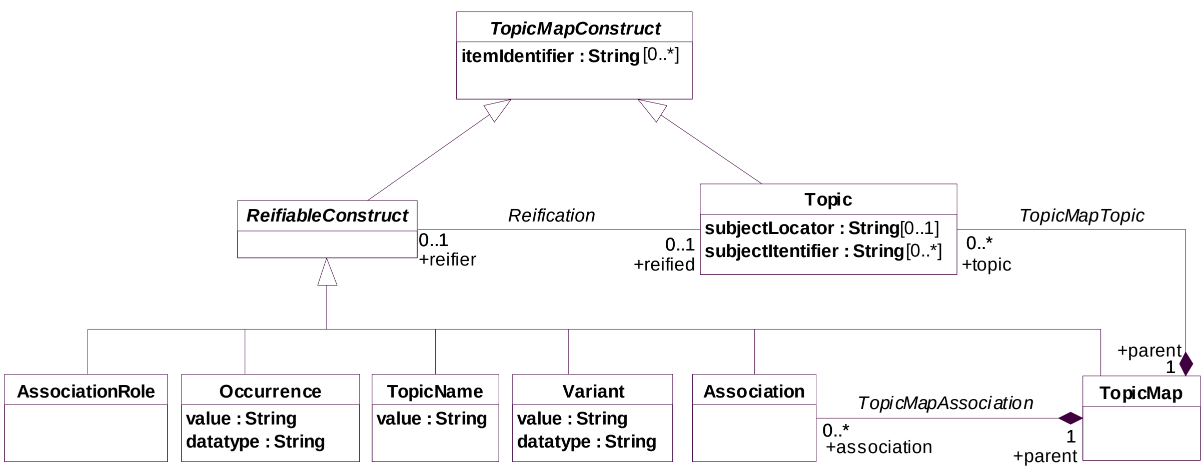

Figure 13.1 - Primary Elements in the Topic Map Metamodel

Each Topic is about a single Subject. Subjects in TM may be anything physical or conceptual. A machine addressable Subject will have a locator (e.g., a URL) while non-machine addressable subjects will have an identifier (e.g., the URL of a page about the subject or a URN). Topics are roughly equivalent to RDF Resources, describing elements in a world of discourse. Note that this similarity does not include RDF Literals that in TMs are not normally considered Topics.

13.2.1 TopicMapConstruct

Description

TopicMapConstructs are the abstract collection of elements that are part of any Topic Map. All first class elements are a sub-type of Topic Map Construct and may optionally have a Source Locator.

Attributes

- itemIdentifier [0..*] : String - each instance is identifying.

Associations

None.

Constraints

[1] It is an error for two different Topic Map Constructs to have source locators that are equal, expressed as the following OCL:

```
context TopicMapConstruct inv:
    TopicMapConstruct.allInstances()->
     forAll(v_tmc1, v_tmc2 | v_tmc1.itemIdentifier->
          forAll(v_sl1 | not(v_tmc2.itemIdentifier-> includes(v_sl1)))
    )
```

Semantics

The itemIdentifier assigned to a TopicMapConstruct allows references to it. ItemIdentifiers may be freely assigned to TopicMapConstructs based upon source syntax or other implementation defined methods.

13.2.2 ReifiableConstruct

Description

ReifiableConstruct defines an abstract class that groups together the kinds of topic map constructs that can be reified by being associated with a topic. All constructs except Topic can be reified.

Attributes

None.

Associations

- reified [0..1]: Topic in association Reification - relates the reifiable construct to the topic it reifies.
- Specialize class TopicMapConstruct.

Constraints

None.

Semantics

The act of reification is the act of making a topic represent the subject of another topic map construct in the same topic map. For example, creating a topic that represents the relationship represented by an Association is reification.

13.2.3 TopicMap

Description

A Topic Map represents a particular view of a set of subjects. It is a collection of MapItems.

Similar Terms

RDF Graph, Ontology.

Attributes

None.

Associations

- topic [0..*]: Topic in association TopicMapTopic – the set of Topics contained in this topic map.
- association [0..*]: Association in association TopicMapAssociation – the set of associations contained in this topic map.
- Specialize class ReifiableConstruct.

Constraints

None.

Semantics

A TopicMap itself does not represent anything, and in particular has no subject associated with it. It has no significance beyond its use as a container for Topics and Associations and the information about subjects they represent.

13.2.4 Topic

Description

Topic is the fundamental MapItem in a Topic Map. The class diagram for Topic is shown in Figure 13.2. Each Topic represents a Subject in the domain of discourse.

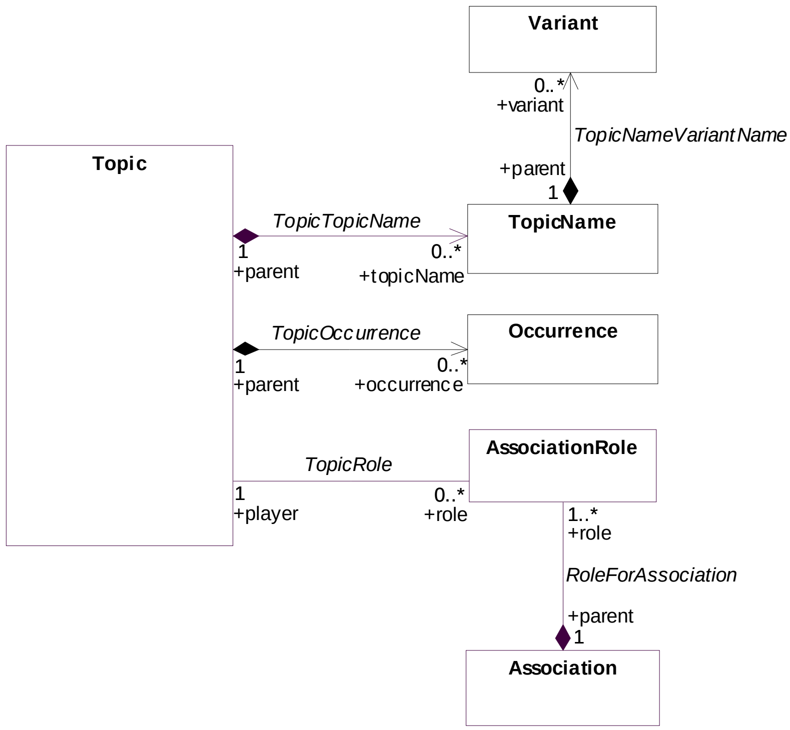

Figure 13.2 - The Topic Class

Similar Terms

Node, Resource, Entity.

Attributes

- subjectLocator [0..1]: String – an optional resource reference that locates a machine addressable subject.
- subjectIdentifier [0..*]: String – the set of 0 or more resource references that identify a machine addressable indicator of a non-machine addressable subject.

Associations

- occurrence [0..*] : Occurrence in association TopicOccurrence – the set of 0 or more occurrences for this Topic.
- parent [1] :TopicMap in association TopicMapTopic – the required TopicMap that this Topic is part of.
- reifier [0..1] : TopicMapConstruct in association Reification – a TopicMapConstruct may optionally be reified, becoming the subject of a Topic. A TopicMapConstruct is reified if it is another Topic's subjectIdentifier.
- role [0..*] : AssociationRole in association TopicRole – the collection of AssociationRoles that are the roles that this Topic plays in Associations.
- scopedConstruct [0..*] : ScopeAble in association ScopeTopic - the set of 0 or more instaces of ScopeAble of which this Topic is the scope.

- scopedVariant [0..*] : Variant in association VscopeTopic - the set of 0 or more instaces of Variant of which this Topic is the scope.
- topicName [0..*] : TopicName in association TopicTopicName – the set of 0 or more topic names for this Topic.
- typedConstruct [0..*] : TypeAble in association TypeTopic - the set of 0 or more instaces of TypeAble of which this Topic is the type.
- Specialize class TopicMapConstruct.

Constraints

[1] All topics must have a value for at least one subject identifier or subject locator that is neither the empty set nor null, expressed in the following OCL.

```
context Topic inv:
   self.subjectIdentifier->notEmpty() or
   self.subjectLocator->notEmpty()
```

Semantics

Each instance of Topic is associated with exactly one Subject. A subject indicator, subject identifier, or a subject locator identifies that subject. The Topic Map Data Model defines these terms in part as:

- A **subject indicator** is an information resource that is referred to from a topic map in an attempt to unambiguously identify the subject of a topic to a human being.
- A **subject identifier** is a locator that refers to a subject indicator.
- A **subject locator** is a locator that refers to the information resource that is the subject of a topic.

Topic maps contain only subject identifiers and subject locators, both of which refer to a subject indicator.

Topic is a very flexible concept. A topic can be a type whose instances are other topics, or associations, or association roles, or occurrences, or topic names. But a topic can be outside any type system, or a type which is an instance can itself be a type.

13.2.5 Association

Description

An Association is a multi-way relationship between one or more Topics. Associations must have a type and may be defined for a specified scope.

Similar Terms

Relation, Property.

Attributes

None.

Associations

- parent[1]:TopicMap in association TopicMapAssociation – the required TopicMap that this Association is part of.

- role [1..*]:AssociationRole in association RoleForAssociation – an instance of Association is required to be linked to at least one instance of AssociationRole.
- Specialize class ReifiableConstruct.
- Specialize class ScopeAble.
- Specialize class TypeAble.

Constraints

None.

Semantics

The relationship defined by an Association is a relationship among the included Topic's subjects, rather than the Topics themselves.

An association is an individual-level concept. Although an association must have a type, there is no constraint preventing different instances of the same type having different association roles, or for an association role for different instances to be linked to topics of different types, or no type at all.

13.3 Scope and Type

These 'Able' abstract classes are intended as a concise mechanism to give a specific set of meta-classes in the TM meta-model the capability to be typed and scoped; those meta-classes are shown in Figure 13.3.

13.3.1 ScopeAble

Description

ScopeAble defines an abstract class that provides the TM scoping mechanism. Subclasses of ScopeAble may have a defined scope of applicability.

Similar Terms

Context, Provenance, Qualification

Attributes

None.

Associations

- scope[0..*]: Topic in association ScopeTopic – the topics which define the scope.

Constraints

None.

Semantics

If the scope association is empty, then the ScopeAble items have the default scope.

13.3.2 TypeAble

Description

TypeAble defines an abstract class that provides the typing mechanism. Subclasses of TypeAble must define types. Elements in TM are singly typed. A typed construct is an instance of its type. Type describes the nature of the represented construct.

Similar Terms

Type, isA, kindOf.

Attributes

None.

Associations

- type [1..1]: Topic in association TypeTopic – the required topic which defines at most a single type.

Constraints

None.

Semantics

Typing is not transitive.

See also 13.4 discussing published subjects.

13.3.3 AssociationRole

Description

An Association is composed of a collection of roles, which are played by Topics. The AssociationRole captures this relation. A Topic in an Association plays a particular part or role in the Association. This is specified in an Association Role. The Association and Association Role construct is similar to a UML Association or to an RDF Property.

Similar Terms

Role, UML Association End, UML Property.

Attributes

None.

Associations

- parent[1]: Association in association RoleFor Association – the required Association which the AssociationRole is part of.
- player[1]: Topic in association TopicRole – the required Topic that plays this role in the parent Association.
- Specialize class ReifiableConstruct.
- Specialize class TypeAble.

Constraints

None.

Semantics

An AssociationRole is the representation of the participation of subjects in an association. The association role has a topic playing the role and a type that defines the nature of the participation of the player in the association. The roles and associations are representing the relationships between the participating Topic's subject, rather than the topics themselves.

AssociationRole is an individual-level concept. An instance of AssociationRole links an instance of Topic with an instance of Association.

13.3.4 Occurrence

Description

An Occurrence is very similar to an attribute.

Similar Terms

Attribute, Slot.

Attributes

- value[1]:String – If the datatype is IRI, a locator referring to the information resource the occurrence connects with the subject, otherwise the string is the information resource.
- datatype[1]:String – A locator identifying the datatype of the occurrence value.

Associations

- parent[1] : Topic in association TopicOccurrence - the required Topic which owns the Occurrence.
- Specialize class ReifiableConstruct.
- Specialize class ScopeAble.
- Specialize class TypeAble.

Constraints

None.

Semantics

It may be mistakenly inferred by the name 'Occurrence' that this construct refers only to instances of a Topic. This is not the case. An Occurrence may be any descriptive information about a Topic, including instances, and may represent any characteristic of a Topic, including an 'occurrence' or instance of the subject. Occurrences are semantically similar to UML Attributes.

.

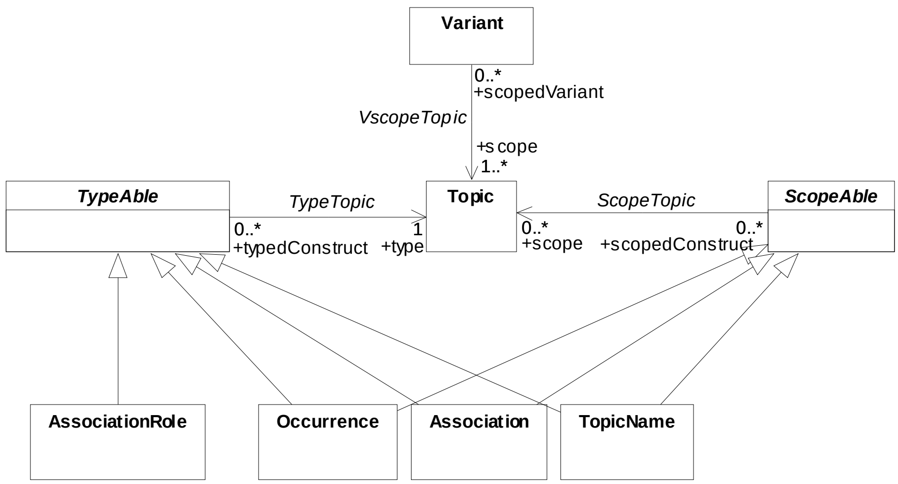

Figure 13.3 - Type and Scope

13.3.5 TopicName

Description

A TopicName is intended to provide a human readable text name for a topic.

Similar Terms

Label, Comment, Description (Brief).

Attributes

- value: String [1] – the Base Name for this Topic; the string is UNICODE.

Associations

- variant [0..*]: VariantName in association TopicNameVariantName – Zero or more variations of the TopicName.
- Specialize class ReifiableConstruct.
- Specialize class ScopeAble.
- Specialize class TypeAble.

Constraints

None.

Semantics

The term 'name' should not be misconstrued to imply uniqueness. Neither the topic name, nor its variants, are identifying; they serve only as human readable labels.

13.3.6 Variant

Description

Variant allows alternative names for a Topic to be specified. These names may be any format, including text, documents, images, or icons. Variants must have scope.

Similar Terms

Label, Comment, Description (Brief), Icon.

Attributes

- value[1]:String – If the datatype is IRI, a locator referring to the information resource the occurrence connects with the subject, otherwise the string is the information resource.
- datatype[1]:String – A locator identifying the datatype of the occurrence value.

Associations

- scope[1..*] : Topic in association VscopeTopic – one or more topics which define the scope.
- Specialize class ReifiableConstruct.

Constraints

[1] A Variant is restricted to being a composite part of a TopicName. It cannot exist as a standalone construct as constrained by the TopicNameVariantName association multiplicity.

Semantics

Like TopicName, variants are not identifying.

13.4 Published Subjects

A Core set of Topic instances, termed Published Subjects, has been defined as part of the TM standard. These topics represent special instances of the TM meta-model and any implementation of the TM meta-model should handle these items as special, reserved topics with meanings as defined in Clause 7 of the Topic Map Data Model [TMDM].

In summary, they represent five key areas:

1. Types and Instance –Types and their instances are related by three subjects representing the type-instance association and, the type and instance association roles. A type is an abstraction that captures characteristics common to a set of instances. A type may itself be an instance of another type, and the type-instance relationship is not transitive.

2. Super and Sub Types – Types may be arranged into a type hierarchy using the supertype-subtype association and, supertype and subtype association roles. The supertype-subtype relationship is the relationship between a more general type (the supertype) and a specialization of that type (the subtype). The supertype-subtype relationship is transitive.

3. Special Variant Names – Display and Sort are two special types of variant names appropriate for human display and sorting.

4. Uniqueness – A unique topic characteristic can be used to definitively identify a topic.

5. Topic Map Constructs –Subjects that represent the reification of topic map constructs, such as association, associations-role or occurrence.

These published subjects are identified by uri with base http://psi.topicmaps.com/iso13250/, called in the ODM by the Qname prefix ‘tmcore:’.

13.4.1 Type-Instance Relationship Among Topics

Besides being a type whose instances are Associations, AssociationRoles, Occurrences, or TopicNames a Topic can be a type whose instances are other Topics. This relationship is constructed as an instance of Association having two roles, the Association and each role having a type from the published subjects, as in the Instances diagram in Figure 13.4. The instance of Topic labeled ‘inst’ is declared as an instance of the instance of Topic labeled ‘type.’

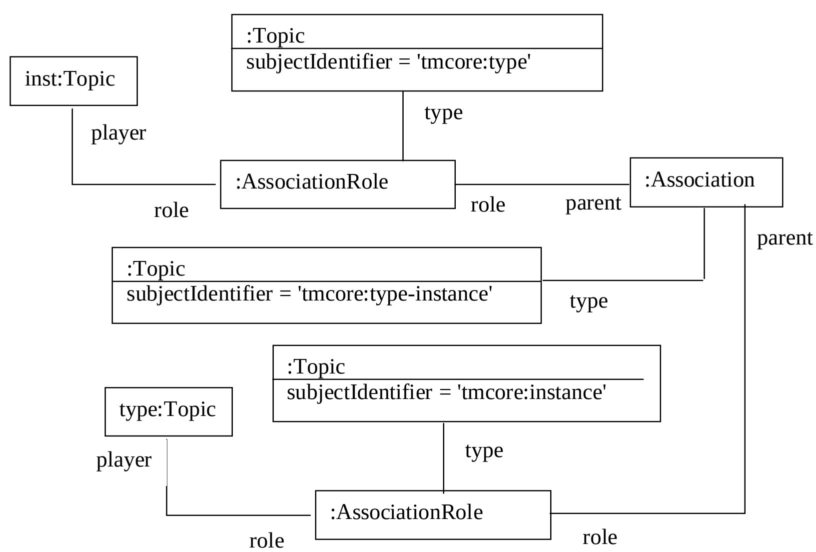

Figure 13.4 - Instances Diagram for Topic Type-Instance Relationship

13.4.2 Subtype-Supertype Relationship Among Topics

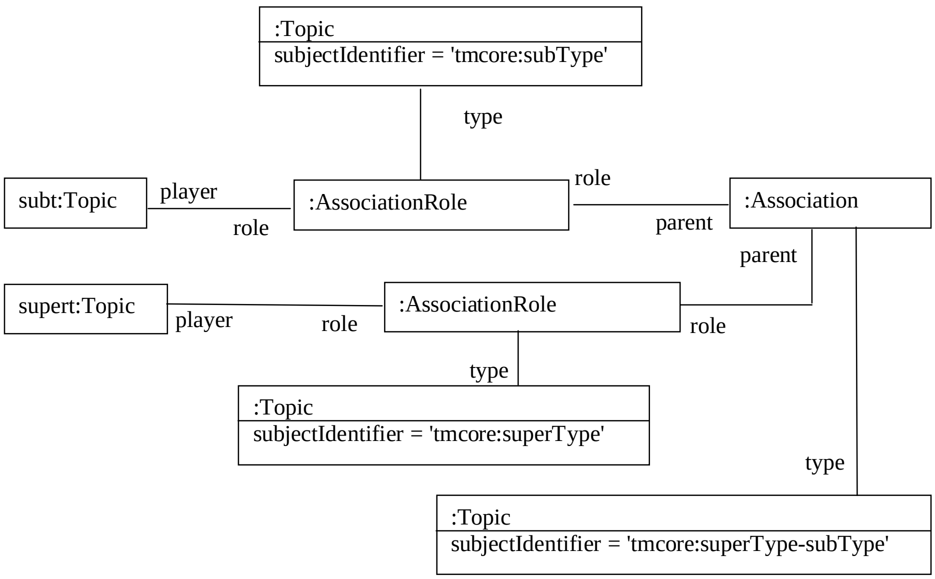

Figure 13.5 - Instances Diagram for Topic Subtype-Supertype Relationship

In a similar way, that two topics are in a subtype-supertype relationship is declared by an association with two roles, the association and both roles of types taken from the published subjects, as in Figure 13.5. The instance of Topic labeled 'subt' is declared as a subtype of the instance of Topic labeled 'supert.'

13.5 Example

Figure 13.6 depicts a simple instance model of the TM meta-model. The model depicted represents the following statements:

- A Personal Car is a Car (which may be owned by a Person).
- A Car is a Vehicle (which may have a Color).
- Carl is a person that owns a Personal Car that is red.

The parenthetical statements are not directly represented in Topic Maps.

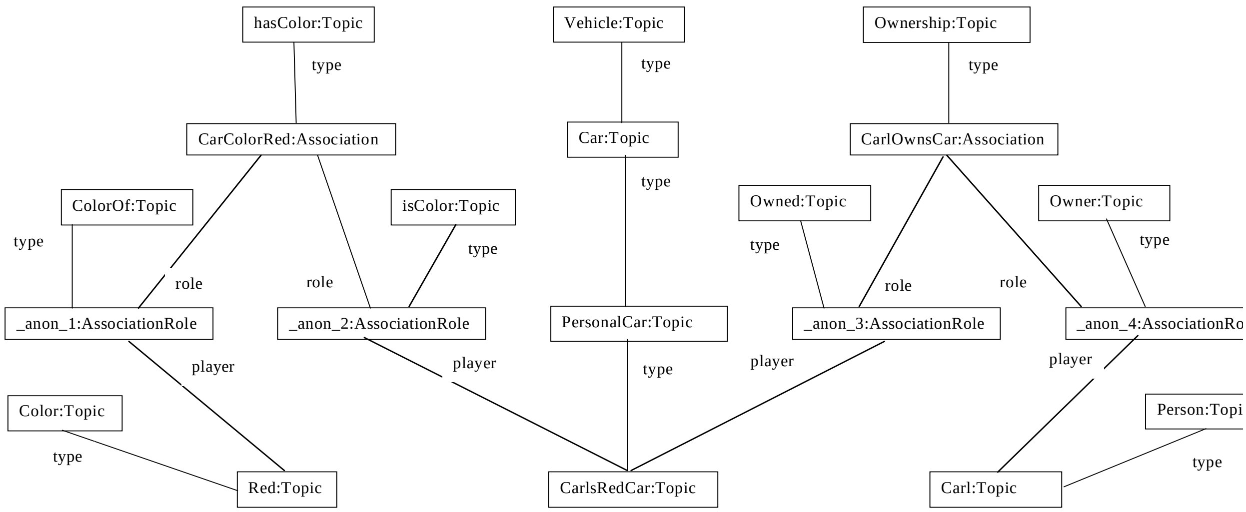

Figure 13.6 - Instance of Topic Map Metamodel

14 UML Profile for RDF and OWL

14.1 General

This profile is based on the UML Kernel package defined in “Unified Modeling Language: Superstructure,” version 2 [UML2] as well as on the Profiles section of the same document. It is designed to support modelers developing vocabularies in RDF and richer ontologies in the Web Ontology Language (OWL) through reuse of UML notation using tools that support UML2 extension mechanisms. The profile:

- Reuses UML constructs when they have the same semantics as OWL, or, when this is not possible, stereotypes UML constructs that are consistent and as close as possible to OWL semantics.
- Uses standard UML 2 notation, or, in the few cases where this is not possible, follows the clarifications and elaborations of stereotype notation defined in UML 2.1 (See [UML2.1], Profiles clause).
- Leverages the model library provided in Annex A.

The profile has been partitioned to support users who wish to restrict their vocabularies to RDF/S, as well as to reflect the structure of the RDF and OWL metamodels (and the languages themselves). It leverages stereotypes extensively and uses stereotype properties in traditional fashion. It complements the metamodels defined in Clause 10, and in Clause 11, respectively, for overall structure, semantics and language mapping. It also depends on model elements included in Annex A, for certain basic definitions, such as the M1 level elements discussed in Clause 8, Design Rationale.

14.2 UML Profile for RDF

14.2.1 RDF Profile Package

Description

The following sub clauses specify the set of stereotypes that comprise the UML2 profile for using UML to represent RDF/S vocabularies. It is designed to support all of RDF, as reflected in the RDF metamodel provided in Clause 10, with minor limitations, is based on the presumption that vendors (and users) are interested in developing RDF vocabularies that make use of the schema elements of the language in the context of the Web. The document related elements are recommended as a basis for exchanging namespace information, at a minimum, even in cases where a document representation is not required.

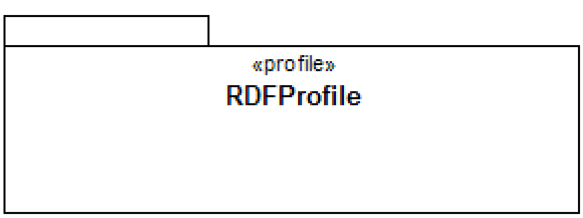

Figure 14.1 - RDF Profile Package

Constraints

[1] All classes in a package stereotyped by «rdfSource», «dataset», or «rdfDocument» must be stereotyped by «rdfsClass» (or an appropriate subclass, such as «owlClass»).

14.2.2 RDF Datasets, Documents and Sources

Stereotypes and other profile elements corresponding to the definitions for datasets, documents, and sources defined in the "RDF 1.1 Concepts and Abstract Syntax[3]" are defined in this sub clause and are shown in Figure 14.2.

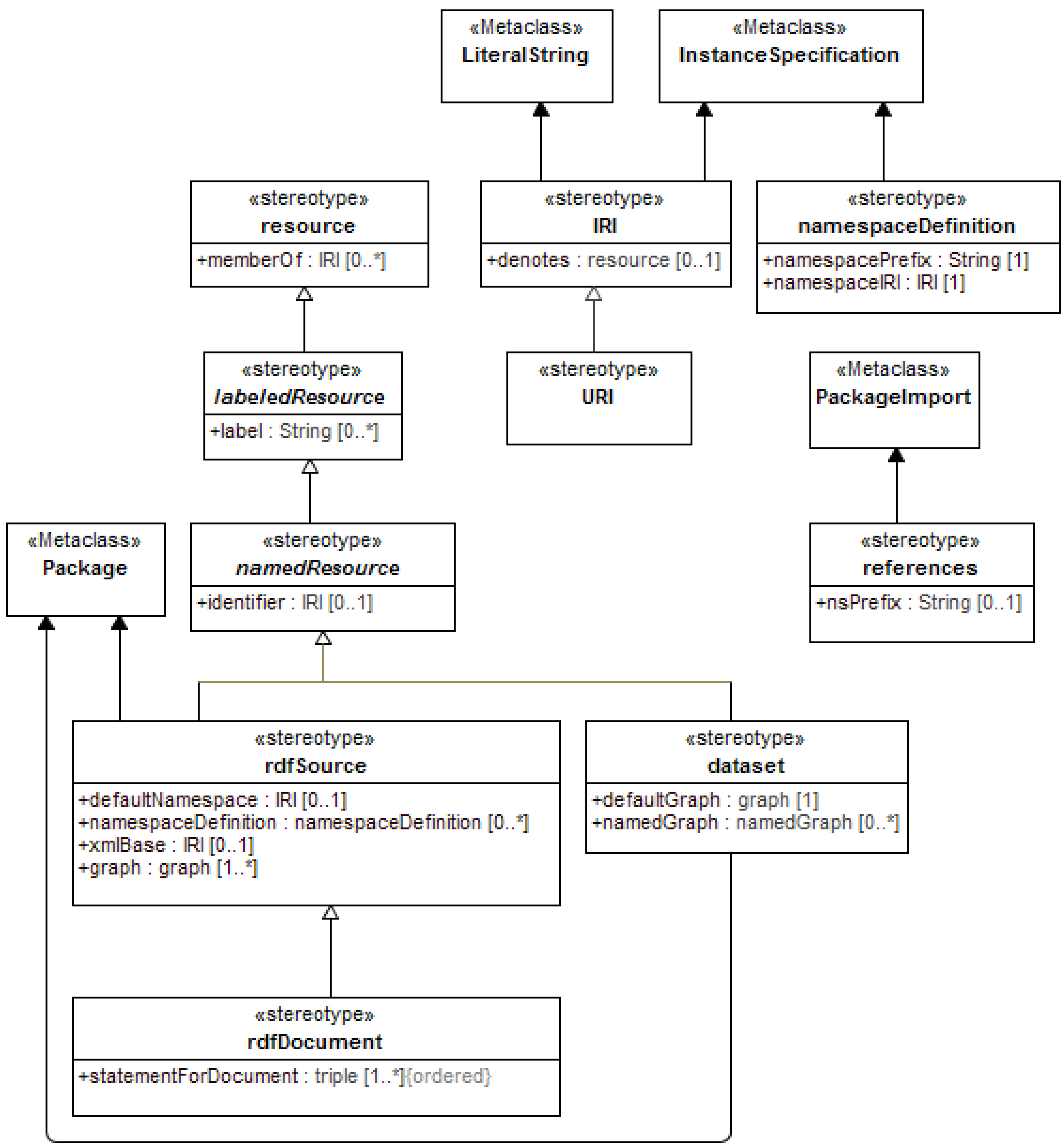

Figure 14.2 RDF Datasets, Documents, and Sources

3. http://www.w3.org/TR/2013/CR-rdf11-concepts-20131105/

Definitions for RDF graphs and named graphs are given in 14.2.3, together with the basic definition for RDF resource; labeled and named resources are defined in 14.2.9, below.

14.2.2.1 Datasets in RDF

Description

An RDF dataset is a collection of RDF graphs. Blank nodes may be shared between graphs in an RDF dataset.

Stereotype and Base Class

«dataset» with base class of UML::Package.

Parent

«namedResource».

Properties

- defaultGraph: graph [1] - the default graph for the dataset, which does not have a name and may be empty.
- namedGraph: namedGraph [0..*] - zero or more named graphs; each named graph is a pair consisting of the graph name and an RDF graph.

Constraints

None.

14.2.2.2 Internationalized Resource Identifiers (IRIs)

Description

An IRI (Internationalized Resource Identifier) is a Unicode string that conforms to the syntax defined in RFC 3987. IRIs are a generalization of URIs (Uniform Resource Identifiers, as defined in RFC 3986), that can include a broader range of Unicode characters.

IRIs in the RDF abstract syntax must be absolute, and may contain a fragment identifier. Some concrete RDF syntaxes allow the use of relative IRIs as a shorthand notation to make it easier to develop documents independently from their final publishing location. Relative IRIs must be resolved against a base IRI to make them absolute. Therefore, the RDF graph serialized in such syntaxes is well-defined only if a base IRI can be established per RFC 3986.

Stereotype and Base Class

«IRI» with base classes of UML::InstanceSpecification and UML::LiteralString.

Parent

None.

Properties

IRIs should be encoded directly in the *specification* property (for InstanceSpecifications) or the *value* property (for LiteralStrings) for the element the stereotype is applied to.

- denotes: resource [0..1] - provides a link to the referent that this IRI denotes, i.e., points to the resource it identifies.

Constraints

[1] The *specification* for the «IRI», in the case of an InstanceSpecification, or *value*, in the case of a LiteralString, must conform to the character encoding (including escape sequences, as necessary) defined in RFC 3987.

[2] The *specification* or *value* for the «IRI» must be present (i.e., is required).

14.2.2.3 Namespace Definitions

Description

A namespace is declared using a family of reserved attributes. These attributes, like any other XML attributes, may be provided directly or by default. Some names in XML documents (constructs corresponding to the non-terminal Name) may be given as qualified names. The prefix provides the namespace prefix part of the qualified name, and must be associated with a namespace URI in a namespace declaration.

Stereotype and Base Class

«namespaceDefinition» with base class of UML::InstanceSpecification.

Parent

None.

Properties

- namespacePrefix: String [1] - the string representing the namespace prefix.
- namespaceIRI: IRI [1] - the instance specification or literal string representing the namespace IRI.

Constraints

[1] The string value of namespacePrefix must conform to the specification given in [XMLNS].

[2] The string value of the namespace IRI must be a Unicode string that conforms to the syntax defined in RFC 3987, normalized according to Clause 5 of RFC 3987 if possible.

14.2.2.4 Documents in RDF

Description

An RDF document represents one of the common realization forms for an RDF/S vocabulary. The statements that comprise an RDF vocabulary may be considered ordered in the context of a document. This set of statements may also correspond to one or more graphs contained in the document.

Stereotype and Base Class

«rdfDocument» with base class of UML::Package.

Parent

«rdfSource».

Properties

- statementForDocument: triple [1..*] {ordered} – the set of ordered statements in the document.

Constraints

None.

14.2.2.5 Sources of RDF Graphs

Description

An RDF source is a persistent yet mutable source or container of RDF graphs. It is a resource that may be said to have a state that can change over time. A snapshot of the state can be expressed as an RDF graph. For example, any web document that has an RDF-bearing representation may be considered an RDF source. Like all resources, RDF sources may be named with IRIs and therefore described in other RDF graphs.

Stereotype and Base Class

«rdfSource» with base class of UML::Package.

Parent

«namedResource».

Properties

- defaultNamespace: IRI [0..1] - an optional default namespace for the source.
- namespaceDefinition: namespaceDefinition [0..*] - zero or more namespace definitions for the source, relating namespaces to preferred namespace prefixes, including an optional namespace prefix for the source as well as others used in the context of the source.
- namespaceSuffixDelimiter: String [0..1] - the default is '/' and should not need to be specified; used to indicate when "#" is preferred.
- xmlBase: IRI [0..*] - zero or more XML base namespace definitions associated with the source.
- graph: graph [1..*] - one or more graphs associated with the source.

Constraints

[1] Either a defaultNamespace or at least one xmlBase must be specified.

[2] The string value for any defaultNamespace property must conform to the character encoding defined in RFC 3986.

[3] The string value for any xmlBase property must conform to the character encoding defined in RFC 3986.

14.2.2.6 Referencing External Vocabularies

Description

Many RDF vocabularies and OWL ontologies refer to or import others. In a UML environment, this is typically accomplished through package imports. However, there are important distinctions in OWL in particular between referencing and reusing terms from an external ontology and importing it, which means committing to all of the statements and axioms in the imported ontology. The «references» stereotype provides the means to refer to an external vocabulary or ontology *without formally importing it*.

Stereotype and Base Class

«references» with base class of UML::PackageImport.

Parent

None.

Properties

- nsPrefix: String [0..1] - optional namespace prefix for the imported document, dataset or source.

Constraints

[1] The target of the «references» stereotype must be a UML::Package stereotyped by «dataset» or «rdfSource», or one of their children.

14.2.2.7 Uniform Resource Identifiers (URIs)

Description

The RDF 1.1 abstract syntax is concerned primarily with Internationalized Resource Identifiers, or IRIs, and uses the term IRI synonymously with what was called a URI or URI reference in RDF 1.0. A URI is actually defined as having a subset of the characters permitted in an IRI, and implementations should be careful to ensure conformance to RFC 3986, if so required. This definition and stereotype for URI is included for backwards compatibility purposes and to support implementations that require the more constrained URIs over IRIs.

Note that URIs and IRI/URI references specified using this RDF profile are globally defined, in contrast to naming and namespace conventions in UML2, which can be limited to the package level or to a set of nested namespaces.

Stereotype and Base Class

«URI» with base class of UML::InstanceSpecification and UML::LiteralString.

Parent

«IRI».

Properties

None.

Constraints

[1] The *specification* or *value* for the «URI» must conform to the character encoding (including escape sequences as necessary) defined in RFC 3986.

[2] The *specification* or *value* for the «URI» must be present (i.e., is required).

14.2.3 RDF Statements

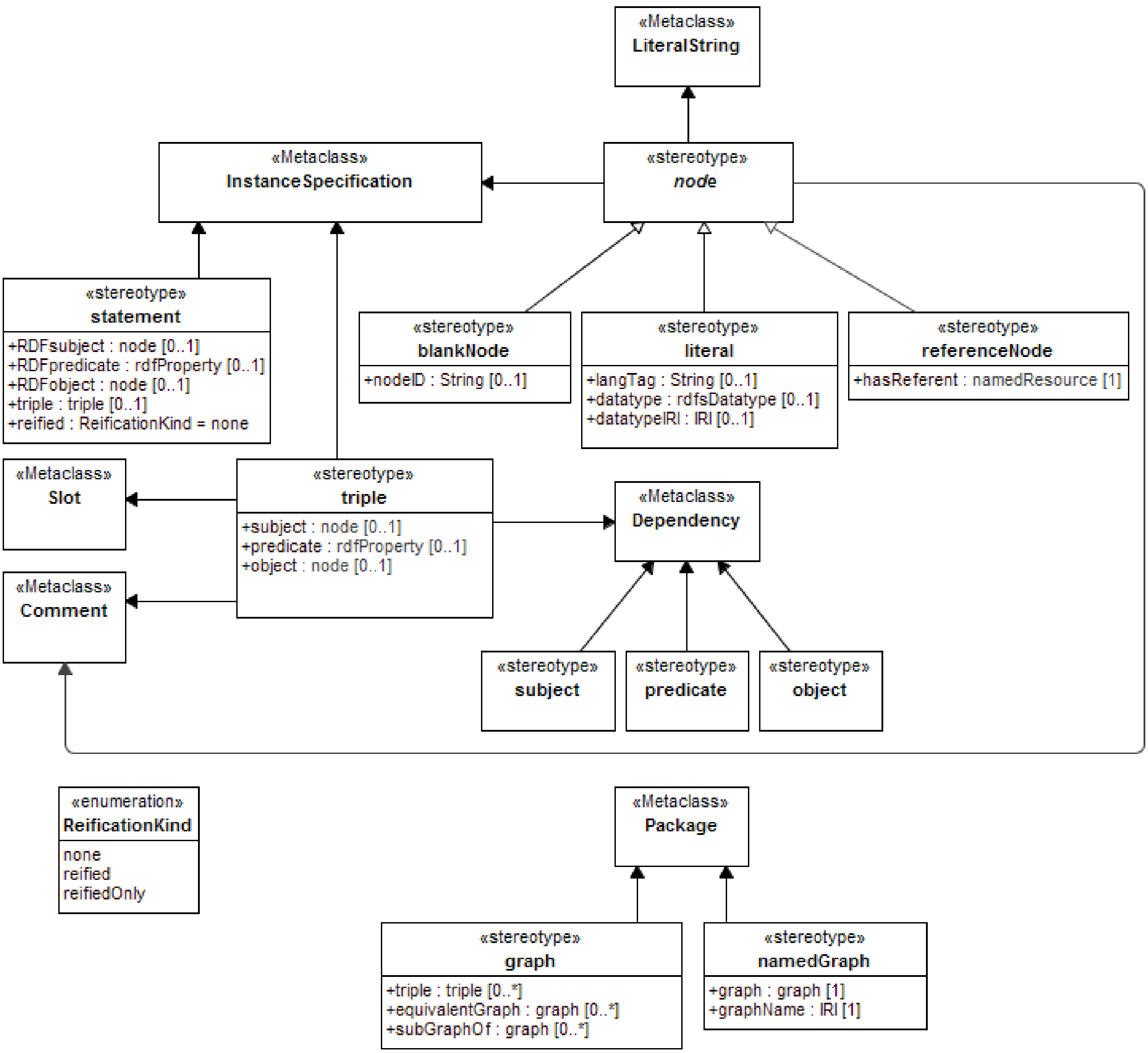

Figure 14.3 Graphs and Statements

The stereotypes and other profile elements corresponding to the RDF base definitions given in 10.2 are defined in this sub clause.

14.2.3.1 Blank Nodes

Description

A blank node is a node that is not an IRI or a literal. In the RDF abstract syntax, a blank node is simply a unique node that can be used in one or more RDF statements, but has no intrinsic name. Blank nodes are an integral part of the RDF language, and are used extensively in OWL class descriptions. In practice in a UML tool environment, it is likely that they will be needed when reverse engineering RDF vocabularies and OWL ontologies, and most importantly for coreference resolution when mapping across ontologies.

Stereotype and Base Class

«blankNode» with base classes of UML::Comment, UML::InstanceSpecification, and UML::LiteralString.

Parent

«node».

Properties

- nodeID: String [0..1] – provides an optional blank node identifier.

Constraints

[1] A Comment, InstanceSpecification, or LiteralString cannot be stereotyped by more than one of «blankNode», «literal», or «referenceNode» at a time. In other words, «blankNode» is disjoint with «literal» and «referenceNode».

14.2.3.2 Graphs

Description

An RDF graph is the container for the set of statements (subject, predicate, object subgraphs) in an RDF/S vocabulary. In UML this container is a package. This definition of a graph is included in the profile to support identification / componentization of RDF vocabularies through the use of named graphs, and for vocabulary mapping purposes.

Stereotype and Base Class

«graph» with base class of UML::Package.

Parent

None.

Properties

- equivalentGraph: graph [0..*] - links a graph to zero or more equivalent graphs.
- subGraphOf: graph [0..*] - links a graph to zero or more graphs that it is a sub-graph of.
- triple: triple [0..*] - links a graph to the triples it contains.

Constraints

[1] A package representing the value of an equivalentGraph or subGraphOf property must be stereotyped by «graph» or «namedGraph».

[2] The element representing the value of a triple property must be stereotyped by «triple».

14.2.3.3 Literals

Description

Literals are used to identify values such as numbers and dates by means of a lexical representation. Anything represented by a literal could also be represented by an IRI, but it is often more convenient or intuitive to use literals.

A literal may be the object of an RDF statement, but not the subject or the predicate.

Stereotype and Base Class

«literal» with base classes of UML::Comment, UML::InstanceSpecification, UML::LiteralString.

Parent

«node».

Properties

- langTag: String [0..1] - an optional language tag.
- datatype: rdfsDatatype [1] - associates a datatype with the literal.
- datatypeIRI: IRI [0..1] - associates the literal with its corresponding datatype IRI.

Constraints

[1] The string value associated with a «literal» must be a Unicode string in Normal Form C [Unicode].

[2] The language tag may only be used if the datatype IRI for the target literal is http://www.w3.org/1999/02/22-rdf-syntax-ns#langString.

[3] Language tags MUST be well-formed according to section 2.2.9 of BCP47.

[4] If the «literal» stereotype is applied to a literal string, then the value of the literal is the value of the literal string.

[5] If the «literal» stereotype is applied to an instance specification, then the value of the literal is the value of the specification property.

[6] If the «literal» stereotype is applied to comment, then the value of the literal is the value of the body of the comment.

14.2.3.4 Named Graphs

Description

Each named graph is a pair consisting of an IRI or a skolemized blank node (the graph name), and an RDF graph.

Stereotype and Base Class

«namedGraph» with base classes of UML::Package.

Parent

None.

Properties

- graph: graph [1] - links a named graph to the graph it names.
- graphName: IRI [1] - links a named graph to an IRI or skolemized blank node.

Constraints

[1] The *specification* for the «IRI», in the case of an InstanceSpecification, or *value*, in the case of a LiteralString, must conform to the character encoding (including escape sequences, as necessary) defined in RFC 3987, if present.

[2] The specification or *value* for the «IRI» or a skolemized blank node must be present (i.e., is required).

14.2.3.5 Nodes

Description

The set of nodes of an RDF graph is the set of subjects and objects of triples in the graph. It is possible for a predicate IRI to also occur as a node in the same graph.

Stereotype and Base Class

«node», which is an abstract stereotype, with base classes of UML::Comment, UML::InstanceSpecification, and UML::LiteralString.

Parent

None.

Properties

None.

Constraints

None.

14.2.3.6 The *Object* of a Triple

Description

This stereotype provides one of two mechanisms for referring to the object of a triple or statement.

Stereotype and Base Class

«object» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

None.

14.2.3.7 The *Predicate* of a Triple

Description

This stereotype provides one of two mechanisms for referring to the predicate of a triple or statement.

Stereotype and Base Class

«predicate» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

None.

14.2.3.8 Resources

Description

Any IRI or literal denotes something in the world (the universe of discourse). These things are called resources. Anything can be a resource, including physical things, documents, abstract concepts, numbers and strings; the term is synonymous in RDF with entity in other representation paradigms.

Stereotype and Base Class

«resource» with base class of UML::Element.

Parent

None.

Properties

- memberOf: IRI [0..*] – zero or more IRIs linking the resource to other resources in terms of membership (i.e., as in a class or container).

Constraints

[1] If the memberOf property is present, the value(s) must be stereotyped by «IRI», and the *specification* for the «IRI», in the case of an InstanceSpecification, or *value*, in the case of a LiteralString, must conform to the character encoding (including escape sequences, as necessary) defined in RFC 3987.

14.2.3.9 RDF Statements

Description

An RDF statement is an instance of an `rdfs:Class` used to talk about a particular triple, or element of a triple. It provides a link to the elements of the triple and then allows statements to be made about those elements.

Note that the properties of «statement» are optional in that there are two mechanisms for using the «statement» stereotype to connect elements of a triple with something one wants to say about that triple. These are (1) the properties defined on the stereotype itself, and (2) the set of dependencies defined in this section that allow users to link to the nodes and arcs one is reifying rather than encode them as properties of the stereotype. Modelers can then either link a specific triple that makes the statement via the property or use, for example, annotations to make such statements.

Stereotype and Base Class

«statement» with base class of UML::InstanceSpecification.

Parent

None.

Properties

- subject: node [0..1] - the resource that is the subject of the statement.
- predicate: rdfProperty [0..1] - the predicate for the statement.
- object: node [0..1] - the resource that is the object of the statement.
- triple: triple [0..1] - the triple representing the “thing” we want to say about this reified triple (i.e., the statement about the reified triple).
- reified: ReificationKind [1] – indicates whether or not a particular statement (triple) is reified but not asserted, reified, or neither; default value is “none.”

Constraints

[1] The value of the subject property must be a UML::InstanceSpecification stereotyped by «node» or one of its children.

[2] The value of the predicate property must be stereotyped by «rdfProperty» or one of its children.

[3] The value of the object property must be a UML::InstanceSpecification stereotyped by «node» or one of its children.

[4] If the value of the reified property is “reified,” then the subject, predicate, and object must be filled.

14.2.3.10 Reference Nodes

Description

The set of nodes in an RDF graph include IRIs, literals, and blank nodes. A predicate IRI can also represent a node in a graph. The collection of IRIs, literals, and blank nodes are also called RDF terms. The «referenceNode» stereotype represents the terms that are IRIs and predicate IRI nodes.

Stereotype and Base Class

«referenceNode» with base class of UML::InstanceSpecification and UML::LiteralString.

Parent

«node».

Properties

- hasReferent: namedResource [1] - the model element that the reference node refers to.

Constraints

None.

14.2.3.11 Reification

Description

A particular resource may be identified by more than one URI/IRI reference, and may be reified by another resource, represented by a dependency between instance specifications (resources) stereotyped by «reifies».

Stereotype and Base Class

«reifies» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

[1] The «reifies» stereotype can only be applied to a UML::Dependency that links two model elements that are stereotyped by «resource» (or any of its children). The dependency must be navigable from the reifying resource to the resource it reifies.

14.2.3.12 The *Subject* of a Triple

Description

This stereotype provides one of two mechanisms for referring to the subject of a triple or statement.

Stereotype and Base Class

«subject» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

None.

14.2.3.13 Triples

Description

A triple is the means by which one can say things (i.e., state facts) in RDF. An RDF triple consists of three components:

- the subject, which is an IRI or a blank node.
- the predicate, which is an IRI.
- the object, which is an IRI, a literal or a blank node.

An RDF triple is conventionally written in the order subject, predicate, object.

Note that there are a number of ways to represent triples in the RDF profile, which are typically, but not always, represented as UML::InstanceSpecifications. Flexibility is provided here to facilitate alternative patterns as needed depending on the vocabulary and related modeling use case.

Stereotype and Base Class

«triple» with base classes of UML::Comment, UML::Dependency, UML::InstanceSpecification, and UML::Slot.

Parent

None.

Properties

- subject: node [0..1] - the subject of the triple.
- predicate: rdfProperty [0..1] - the predicate of the triple.
- object: node [0..1] - the object of the triple.

Constraints

None.

14.2.4 ReificationKind

Description

ReificationKind is an enumerated type used by the reification property on RDFStatement. It has three possible values: none (which is the default value), reified (meaning that the statement is both asserted and reified), and reifiedOnly (meaning that a statement is reified but not asserted - providing a placeholder in a UML model representing an RDF vocabulary for such a statement, which is necessary when we want to say things about statements that occur in some external vocabulary that are not available to this model).

Stereotype and Base Class

No stereotype is defined for this enumeration.

Parent

None.

Properties

None.

Constraints

[1] ReificationKind has three possible values: ‘none,’ ‘reified,’ and ‘reifiedOnly.’

[2] The default value of ReificationKind is ‘none.’

14.2.5 Literals

The stereotypes used to specify literals include the definition for «literal» given in 14.2.3.3, and those defined in this section, as shown in Figure 14.4. These include basic literals as well as built-in RDF schema properties such as `rdfs:label` and `rdfs:comment`.

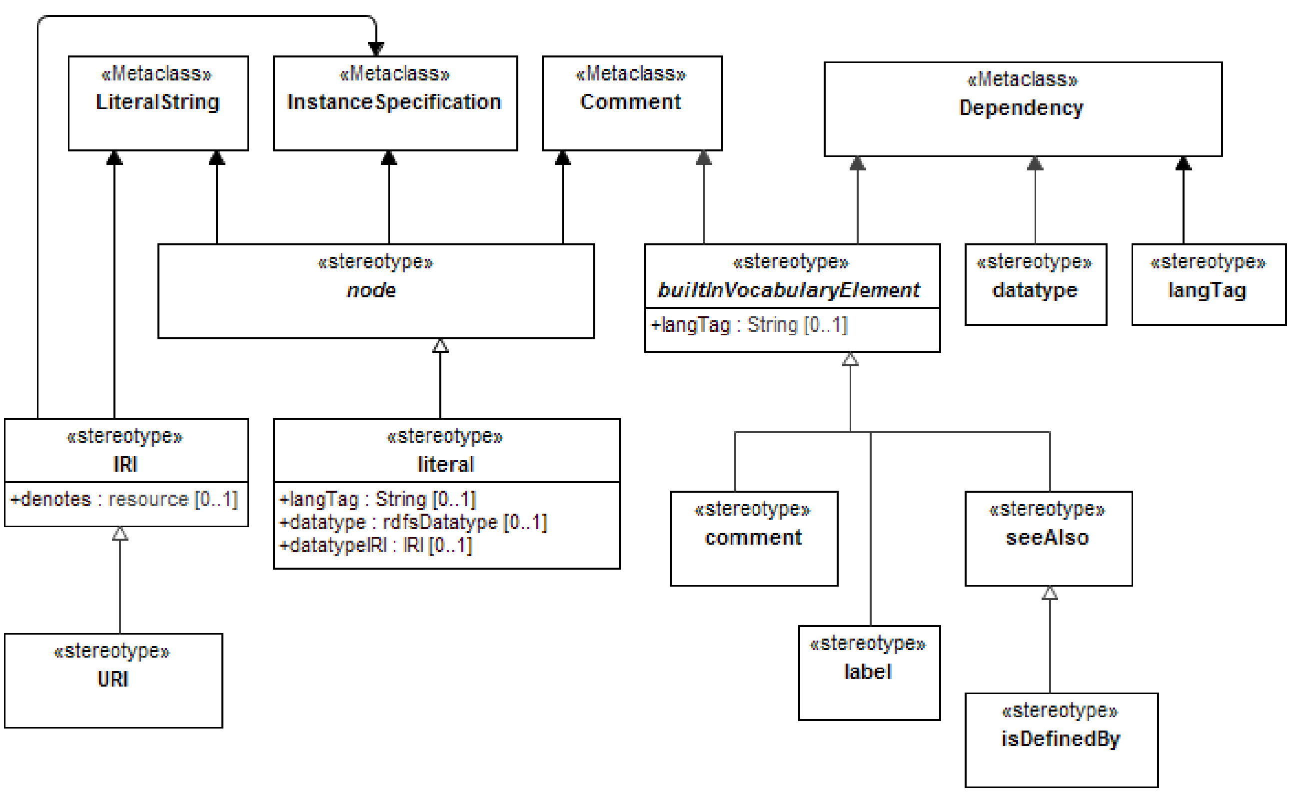

Figure 14.4 Literals and Built-in Vocabulary in RDF/S

14.2.5.1 Built-in Vocabulary Element

Description

RDF Schema includes four built-in properties whose values are literals or resources. These are `rdfs:comment`, `rdfs:isDefinedBy`, `rdfs:label`, and `rdfs:seeAlso`. In order to properly show these as built-ins, the RDF profile includes an abstract stereotype to represent these properties, which each of them extends.

Typically, `rdfs:label` and `rdfs:comment` have string values, with or without language tags, whereas `rdfs:isDefinedBy` and `rdfs:seeAlso` are primarily used to refer to IRIs or other resources, including bibliographic citations. Instances of these built-in properties may have string values that are represented as UML literal strings stereotyped by «literal», with a UML dependency bearing the name of the built-in property linking the element being modified (e.g., a class or individual) to the literal string. For lengthy definitions or comments, however, it may be more convenient to represent these as UML comments directly. In rare circumstances, use of a UML instance specification for the target literal may be preferred, for example for representation of IRIs, with the value of the literal encoded in the *specification* of the InstanceSpecification. In this case, the built-in property dependency would point to the InstanceSpecification rather than to a LiteralString. This approach is also appropriate in cases where a citation or other reference is the target of the property. Use of this common, abstract "builtInVocabularyElement" stereotype that extends both UML::Comment and UML::Dependency provides the means for maximum flexibility in modeling these built-in properties.

Stereotype and Base Class

«builtInVocabularyElement», with base classes of UML::Comment and UML::Dependency.

Parent

None.

Properties

- langTag: String [0..1] - an optional language tag.

Constraints

[1] The language tag may only be used if the datatype IRI for the target literal is http://www.w3.org/1999/02/22-rdf-syntax-ns#langString.

[2] Language tags MUST be well-formed according to 2.2.9 of BCP47.

[3] Language tags may only be used if the base class for the stereotype is UML::Comment.

14.2.5.2 Comment Property

Description

A comment is a literal, contained by the resource it describes, and can be applied to any resource. Typically, an `rdfs:comment` property will be represented as a UML Dependency with a string value as its target, specified as a «literal», with or without a language tag, or as a UML Comment.

Stereotype and Base Class

«comment» with base classes of UML::Comment and UML::Dependency.

Parent

«builtInVocabularyElement»

Properties

None.

Constraints

No additional constraints.

14.2.5.3 Datatype

Description

In cases where LiteralStrings or InstanceSpecifications are used to represent RDF literals, modelers may represent the datatype associated with the literal as a property value, using the datatype tag on the «literal», or via a UML dependency stereotyped by «datatype» whose target is a UML::Datatype stereotyped by «rdfsDatatype». It can be particularly useful when connecting literal values to custom datatypes defined through data restrictions, for example. This datatype stereotype is optional, and does not represent an `rdfs:Datatype` (defined in 14.2.6.3, below), but rather a reference to the datatype that types the literal, and simply provides an alternate mechanism for associating a datatype with a literal for modeling flexibility.

Stereotype and Base Class

«datatype» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

No additional constraints.

14.2.5.4 IsDefinedBy Property

Description

`rdfs:isDefinedBy` provides a means to indicate that a particular resource (the source, or owning classifier) is defined by another resource (the target resource). Note that RDF does not constrain the usage of `rdfs:isDefinedBy`, though in practice, vocabularies that use this construct, such as the Dublin Core, will do so.

Stereotype and Base Class

«isDefinedBy» with base classes of UML::Comment and UML::Dependency.

Parent

«seeAlso».

Properties

None.

Constraints

None.

14.2.5.5 - Label Property

Description

A label is a literal, contained by the resource it describes, and provides a human-readable label, or "pretty name" that can be applied to any resource. Typically, an `rdfs:label` property will be represented as a UML Dependency with a string value as its target, specified as a «literal», with or without a language tag, or as a UML Comment.

Stereotype and Base Class

«label» with base class of UML::Comment and UML::Dependency.

Parent

«builtInVocabularyElement».

Properties

None.

Constraints

No additional constraints.

14.2.5.6 Language Tags

Description

Similarly to the «datatype» stereotype defined in 14.2.5.3, where literal strings or instance specifications are used to represent RDF literals, modelers may choose to represent the language tag using the langTag tag on the «literal», or via a UML dependency stereotyped by «langTag» whose target is another literal, also stereotyped by «literal», whose *specification* or *value* contains the actual language tag conforming to the constraints defined for language tags as described for «literal», above. This «langTag» stereotype is optional and simply provides an alternate mechanism for modeling flexibility. Note that language tags defined using this approach can be shared, i.e., have multiple «langTag» dependencies pointing to the same «literal» representing the tag.

Stereotype and Base Class

«langTag» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

None.

14.2.5.7 SeeAlso Property

Description

`rdfs:seeAlso` indicates that more information about a particular resource (the source, or owning classifier) can be found at the target resource.

Stereotype and Base Class

«seeAlso» with base class of UML::Comment and UML::Dependency.

Parent

«builtInVocabularyElement».

Properties

None.

Constraints

No additional constraints.

14.2.6 Classes and Datatypes

The stereotypes associated with the definitions given in 10.5, in addition to those given above, are defined in this sub clause.

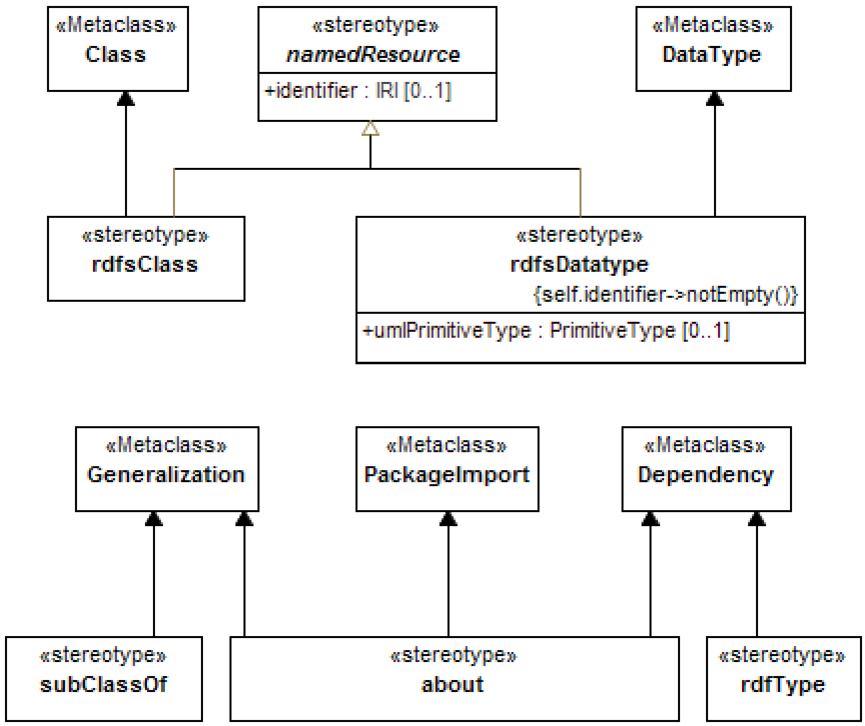

Figure 14.5 Classes and Datatypes

14.2.6.1 Referencing Node Elements

Description

One of the primary goals of the Semantic Web and Linked Data is to be able to link to, or reuse, various nodes in any vocabulary. The more we can link to other definitions, the more connected we are, or our content is. There is a syntactic element in the RDF/XML specification used to reference node elements in the current or an external graph, called `rdf:about`, which we actually need for similar purposes in UML. This allows us to say additional things about the element we are referencing, particularly those that are imported from another vocabulary or ontology, in a systematic way.

Stereotype and Base Class

«about» with base classes of UML::Dependency, UML::Generalization, and UML::PackageImport.

Parent

None.

Properties

None.

Constraints

None.

14.2.6.2 Classes in RDF Schema

Description

The collection of resources that represents RDF Schema classes is itself a class, called `rdfs:Class`. Classes provide an abstraction mechanism for grouping resources with similar characteristics. The members of a class are known as instances of the class. Classes are resources. They are often identified by RDF URI References and may be described using RDF properties.

An RDFS class maps closely to the UML definition of a class; one notable exception is that an RDFS class may have a URI reference. The definition of the «rdfsClass» stereotype corresponds to 10.5.1.

Stereotype and Base Class

«rdfsClass» with base class of UML::Class.

Parent

«namedResource».

Properties

None.

Constraints

None.

14.2.6.3 Datatypes in RDF Schema

Description

`rdfs:Datatype` represents the class of datatypes in RDF. Instances of `rdfs:Datatype` correspond to the RDF model of a datatype described in the RDF Concepts specification [RDF Concepts]. Note that built-in instances of `rdfs:Datatype` correspond to the subset of datatypes (defined in [XML Schema Datatypes]) allowable for use in RDF, as specified in [RDF Concepts]. These are provided for use with the metamodel(s) and profile(s) in the model library given in Annex A ("Foundation Library (M1) for RDF and OWL"). Use of user-defined datatypes should be carefully considered against any desire for reasoning over an RDF vocabulary, OWL ontology, or knowledge base.

Stereotype and Base Class

«rdfsDatatype» with base class of UML::Datatype.

Parent

«namedResource».

Properties

- umlPrimitiveType:PrimitiveType [0..1] - the UML primitive type corresponding to the datatype.

Constraints

None.

14.2.6.4 RDF subClassOf Property

Description

`rdfs:subClassOf` indicates that the resource is a subclass of the general class; it has the semantics of UML Generalization. However, classes on both ends of the generalization must be stereotyped «rdfsClass», or «owlClass», if used with the profile for OWL.

Note in OWL DL that mixing inheritance among RDFS and OWL classes is permitted, as long as proper subclassing semantics is maintained. In order for a model to be well formed an OWL class can be a subclass of an RDFS class but not vice versa. In other words for OWL DL, once you're in OWL you need to stay there.

Stereotype and Base Class

«subClassOf» with base class of UML::Generalization.

Parent

None.

Properties

None.

Constraints

[1] Classes on both ends of the generalization must be stereotyped «rdfsClass», or «owlClass», if used with the profile for OWL.

[2] In OWL DL, a class stereotyped by «owlClass» may specialize a class stereotyped by «rdfsClass», but not vice versa.

14.2.6.5 RDF Types

Description

`rdf:type` maps to the relation between instance and classifier in UML. This can be represented in a UML model by the relation between an instance specification and its classifiers. In some cases, it may also be convenient to show this relationship directly in a diagram, and therefore we provide this optional stereotype as a dependency from the resource (or individual or literal in OWL) to its classifier(s).

Stereotype and Base Class

«rdfType» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

None.

14.2.7 Properties in RDF

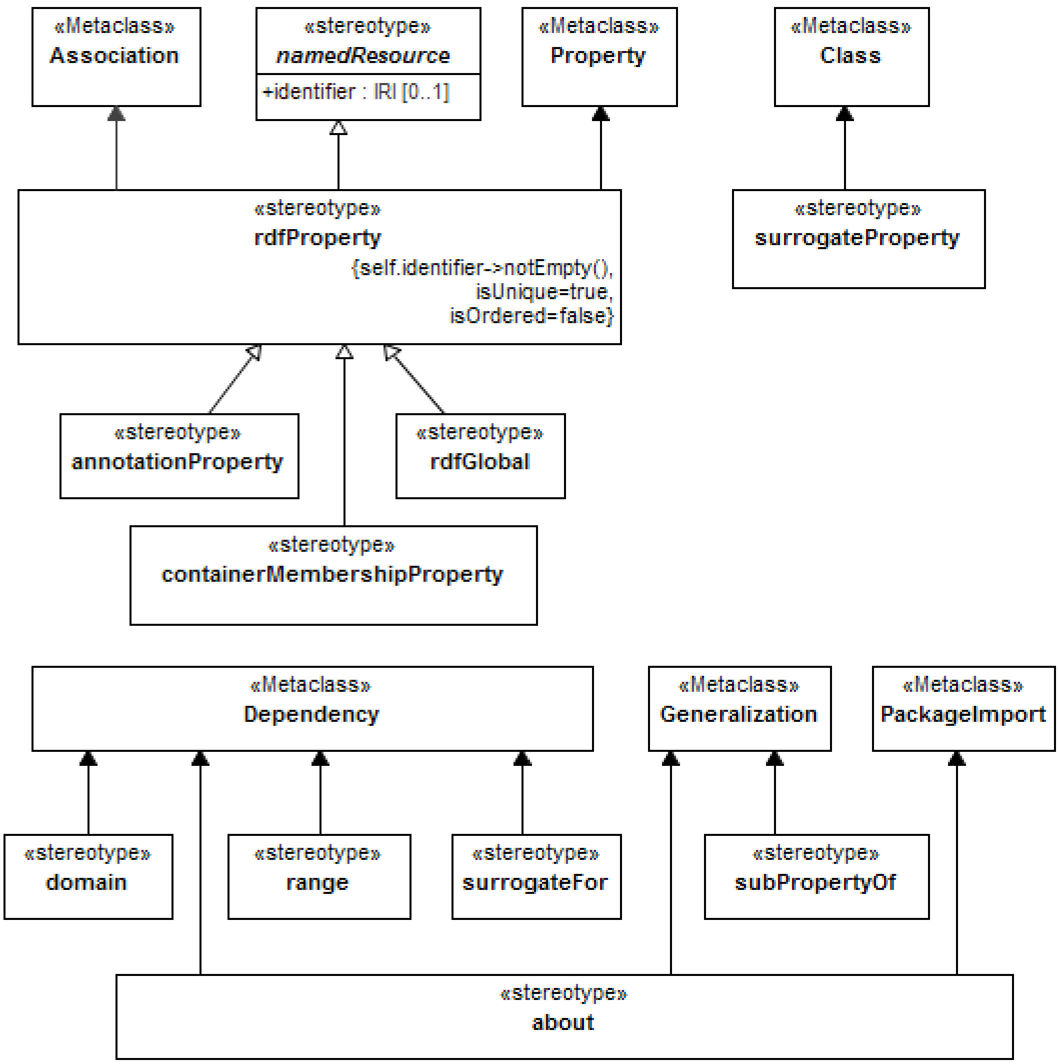

Figure 14.6 Properties in RDF

14.2.7.1 RDF Properties

Description

A property in UML can be defined as part of an association or not. When it is not part of an association, the property is owned by the class that defines its domain, and the type of the property is the class that defines its range. When a property is part of an association, the association is binary, with the class that defines the domain of the property owning that property.

Properties in RDF and OWL are defined globally, that is, they are available to all classes in all vocabularies and ontologies – not only to classes in the vocabulary or ontology they are defined in, but to classes in other vocabularies and ontologies, including those that are imported. For RDF properties that are defined without specifying a domain or range, the profile allows users to

- use an anonymous UML::Class, stereotyped by «rdfsClass», that is either the RDFS library class representing `rdfs:Resource` (analogous to `owl:Thing` in OWL ontologies) or `rdfs:Class`, for the "missing" domain end class, and
- either of those options or a UML::Datatype, stereotyped by «rdfsDatatype», library element representing `rdfs:Literal`, as appropriate, for the "missing" range end class.

Stereotype and Base Class

«rdfProperty» with base class of UML::AssociationClass, UML::Property, and UML::Association.

Parent

«namedResource».

Properties

None.

Constraints

[1] `HasIRI: self.identifier->notEmpty()` - the IRI (identifier) for the property is required.

[2] Properties cannot have the same value twice (i.e., in UML, isUnique=true).

[3] Property values are not ordered (i.e., in UML, isOrdered=false).

Graphical Representation

There are several alternatives for representing various aspects of RDF properties in UML, as follows.

A. There are three ways to represent an RDF property in UML:

1. As a UML::Property, where the property (attribute) is contained in the class that is its domain; the value type for the property is its range. For sub-property relationships, use keyword notation, "subsets x," where x is the name of the property being subsetted (can subset multiple properties). In this case, the «rdfProperty» stereotype on the property itself is optional, although required if the property is intended to be used in an OWL Full ontology. Properties defined in this way may not be reusable, as each occurrence of the property, when contained in another class, will have a distinct UML identifier and will therefore be a different (unique) property definition.

2. As a UML::Association, where the property is represented by a binary association with uni-directional navigation from the domain to the range. The «rdfProperty» stereotype must be applied to the association; property name is the name of the role on the target (range) memberEnd of the association, and may optionally be used as the name of the association. For sub-property relationships, use keyword notation, "subsets x," where x is the name of the property being subsetted (can subset multiple properties), or a generalization arrow between the associations, with stereotype «subPropertyOf» on the generalization. The generalization approach is preferred. Properties defined in this way may not be reusable, as again, each occurrence of the property, when defined on another class, will have a distinct identifier in UML. See below for mitigation strategies.

3. As a UML::AssociationClass, where the property is represented by a binary association with uni-directional navigation from the domain to the range. The «rdfProperty» stereotype must be applied to the association class and may optionally be applied to the association (if the class is not elided in the diagram). The property name is the name of the association class and role on the target (range) memberEnd of the association, and may optionally be

used as the name of the association. For sub-property relationships, use keyword notation, “subsets x,” where x is the name of the property being subsetted (can subset multiple properties), or a generalization arrow between the associations, with stereotype «subPropertyOf» on the generalization. The generalization approach is preferred.

In order to ensure property reuse, if required in RDF and as often is expected in OWL, properties must be defined on anonymous classes, and related to one another and to the various classes that use them via mechanisms such as sub-class and sub-property relationships. See Figure 14.9 for an example taken from the Dublin Core vocabulary.

B. Properties without a specified domain are considered to be defined on an anonymous class, for example, as shown in Figure 14.7.

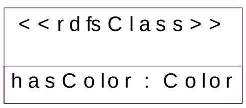

Figure 14.7 - Property `hasColor` Without Specified Domain

From a UML perspective, properties are semantically equivalent to binary associations with unidirectional navigation (“one-way” associations).

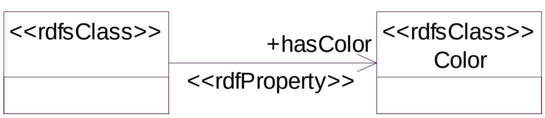

Figure 14.8 - Property `hasColor` Without Specified Domain, Alternate Representation

Figure 14.8 shows that there is efficient navigation from an instance of an anonymous class to an instance of `Color` through the `hasColor` end, just like a UML property. The only difference from a property on the anonymous class is that the underlying repository will have an association with the `hasColor` property as one of its ends. The other end will be owned by the association itself, and be marked as non-navigable.

RDF properties may be represented as AssociationClasses, as shown in Figure 14.9:

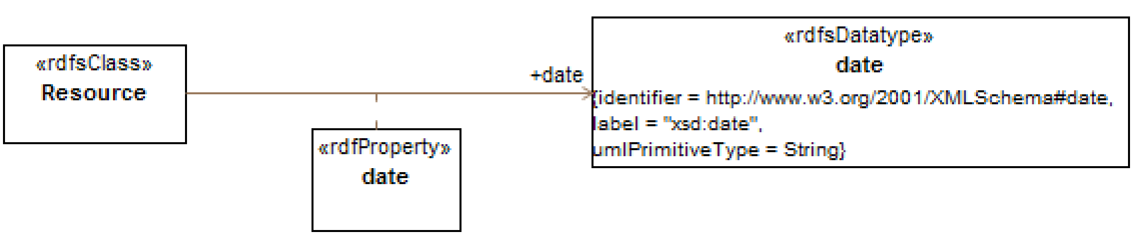

Figure 14.9 - Property `date` Without Specified Domain - Association Class Representation

An association class can have properties, associations, and participate in generalizations as any other class. The stereotype «rdfProperty» is introduced to highlight such binary, unidirectional association classes. The example shown in Figure 14.9 gives a basic definition for the Dublin Core *date* property. Resource is the top level class in RDF Schema, similar to `owl:Thing`, and the resulting RDF/XML, Turtle, N3, or other syntax rendered from such a model should not specify a domain. Alternatively, a blank, entirely unnamed class, as shown in Figure 14.8, would give the same result.

In the examples given in the remainder of the profile, the notation showing properties in association classs is typically used, but the other notation options could be used instead.

C. Properties *with* a domain are defined on a UML class for the domain, where the property is not inherited from a supertype.

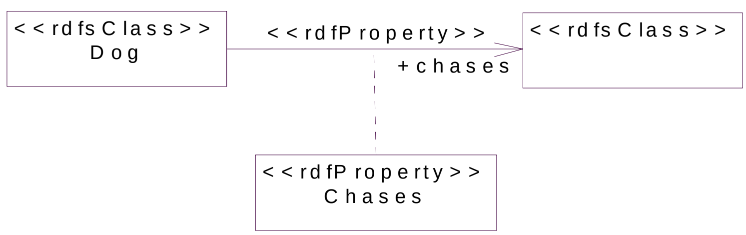

Figure 14.10 - Properties With Defined Domain, Undefined Range

Normally UML models introduce properties and restrict them with multiplicities in the same class. This translates to RDF/OWL as global properties of an anonymous class (or possibly to `owl:Thing` in OWL, and to restrictions on subclasses of `owl:Thing`). An optional stereotype «rdfGlobal» is introduced to highlight properties on the class where they are introduced, which will translate to global properties in OWL. Properties that are inherited are distinguished in UML by subsetting or redefinition, as discussed below.

D. Properties *with* a defined range have the range class as their type in UML. Properties with no range may have an anonymous class as their type in UML, as shown in Figure 14.10.

E. Properties with a range have the range class as their type in UML. Property types are shown to the right of the colon after the property name, as shown in Figure 14.7.

F. The notation for RDF/S and OWL property subtyping (i.e., `rdfs:subPropertyOf`) uses UML property/ unidirectional association subsetting, association class subtyping, or generalization/specialization.

One option for property subsetting in UML is to use “{subsets <super-property-name>}” at the end of the property entry in a class, as shown in Figure 14.11.

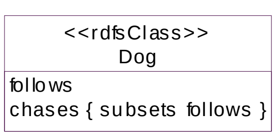

Figure 14.11 - Property Subsetting, Notation on Property Entry for Class

Alternatively, the representation given in Figure 14.12 may be used for unidirectional association subsetting.

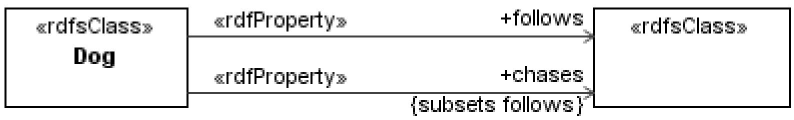

Figure 14.12 - Property Subsetting, Unidirectional Association Representation

For use with association classes, the representation shown in Figure 14.13, which uses a UML Generalization with the stereotype «rdfsSubPropertyOf», is preferred. Note that «rdfsSubPropertyOf» may not be required – it does not change the semantics, only adds constraints on its own usage.

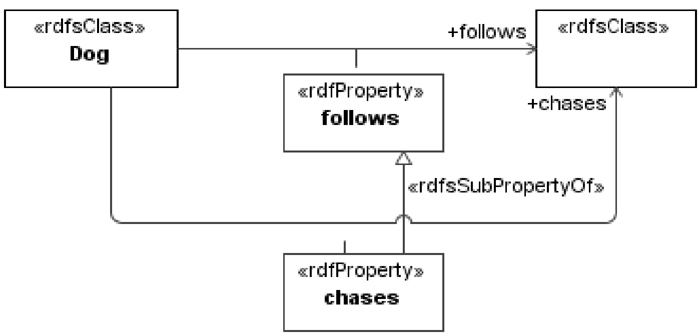

Figure 14.13 - Property Subsetting, Association Class Representation

14.2.7.2 surrogateProperty

Description

A surrogate property is a "stand-in" for an RDF property defined in any of the three ways described in 14.2.7.1. Surrogates are intended for use on diagrams where one wants to show relationships between properties without "dragging all of the related baggage" (domain and range classes, for example) onto those diagrams. Examples where surrogates are useful include property hierarchies in RDF or OWL, and complex restrictions in OWL.

Stereotype and Base Class

«surrogateProperty» with base class of UML::Class.

Parent

«namedResource».

Properties

None.

Constraints

[1] A surrogate property can only be defined for an existing UML property, association, or association class representing the RDF (or OWL) base property (meaning, that the surrogate must have a corresponding "real" property, either locally or in an imported model, in order to be considered well-formed).

[2] A dependency stereotyped by «surrogateFor», linking the surrogate to its base property definition is required.

Graphical Notation

Figure 14.14 provides an example of definition of a surrogate for the Dublin Core Metadata Terms date property, extending the definition given in Figure 14.9.

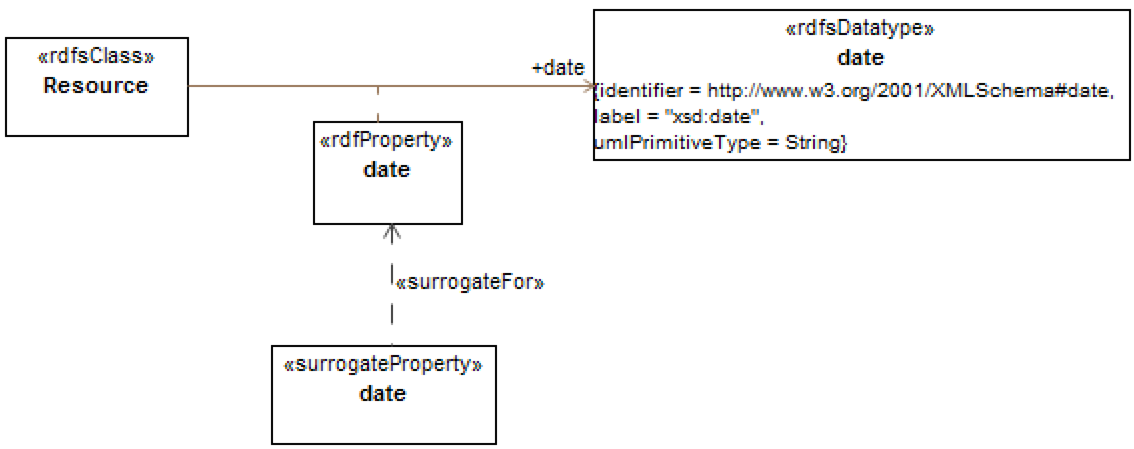

Figure 14.14 - Surrogate Property definition

The surrogate property can be used in sub-property definitions, shown in Figure 14.15.

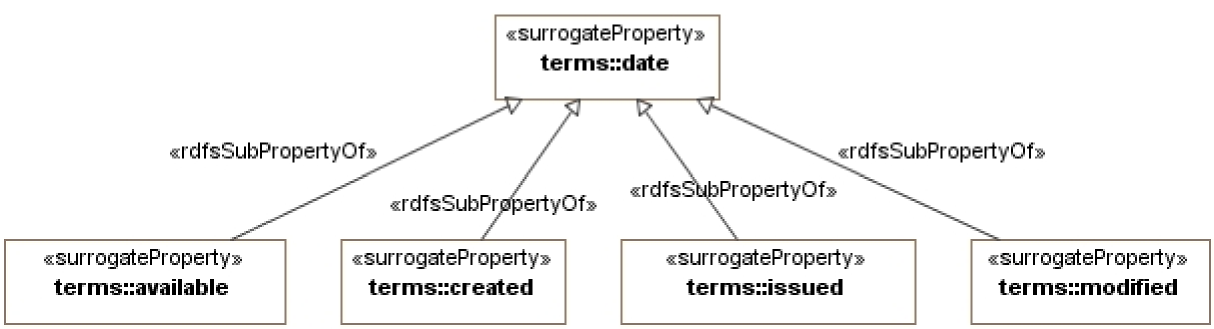

Figure 14.15 - Surrogate property usage example – property hierarchy

14.2.7.3 surrogateFor

Description

This dependency links a surrogate property to the UML element that defines the property for which it is a surrogate.

Stereotype and Base Class

«surrogateFor» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

[1] The «surrogateFor» stereotype can only be applied to a UML::Dependency that links a UML::Class stereotyped by «surrogateProperty» (or «surrogateObjectProperty», or «surrogateDatatypeProperty», or their subtypes in OWL) and a UML::Property, UML::Association, or UML::AssociationClass stereotyped by «rdfProperty» (or the equivalent OWL property). The dependency must be navigable from the surrogate to the base property it stands in for.

14.2.7.4 RDF Global Property

Description

An optional stereotype on a unidirectional association class, association, or property with «rdfProperty» applied, indicating the property is defined globally, i.e., that the class having the property or non-navigable end of the association, is the class on which the class property is introduced, meaning, the property is not inherited from a superclass.

Stereotype and Base Class

«rdfGlobal» with base class of UML::AssociationClass, UML::Property, and UML::Association.

Parent

«rdfProperty».

Properties

None.

Constraints

[1] The property being stereotyped must be on an anonymous class (or possibly on an instance of `owl:Thing` in OWL), or on an association class for a unidirectional association that has an anonymous class on the non-navigable end.

14.2.7.5 Domain Property

Description

`rdfs:domain` indicates that RDF resources denoted by the subjects of triples whose predicate is a given property P are instances of the class denoted by the domain. When a property has multiple domains, then the resources denoted by the subjects of triples whose predicate is P are instances of all of the classes stated by the `rdfs:domain` properties (In OWL, this is typically thought of as an anonymous class representing the intersection of all of the classes said to be in the domain of the property).

Stereotype and Base Class

«domain» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

[1] Applies only to dependencies between classes stereotyped by «rdfsClass» (or any of their children, e.g., «owlClass»), and classes stereotyped by «surrogateProperty» (or the equivalent OWL stereotype).

[2] Dependencies with «domain» applied have as their client a class stereotyped by «surrogateProperty» and have as their supplier a class stereotyped by «rdfsClass».

Graphical Notation

Figure 14.16 - «domain» stereotype usage example

14.2.7.6 Range Property

Description

`rdfs:range` indicates that the values of a given property P are instances of the class denoted by the range. When a property has multiple `rdfs:range` properties, then the resources denoted by the objects of triples whose predicate is P are instances of all of the classes stated by the `rdfs:range` properties. (In OWL, this is typically thought of as an anonymous class representing the intersection of all of the range classes.)

Stereotype and Base Class

«range» with base class of UML::Dependency.

Parent

None.

Properties

None.

Constraints

[1] Applies only to dependencies between classes stereotyped by «rdfsClass» or «rdfsDatatype» (or any of their children, e.g., «owlClass»), and classes stereotyped by «surrogateProperty» (or the equivalent OWL stereotype).

[2] Dependencies with «range» applied have as their client a class stereotyped by «surrogateProperty» and have as their supplier a class stereotyped by «rdfsClass» or «rdfsDatatype».

Graphical Notation

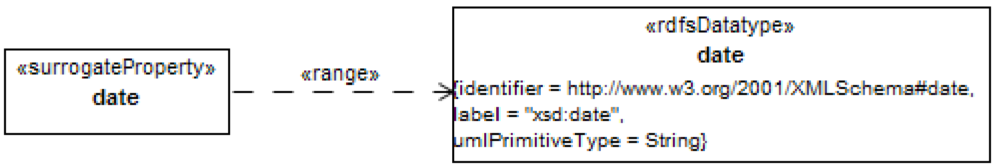

Figure 14.17 - «range» Stereotype Notation

14.2.7.7 RDF subPropertyOf Property

Description

`rdfs:subPropertyOf` is used to specialize RDF properties, similar to class generalization/specialization, and indicates that all the instances of the extension of the subproperty are instances of the extension of the super property. See above for further discussion and representation options, if used.

Stereotype and Base Class

«subPropertyOf» with base class of UML::Generalization.

Parent

None.

Properties

None.

Constraints

[1] Association classes on both ends of the generalization must be stereotyped «rdfProperty» or «surrogateProperty», or «objectProperty», «datatypeProperty», «surrogateObjectProperty», or «surrogateDatatypeProperty» if used with the profile for OWL.

[2] In OWL DL, an association class stereotyped by «objectProperty» or «datatypeProperty» (or their surrogates) may specialize a class stereotyped by «rdfProperty» (or its surrogate), but not vice versa.

14.2.8 Containers and Collections

Most of the stereotypes associated with the definitions given in 10.7 are defined in Annex A (“Foundation Library (M1) for RDF and OWL”) including lists.

14.2.8.1 Container Membership Property

Description

`rdfs:ContainerMembershipProperty` provides a mechanism for defining ordered properties and linking them to a specified container in RDF. Instances of this property include `rdf:_1, rdf:_2,rdf:_3 ...`, and are used to state that the object of the property is member of the container specified as the subject (see 10.7.5).

Stereotype and Base Class

«containerMembershipProperty» with base class of UML::Property, UML::Association.

Parent

«rdfProperty».

Properties

None.

Constraints

None.

14.2.9 Labeled and Named Resources

A number of first class elements in RDF may be labeled and identified via an IRI. In contrast with the UML "name" property on an element, there may be multiple labels in any language associated with the same model element, including labels that have blanks or special characters in them, which might not be allowed as the UML element name. Any first class element in RDF and OWL may (and often must), also have an IRI (identifier), which may or may not be in the same namespace as the document (or RDF source). These first class elements include any element that may be referenced from a node in a graph that is not a blank node or literal. In order to ensure that the same properties for labels and names (identifiers) are used for all stereotypes that represent such resources, two additional abstract stereotypes are introduced: «labeledResource» and «namedResource». These stereotypes and their children are shown in Figure 14.18.

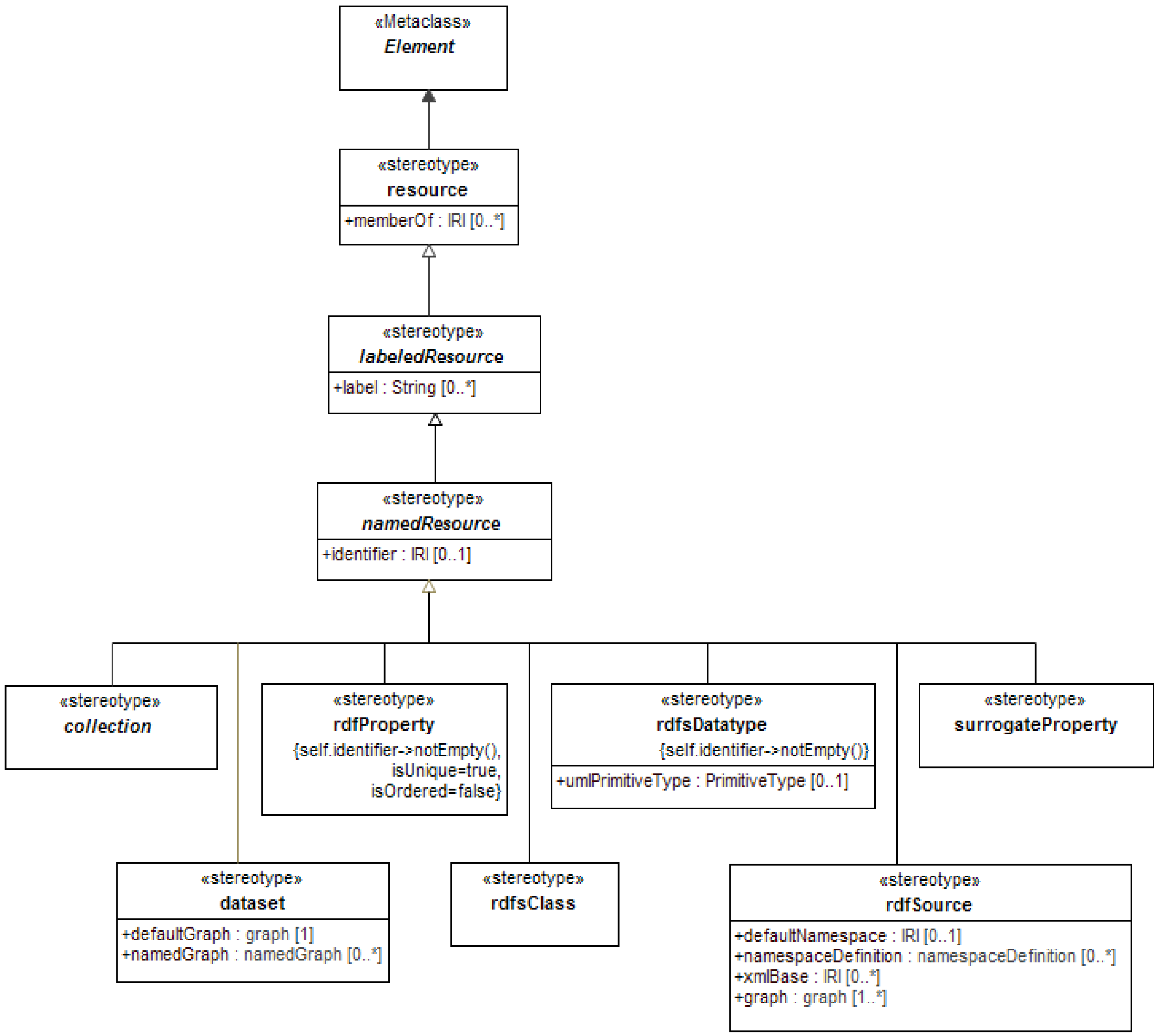

Figure 14.18 Labeled and Named Resources

14.2.9.1 Labeled Resources

Description

Any resource in an RDF graph can be labeled using `rdfs:label`. This stereotype provides the facility to optionally add a label, *using the same unique property*, to any appropriate element using the RDF profile. This optional label stereotype is an alternative to using the «label» stereotyped dependency together with a literal string or instance specification stereotyped by «literal» for the same purpose. Note that this "shortcut" label tagged value option does not include support for multi-lingual labels, which the approach using the «label» dependency with a «literal» provides, unless the language is encoded as a part of the label itself.

Annotation properties (e.g., `skos:prefLabel`) can be used to create multiple, more complex and/or multi-lingual labels for any resource in a model.

Stereotype and Base Class

«labeledResource», which is an abstract stereotype, with base classes of UML::Element.

Parent

«resource».

Properties

- label: String [0..*] - zero or more strings representing the label(s).

Constraints

None.

14.2.9.2 Named Resources

Description

Any node in an RDF graph can be identified via an IRI. This stereotype provides the facility to optionally add an IRI, using the same property, to any appropriate element using the RDF profile.

This capability is especially important in cases where the referenced node is part of an external ontology, for example, for use in referencing but not importing elements in other ontologies in OWL as the range of some property, or in extending classes in imported or referenced vocabularies and ontologies without creating local copies of those classes. In other words, the facility is needed in any scenario where the IRI for the element cannot be computed automatically from the context of the vocabulary or ontology. The identifier is optional if the element is part of the current vocabulary or ontology, or part of an imported or referenced model that is available in the UML model context, but not otherwise.

Stereotype and Base Class

«namedResource», which is an abstract stereotype, with base classes of UML::Element.

Parent

«labeledResource».

Properties

- identifier: IRI [0..1] - an optional identifier for the resource.

Constraints

[1] The element representing the value for the identifier must be stereotyped by «IRI».

[2] The *specification* for the IRI, in the case of an InstanceSpecification, or *value*, in the case of a LiteralString, if present, must conform to the character encoding (including escape sequences, as necessary) defined in RFC 3987 and be stereotyped by «IRI».

14.3 UML Profile for OWL

This sub clause specifies the UML profile for OWL. It is organized loosely on the structure of the OWL metamodel, with sections reordered to facilitate understanding and utility.

14.3.1 OWL Profile Package

The following sub clauses specify the set of stereotypes, stereotype properties, and other elements that comprise the UML2 profile for OWL. As shown in Figure 14.19, the OWL profile package provides the container for the profile and imports the «rdf» profile.

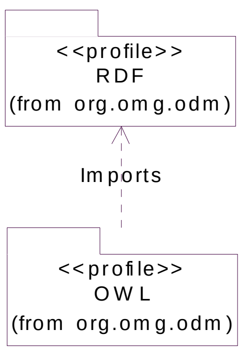

Figure 14.19 - Web Ontology Language (OWL) Profile Package

14.3.2 OWL Ontology

Description

An OWL ontology consists of zero or more optional ontology headers (typically at most one), which may include a set of ontology properties, such as `owl:Imports` statements, plus any number of class axioms, property axioms, and facts about individuals.

In a UML representation, we capture some of the header constructs by specializing «rdfDocument». Others are specified as ontology and annotation properties, defined below.

Stereotype and Base Class

«owlOntology» with base class of UML::Package.

Parent

«rdfDocument».

Properties

None.

Constraints

[1] All classes (except association classes) in a package stereotyped by «owlOntology» must be stereotyped by «rdfsClass» or by «owlClass» (or an appropriate subclass).

[2] For applications intending to support OWL DL, all classes (except association classes) in a package stereotyped by «owlOntology» must be stereotyped by «owlClass» (or one of its subclasses).

14.3.3 OWL Annotation Properties

OWL annotation properties correspond, for the most part, to other stereotype properties defined in RDF or in this profile, although users may define their own.

14.3.3.1 Annotation Property

Description

«annotationProperty» represents the class of user-defined annotation properties in OWL (corresponding roughly to 11.4.1). OWL annotations can be applied to any ontology element (e.g., ontologies, classes, properties, individuals).

Stereotype and Base Class

«annotationProperty» with base class of UML::AssociationClass, UML::Property, and UML::Association.

Parent

«rdfProperty».

Properties

None.

Constraints

None.

14.3.3.2 owl:versionInfo

Description

An `owl:versionInfo` property generally has a string that provides information about the version of the element to which it is applied, for example RCS/CVS keywords. It does not contribute to the logical meaning of the ontology other than that given by the RDF(S) model theory.

Although typically used to make statements about ontologies, it may be applied to instance of any OWL construct. For example, one could apply an `owl:versionInfo` property to a class stereotyped by «owlClass», or to an instance of the RDFStatement class.

Stereotype and Base Class

No stereotype; implemented as a UML Property of the stereotype or model library class it describes.

- versionInfo: String [0..*] – the string containing the version information.

Parent

None.

Properties

None.

Constraints

None.

Graphical Representation

In the case of an ontology, with a package stereotyped by «owlOntology» or «rdfDocument», the normal stereotype notation can be used, with property values specified in braces under the stereotype label[4], as shown in Figure 14.20.

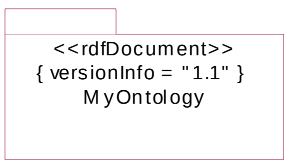

Figure 14.20 - Representation for `versionInfo` Applied to an Ontology or RDF Document

14.3.4 OWL Ontology Properties

OWL ontology properties are similar to annotation properties, in that they support annotations on OWL ontologies. The «owlOntologyProperty» stereotype can be applied to a property on a package stereotyped by «owlOntology» or «rdfDocument», and should be typed by another package that is similarly stereotyped.

OWL provides several built-in ontology properties, and also allows users to define such properties. Users can use some discretion in defining ontology properties, using either UML::Property or UML::Constraint as a base class, as appropriate.

14.3.4.1 owl:OntologyProperty

Description

`owl:OntologyProperty` represents the class of ontology properties in OWL, both built-in and user defined, corresponding to 11.4.4.

User-defined ontology properties are properties defined on the «owlOntology» or «rdfDocument» stereotypes, that can apply only between packages having these stereotypes.

Stereotype and Base Class

«owlOntologyProperty» with base class of UML::Property and UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies only to properties of «owlOntology» or «rdfDocument».

4. The stereotype property follows the clarifications and elaborations of stereotype notation defined in Unified Modeling Language: (UML) Superstructure, Profiles clause, http://www.omg.org/spec/UML..

[2] Types of properties stereotyped by «owlOntologyProperty» must be stereotyped by «owlOntology» or «rdfDocument».

14.3.4.2 owl:backwardCompatibleWith

Description

`owl:backwardCompatibleWith` refers to another ontology, and identifies the specified ontology as a prior version, and further indicates that it is backward compatible with it.

Stereotype and Base Class

«backwardCompatibleWith» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between packages stereotyped by «owlOntology» or «rdfDocument».

[2] Classes and properties in the new version that have the same name as classes and properties in the earlier version must either be equivalent to or extend those in the earlier versions.

[3] The later version must be logically consistent with the earlier version.

[4] (Semantic) Identifiers in the later version have the same interpretation in the earlier version.

Graphical Representation

Dashed line between two instances with stereotype label, arrowhead towards the earlier version, as shown in Figure 14.21.

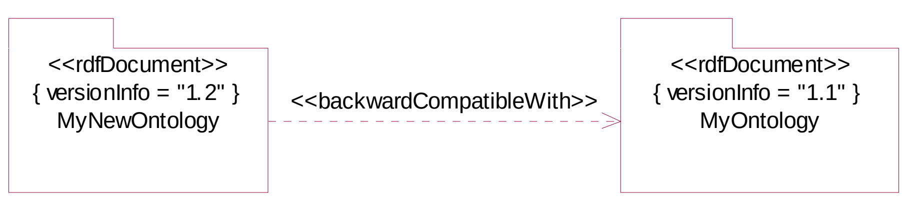

Figure 14.21 - Stereotype Representation for `owl:backwardCompatibleWith`

14.3.4.3 owl:imports

Description

`owl:imports` references another OWL ontology containing definitions, whose meaning is considered to be part of the meaning of the importing ontology. Each reference consists of an IRI specifying from where the ontology is to be imported.

Stereotype and Base Class

«owlImports» with base class of UML::PackageImports.

Parent

None.

Properties

None.

Constraints

[1] Applies only to imports between packages stereotyped by «owlOntology» or «rdfDocument».

Graphical Representation

Dashed line between two instances with stereotype label, arrowhead towards the imported ontology, as shown in Figure 14.22.

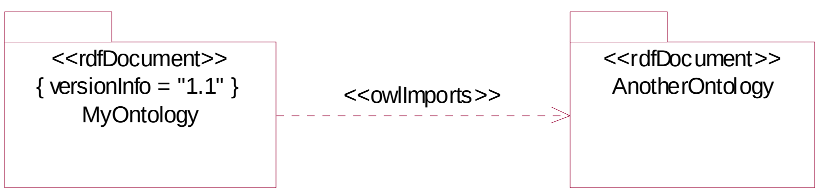

Figure 14.22 - Stereotype Representation for owl:imports

14.3.4.4 owl:incompatibleWith

Description

`owl:incompatibleWith` indicates that the containing ontology is a later version of the referenced ontology, but is not necessarily backward compatible with it. Essentially, this allows ontology authors to specify that a document cannot be upgraded without verifying consistency with the specified ontology.

Stereotype and Base Class

«incompatibleWith» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between packages stereotyped by «owlOntology» or «rdfDocument».

Graphical Representation

Dashed line between two instances with stereotype label, arrowhead towards the earlier version, as shown in Figure 14.23.

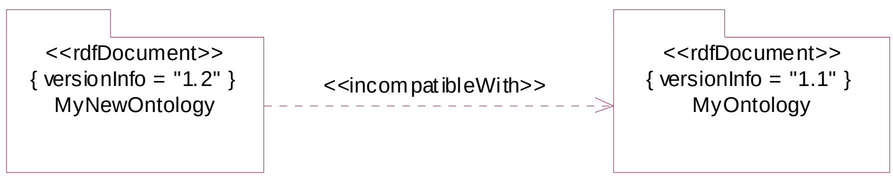

Figure 14.23 - Stereotype Representation for owl:incompatibleWith

Note: While it might seem reasonable to eliminate the arrowhead in this case, and make the relationship bi-directional, all RDF graphs and thus such relationships are unidirectional in RDF, RDF Schema, and OWL. Applications that leverage this representation may optionally allow the user to indicate that they want a particular instance of «incompatibleWith» to be bidirectional, eliminate the arrowhead, and use a single dashed line; the interpretation of such notation should be two instances of «incompatibleWith», however.

14.3.4.5 owl:priorVersion

Description

`owl:priorVersion` identifies the specified ontology as a prior version of the containing ontology. This has no meaning in the model-theoretic semantics other than that given by the RDF(S) model theory. However, it may be used by software to organize ontologies by versions.

Because of the lack of semantics, there is no obvious UML element to reuse or stereotype for this particular OWL property. However, assuming that the spirit of this property is similar to though not quite as strong as that of «backwardCompatibleWith», the same base class is used.

Stereotype and Base Class

«priorVersion» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between packages stereotyped by «owlOntology» or «rdfDocument».

Graphical Representation

Dashed line between two instances with stereotype label, arrowhead towards the earlier version, as shown in Figure 14.24.

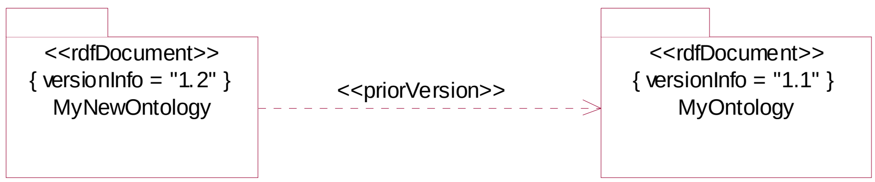

Figure 14.24 - Stereotype Representation for owl:priorVersion

14.3.5 Class Expressions

Classes provide an abstraction mechanism for identifying the common characteristics among a group of resources. Like RDF classes, every OWL class is associated with a set of individuals, called the class extension. The individuals in the class extension are called the instances of the class. A class has an intensional meaning (the underlying concept) that is related but not equal to its class extension. Thus, two classes may have the same class extension, but still be different classes, (e.g., classes representing “the morning star” and “the evening star”).

A class description is the term used in [OWL S&AS] for the basic building blocks of class axioms. A class description describes an OWL class, either by a class name or by specifying the class extension of an unnamed anonymous class.

OWL distinguishes six types of class descriptions:

1. a class identifier (an IRI)
2. an exhaustive enumeration of individuals that together form the instances of a class
3. a property restriction
4. the intersection of two or more class descriptions
5. the union of two or more class descriptions
6. the complement of a class description

The first type is special in the sense that it describes a class through a class name (syntactically represented as an IRI). The other five types of class descriptions typically describe an anonymous class by placing constraints on the class extension. They consist of a set of RDF triples in which a blank node represents the class being described. This blank node has a type property whose value is «owlClass». Note that multiple class descriptions can be applied to the same class, however, such that these anonymous classes can ultimately also be named.

Class descriptions of type 2-6 describe, respectively, a class that contains exactly the enumerated individuals (2nd type), a class of all individuals that satisfy a particular property restriction (3rd type), or a class that satisfies boolean combinations of class descriptions (4th, 5th, and 6th type). Intersection, union, and complement can be respectively seen as the logical AND, OR, and NOT operators. The four latter types of class descriptions lead to nested class descriptions and can thus in theory lead to arbitrarily complex class descriptions. In practice, the level of nesting is usually limited. Stereotypes for OWL class descriptions are given below.

14.3.5.1 OWLClass

Description

`owl:Class` describes a class through a class name, and corresponds to 11.3.1.

Note: `owl:Class` is defined as a subclass of `rdfs:Class`. The rationale for having a separate OWL class construct lies in the restrictions on OWL DL (and thus also on OWL Lite), which imply that not all RDFS classes are legal OWL DL classes. In OWL Full these restrictions do not exist and therefore `owl:Class` and `rdfs:Class` are equivalent in OWL Full.

Stereotype and Base Class

«owlClass» with base class of UML::Class

Parent

«rdfsClass»

Properties

- isDeprecated: Boolean [0..1] – provides an additional annotation that indicates a particular class definition is deprecated.

Constraints

None.

14.3.5.2 EnumeratedClass

Description

«enumeratedClass» describes a class by enumerating the set of individuals, or UML instance specifications, that are members of the class, and corresponds to 11.3.3.

Stereotype and Base Class

«enumeratedClass» with base class of UML::Class.

Parent

«owlClass»

Properties

- isComplete: Boolean [0..1] – indicates whether the set of enumerated individuals is complete, meaning, that this provides a complete specification for the class.

Constraints

None.

14.3.5.3 Property Restrictions

Description

Class expressions can be formed by restricting the possible values of object or data properties in OWL. Such class expressions fall into two broad categories: (1) number or cardinality restrictions, which restrict the number of values possible for a particular property, and (2) value restrictions, which provide the means for expressing universal and existential quantification over property expressions, stating specific value restrictions, and defining self-reflective property expressions.

In this sub clause, the set of stereotypes needed for the restriction expressions available in OWL 2 are provided. These include a stereotyped UML class to represent the restriction class expression, as stated above, together with a number of stereotyped dependencies to further qualify the restriction. Extensive examples are provided to clarify their intended use, including the corresponding RDF/XML serialized OWL. While some restrictions can be quite simple in nature, similar to UML multiplicity expressions, the majority are typically more complex in practice. Patterns are built up using restrictions together with other kinds of class expressions in many cases. Examples of increasingly complex patterns are provided to assist implementers in understanding how such expressions can be built up using ODM stereotypes as well.

Stereotypes and Base Classes

«owlRestriction» with base class of UML::Class

«allValuesFrom», «hasValue», «onClass», «onDataRange», «onDatatype», «onProperties», «onProperty», and «someValuesFrom» with base class of UML::Dependency.

Parent

«owlClass» is the parent of «owlRestriction»

Properties

The following are properties on «owlRestriction»:

- cardinality: UnlimitedNatural [0..1] - specifies exact cardinality for the property to which the restriction applies.
- hasSelf: Boolean [0..1] - specifies local reflexivity in terms of a restriction class for a particular property, for example, a class of self-regulating processes, that relate members of a class expression to itself.
- maxCardinality: UnlimitedNatural [0..1] - specifies a ceiling on the cardinality (meaning, at most n) for the property to which the restriction applies.
- minCardinality: UnlimitedNatural [0..1] - specifies a minimum on the cardinality (meaning, at least n) for the property to which the restriction applies.

Constraints

[1] (Semantic) Instances of the restriction class in a given class expression are all and only those instances satisfying the restriction.

Graphical Notation for Cardinality Restrictions

In OWL, like in RDF, it is assumed that any instance of a class may have an arbitrary number (zero or more) of values for a particular property. To make a property required (at least one), to allow only a specific number of values for that property, or to insist that a property must not occur, cardinality restrictions can be used. OWL provides several constructs for restricting the cardinality of properties locally within a class context.

These constructs can be used to create class expressions consisting of values for the property that conform to the restriction, namely at least, at most, or exactly the number of individuals or literal values for that property.

Figure 14.25 provides a simple example, creating a restriction class of individuals that have exactly one color and a second restriction class of individuals that have more than one color. Note that the relationship between the named class SingleColoredThing and the restriction class is modeled as an equivalence relationship, meaning that membership in the restriction class is a necessary and sufficient condition for being a SingleColoredThing.

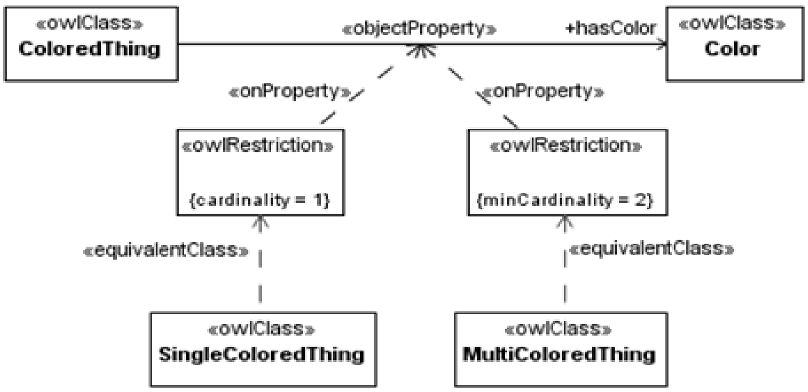

Figure 14.25 - Cardinality restrictions with equivalence relation using Association notation for the property

The RDF/XML serialized OWL corresponding to Figure 14.25 is:

```
<owl:ObjectProperty rdf:about="&re;hasColor">
    <rdfs:range rdf:resource="&re;Color"/>
    <rdfs:domain rdf:resource="&re;ColoredThing"/>
</owl:ObjectProperty>

<owl:Class rdf:about="&re;Color"/>
<owl:Class rdf:about="&re;ColoredThing"/>

<owl:Class rdf:about="&re;MultiColoredThing">
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasColor"/>
            <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">2</owl:minCardinality>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="&re;SingleColoredThing">
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasColor"/>
            <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

Figure 14.26 depicts a common pattern in OWL used to indicate that in order for an individual to be considered a member of a particular class, it must have at least one value for certain properties of that class.

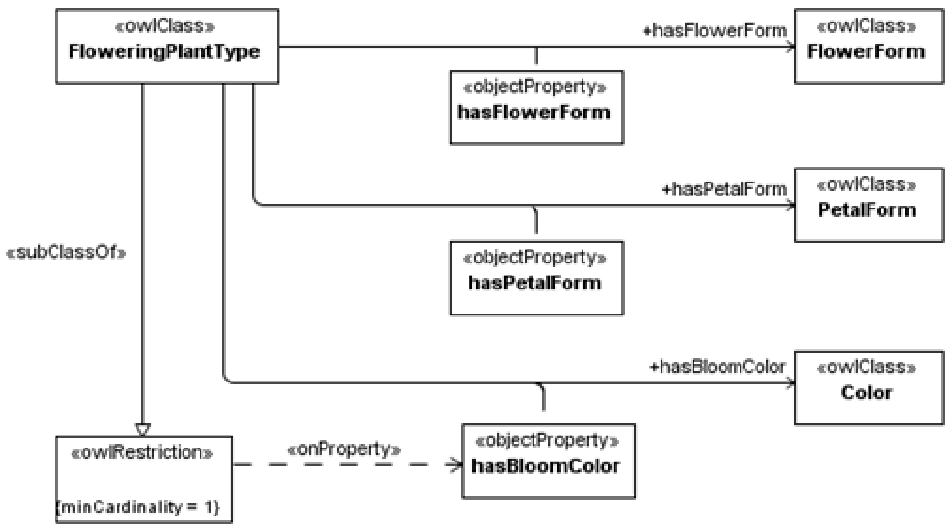

Figure 14.26 - Property restriction for owl:minCardinality with subclass relation using AssociationClass notation for the property

The RDF/XML serialized OWL corresponding to Figure 14.26 is:

```
<owl:ObjectProperty rdf:about="&re;hasBloomColor">
    <rdfs:range rdf:resource="&re;Color"/>
    <rdfs:domain rdf:resource="&re;FloweringPlantType"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="&re;hasFlowerForm">
    <rdfs:range rdf:resource="&re;FlowerForm"/>
    <rdfs:domain rdf:resource="&re;FloweringPlantType"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="&re;hasPetalForm">
    <rdfs:domain rdf:resource="&re;FloweringPlantType"/>
    <rdfs:range rdf:resource="&re;PetalForm"/>
</owl:ObjectProperty>

<owl:Class rdf:about="&re;Color"/>
<owl:Class rdf:about="&re;FlowerForm"/>
<owl:Class rdf:about="&re;PetalForm"/>

<owl:Class rdf:about="&re;FloweringPlantType">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

In addition to basic cardinality restrictions, OWL 2 provides the capability to further qualify cardinality by specific classes and data ranges, for restrictions on object and data properties respectively. Suppose, for example, that rather than explicitly creating a “has<Characteristic>” property for every feature of flowering plants that is important to the use case for this ontology, one were to instead create a higher level “has characteristic” property that might be reused multiple times. The alternative modeling pattern required to stipulate that every flowering plant type must have at least one bloom color is given in Figure 14.27.

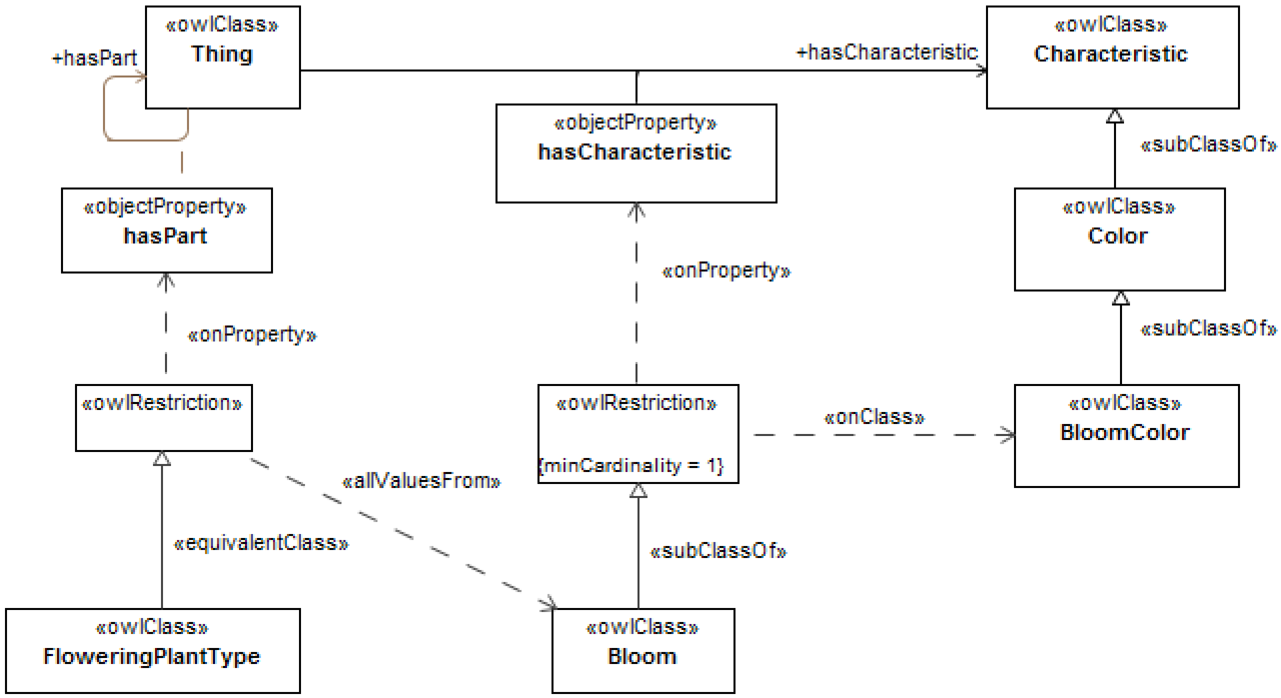

Figure 14.27 - Property restriction for owl:minQualifiedCardinality with subclass relation using AssociationClass notation for the property

The RDF/XML serialized OWL corresponding to Figure 14.27 is:

```
<owl:ObjectProperty rdf:about="&re;hasCharacteristic">
    <rdfs:range rdf:resource="&re;Characteristic"/>
</owl:ObjectProperty>

<owl:Class rdf:about="&re;BloomColor">
    <rdfs:subClassOf rdf:resource="&re;Characteristic"/>
    <rdfs:subClassOf rdf:resource="&re;Color"/>
</owl:Class>

<owl:Class rdf:about="&re;Characteristic"/>

<owl:Class rdf:about="&re;Color"/>

<owl:Class rdf:about="&re;FloweringPlantType">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasCharacteristic"/>
            <owl:onClass rdf:resource="&re;BloomColor"/>
            <owl:minQualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minQualified-
Cardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Graphical Representation for Universal Quantification (all values) Restrictions

The `owl:allValuesFrom` class expression is a built-in OWL construct used to support universal quantification. In other words, a restriction containing `owl:allValuesFrom` specifies a class or data range for which all values of the property under consideration are either members of the described class, or are data values within the specified data range.

Several representation approaches are provided here, in keeping with the representation used for properties in the profile for RDF/S. First, we can show the same thing using unidirectional association style properties, as in Figure 14.28.

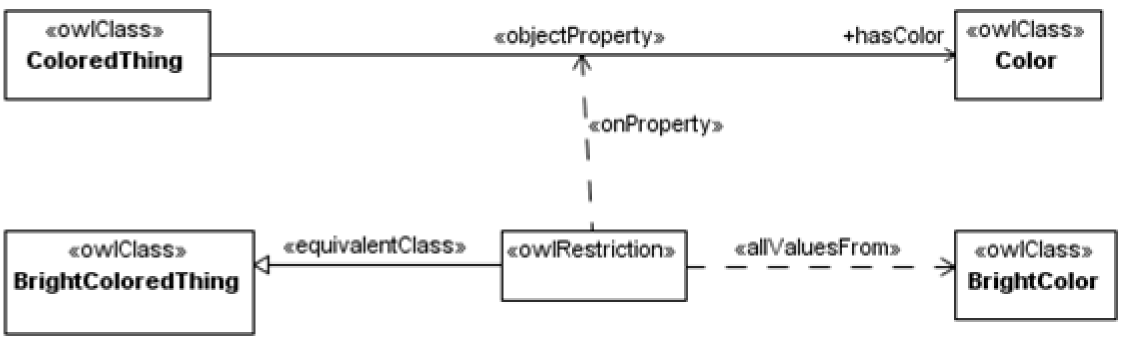

Figure 14.28 - Property restriction for owl:allValuesFrom with equivalence relation using unidirectional Association notation for the property

The RDF/XML serialized OWL corresponding to Figure 14.28 is:

```
<owl:ObjectProperty rdf:about="&re;hasColor">
    <rdfs:range rdf:resource="&re;Color"/>
    <rdfs:domain rdf:resource="&re;ColoredThing"/>
</owl:ObjectProperty>

<owl:Class rdf:about="&re;BrightColor"/>
<owl:Class rdf:about="&re;Color"/>
<owl:Class rdf:about="&re;ColoredThing"/>

<owl:Class rdf:about="&re;BrightColoredThing">
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasColor"/>
            <owl:allValuesFrom rdf:resource="&re;BrightColor"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

In Figure 14.28, the restriction is explicitly defined as being logically equivalent to the named class BrightColoredThing. Figure 14.29 depicts a similar pattern, used here to define the set of flowering plants whose color must be a member of the set of red colors specified by the Royal Horticultural Society's classification scheme.

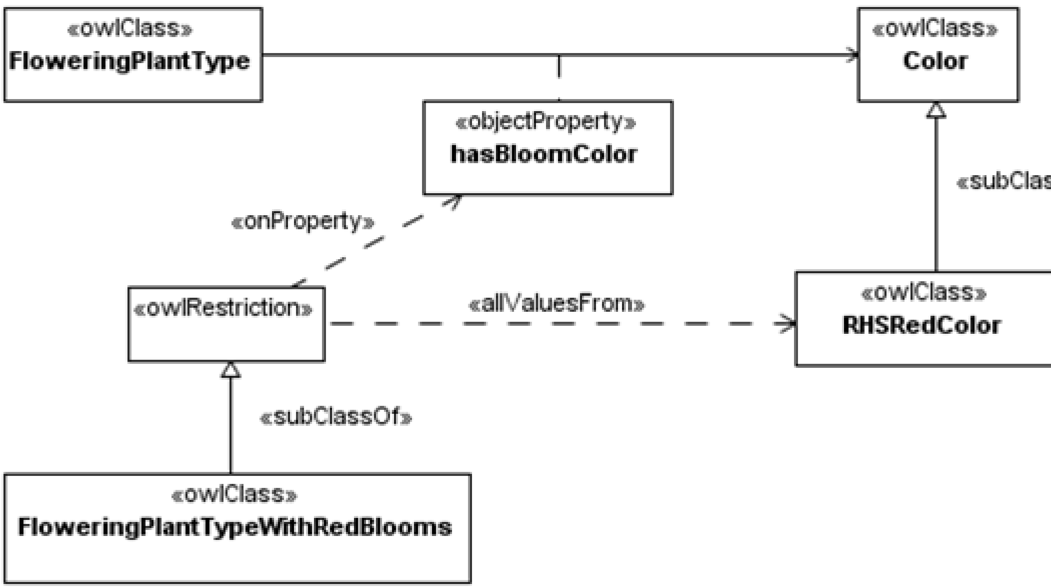

Figure 14.29 - Property restriction for owl:allValuesFrom with subclass relation using AssociationClass notation for the property

The RDF/XML serialized OWL corresponding to Figure 14.29 is:

```
<owl:ObjectProperty rdf:about="&re;hasBloomColor">
    <rdfs:range rdf:resource="&re;Color"/>
    <rdfs:domain rdf:resource="&re;FloweringPlantType"/>
</owl:ObjectProperty>

<owl:Class rdf:about="&re;Color"/>

<owl:Class rdf:about="&re;FloweringPlantType">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&re;RHSRedColor">
    <rdfs:subClassOf rdf:resource="&re;Color"/>
</owl:Class>

<owl:Class rdf:about="&re;FloweringPlantTypeWithRedBlooms">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:allValuesFrom rdf:resource="&re;RHSRedColor"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

The examples given so far show single restrictions whose values are specified by simple named classes. In practice, ontologies frequently include complex class expressions defined using multiple restrictions together with other constructs. For example, we could replace the single source for color definitions with a broader set, as shown in Figure 14.30.

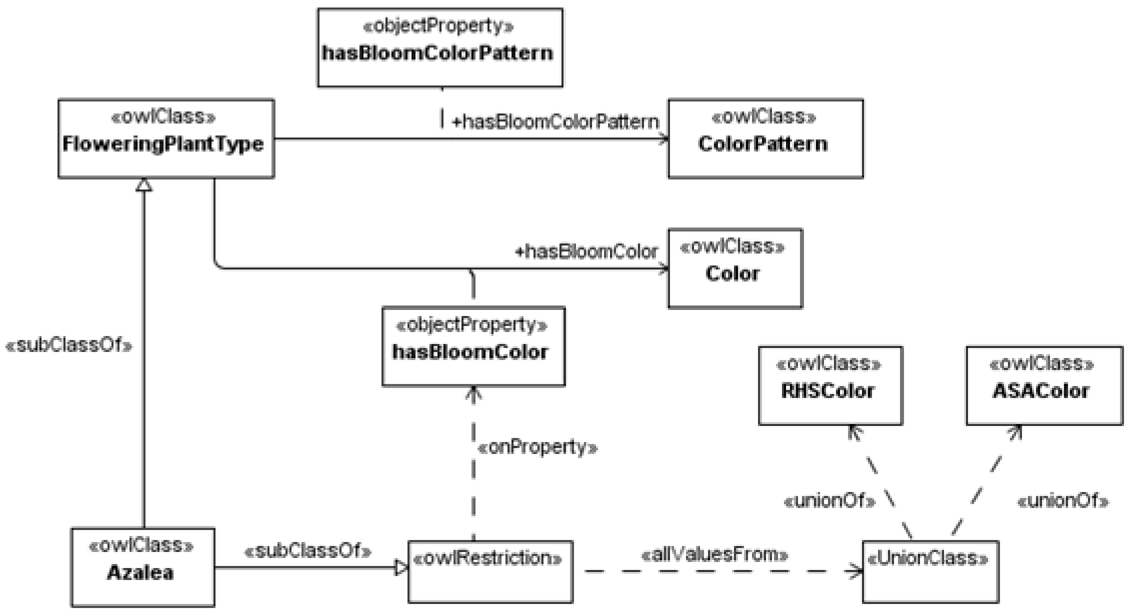

Figure 14.30 Property restriction for owl:allValuesFrom with subclass relation using a class expression as the target of the restriction

The RDF/XML serialized OWL corresponding to Figure 14.30 is:

```
<owl:ObjectProperty rdf:about="&re;hasBloomColor">
    <rdfs:range rdf:resource="&re;Color"/>
    <rdfs:domain rdf:resource="&re;FloweringPlantType"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="&re;hasBloomColorPattern">
    <rdfs:range rdf:resource="&re;ColorPattern"/>
    <rdfs:domain rdf:resource="&re;FloweringPlantType "/>
</owl:ObjectProperty>

<owl:Class rdf:about="&re;ASAColor"/>
<owl:Class rdf:about="&re;Color"/>
<owl:Class rdf:about="&re;ColorPattern"/>

<owl:Class rdf:about="&re;FloweringPlantType">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&re;RHSColor"/>
```

```
<owl:Class rdf:about="&re;Azalea">
    <rdfs:subClassOf rdf:resource="&re;FloweringPlantType"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:allValuesFrom>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
                        <rdf:Description rdf:about="&re;ASAColor"/>
                        <rdf:Description rdf:about="&re;RHSColor"/>
                    </owl:unionOf>
                </owl:Class>
            </owl:allValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

In this case, the colors available to describe azalea blooms can only come those defined by the Royal Horticultural Society or those defined by the Azalea Society of America.

Another typical OWL pattern using equivalence relations together with class expressions and restrictions to define given class is shown in Figure 14.31. This example more precisely specifies the definition of the class RHSFloweringPlantType than the example in Figure 14.30, and can be useful in cases where precise classification is desired.

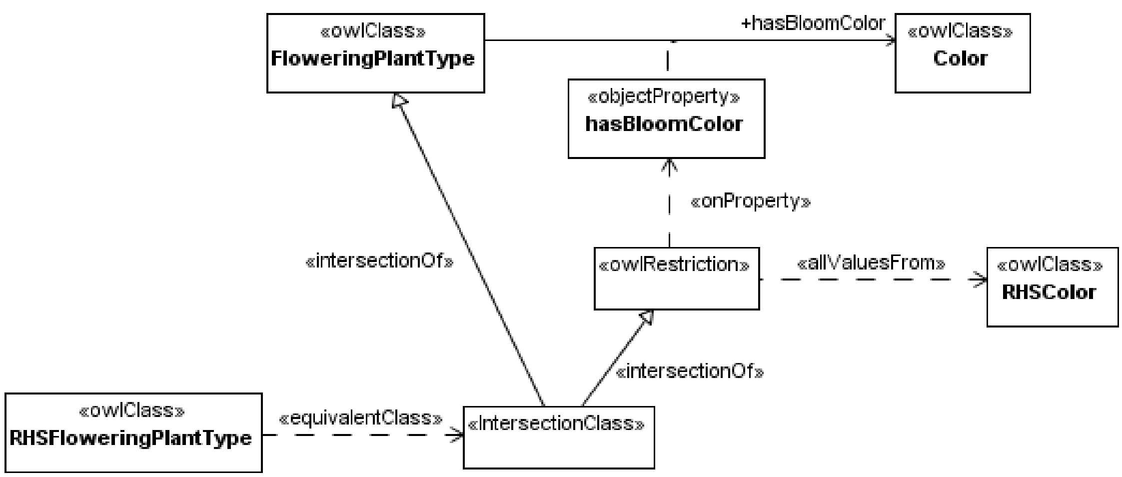

Figure 14.31 - Property restriction for owl:allValuesFrom with equivalence relation depicting a typical complex class expression

The RDF/XML serialized OWL corresponding toFigure 14.31 is:

```
<owl:ObjectProperty rdf:about="&re;hasBloomColor">
    <rdfs:range rdf:resource="&re;Color"/>
    <rdfs:domain rdf:resource="&re;FloweringPlantType"/>
</owl:ObjectProperty>

<owl:Class rdf:about="&re;Color"/>
<owl:Class rdf:about="&re;FloweringPlantType"/>
<owl:Class rdf:about="&re;RHSColor"/>

<owl:Class rdf:about="&re;RHSFloweringPlantType">
```

```
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="&re;FloweringPlantType"/>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="&re;hasBloomColor"/>
                        <owl:allValuesFrom rdf:resource="&re;RHSColor"/>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
```

Graphical Representation for Existential Quantification (some values) and Specific Value (has value) Restrictions

Similar to `owl:allValuesFrom`, `owl:someValuesFrom` is a built-in OWL property that links a restriction class to either a class description or a data range. A restriction containing an `owl:someValuesFrom` constraint is used to describe a class or data range for which at least one value of the property concerned is either a member of the class extension of the class description or a data value within the specified data range. In other words, it defines a class of individuals x for which there is at least one y (either an instance of the class description or value of the data range) such that the pair (x,y) is an instance of P. This does not exclude that there are other instances (x,y') of P for which y' does not belong to the class description or data range.

The value constraint `owl:hasValue` is a built-in OWL property that links a restriction class to a value V, which can be either an individual or a data value (or class in OWL Full). A restriction containing an `owl:hasValue` constraint describes a class of all individuals for which the property concerned has at least one value semantically equal to V (it may have other values as well).

The graphical notation for owl:someValuesFrom and owl:hasValue is similar to that shown above for owl:allValuesFrom. Figure 14.32 includes examples of each kind of restriction, used in yet another common pattern. Here, an Azalea is defined as having at least one value for hasBloomColorPattern specified by the ASAColorPattern class. A SingleColoredAzalea must have a solid color pattern and exactly one bloom color.

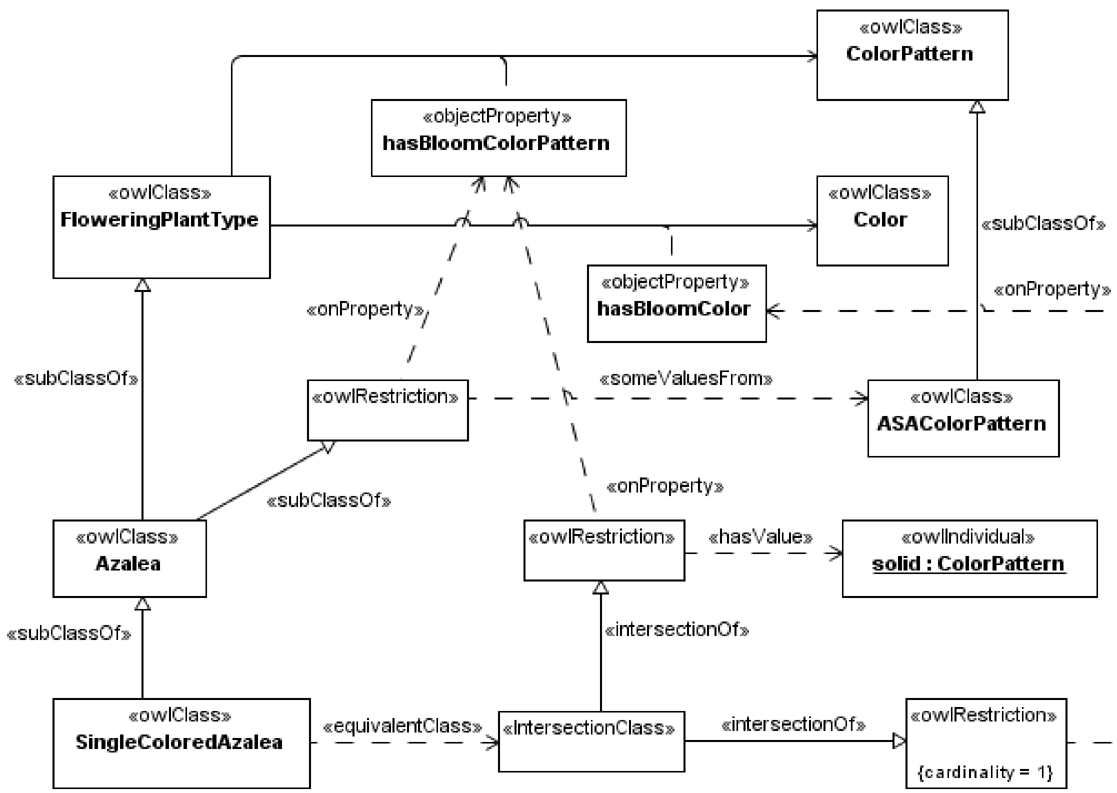

Figure 14.32 - Property restrictions for owl:someValuesFrom and owl:hasValue

The RDF/XML serialized OWL corresponding to Figure 14.32 is:

```
 <owl:ObjectProperty rdf:about="&re;hasBloomColor">
    <rdfs:range rdf:resource="&re;Color"/>
    <rdfs:domain rdf:resource="&re;FloweringPlantType"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="&re;hasBloomColorPattern">
    <rdfs:range rdf:resource="&re;ColorPattern"/>
    <rdfs:domain rdf:resource="&re;FloweringPlantType"/>
</owl:ObjectProperty>

<owl:Class rdf:about="&re;ASAColor"/>

<owl:Class rdf:about="&re;ASAColorPattern">
    <rdfs:subClassOf rdf:resource="&re;ColorPattern"/>
</owl:Class>

<owl:Class rdf:about="&re;Azalea">
    <rdfs:subClassOf rdf:resource="&re;FloweringPlantType"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:allValuesFrom>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
```

```
                        <rdf:Description rdf:about="&re;ASAColor"/>
                        <rdf:Description rdf:about="&re;RHSColor"/>
                    </owl:unionOf>
                </owl:Class>
            </owl:allValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColorPattern"/>
            <owl:someValuesFrom rdf:resource="&re;ASAColorPattern"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&re;Color"/>

<owl:Class rdf:about="&re;ColorPattern"/>

<owl:Class rdf:about="&re;FloweringPlantType">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&re;RHSColor"/>

<owl:Class rdf:about="&re;SingleColoredAzalea">
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&re;hasBloomColorPattern"/>
                    <owl:hasValue rdf:resource="&re;solid"/>
                </owl:Restriction>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&re;hasBloomColor"/>
                    <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
                </owl:Restriction>
            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource="&re;Azalea"/>
</owl:Class>

<owl:NamedIndividual rdf:about="&re;solid">
    <rdfs:label>solid colored, also called self</rdfs:label>
    <rdf:type rdf:resource="&re;ColorPattern"/>
</owl:NamedIndividual>
```

The graphical notation and patterns typically found in OWL ontologies for data properties are similar to those given above for object properties. Figure 14.33 includes example data restrictions using typical patterns, with tag detail shown on a number of the elements. Here, the example shows that azaleas prefer a moderate climate and full shade.

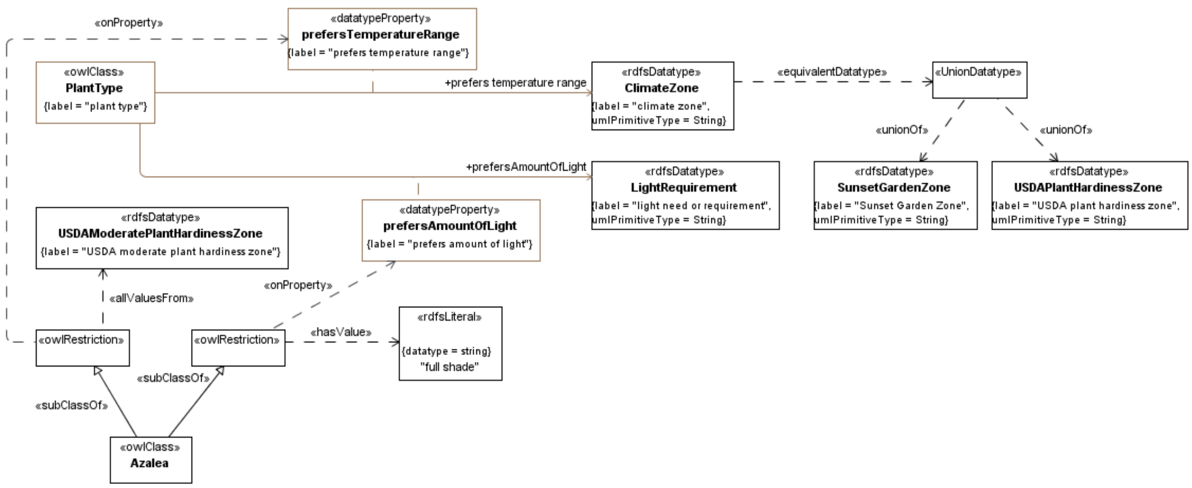

Figure 14.33 - Data property restrictions for owl:allValuesFrom and owl:hasValue

The RDF/XML serialized OWL corresponding to Figure 14.33 is:

```
 <owl:DatatypeProperty rdf:about="&re;prefersAmountOfLight">
    <rdfs:label>prefers amount of light</rdfs:label>
    <rdfs:range rdf:resource="&re;LightRequirement"/>
    <rdfs:domain rdf:resource="&re;PlantType"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="&re;prefersTemperatureRange">
    <rdfs:label>prefers temperature range</rdfs:label>
    <rdfs:range rdf:resource="&re;ClimateZone"/>
    <rdfs:domain rdf:resource="&re;PlantType"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="&re;Azalea">
    <rdfs:subClassOf rdf:resource="&re;FloweringPlantType"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColorPattern"/>
            <owl:someValuesFrom rdf:resource="&re;ASAColorPattern"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;prefersTemperatureRange"/>
            <owl:allValuesFrom rdf:resource="&re;USDAModeratePlantHardinessZone"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;prefersAmountOfLight"/>
            <owl:hasValue rdf:datatype="&xsd;string">full shade</owl:hasValue>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:allValuesFrom>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
                        <rdf:Description rdf:about="&re;ASAColor"/>
                        <rdf:Description rdf:about="&re;RHSColor"/>
                    </owl:unionOf>
                </owl:Class>
            </owl:allValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>
```

```
</owl:Class>

<owl:Class rdf:about="&re;PlantType">
    <rdfs:label>plant type</rdfs:label>
</owl:Class>

<rdfs:Datatype rdf:about="&re;ClimateZone">
    <rdfs:label>climate zone</rdfs:label>
    <owl:equivalentClass>
        <rdfs:Datatype>
            <owl:unionOf rdf:parseType="Collection">
                <rdf:Description rdf:about="&re;SunsetGardenZone"/>
                <rdf:Description rdf:about="&re;USDAPlantHardinessZone"/>
            </owl:unionOf>
        </rdfs:Datatype>
    </owl:equivalentClass>
</rdfs:Datatype>

<rdfs:Datatype rdf:about="&re;LightRequirement">
    <rdfs:label>light need or requirement</rdfs:label>
    <owl:equivalentClass>
        <rdfs:Datatype>
            <owl:oneOf>
                <rdf:Description>
                    <rdf:type rdf:resource="&rdf;List"/>
                    <rdf:first rdf:datatype="&xsd;string">filtered sun</rdf:first>
                    <rdf:rest>
                        <rdf:Description>
                            <rdf:type rdf:resource="&rdf;List"/>
                            <rdf:first rdf:datatype="&xsd;string">full shade</rdf:first>
                            <rdf:rest>
                                <rdf:Description>
                                    <rdf:type rdf:resource="&rdf;List"/>
                                    <rdf:first rdf:datatype="&xsd;string">full sun</rdf:first>
                                    <rdf:rest>
                                        <rdf:Description>
                                            <rdf:type rdf:resource="&rdf;List"/>
                                            <rdf:first rdf:datatype="&xsd;string">partial shade</rdf:first>
                                            <rdf:rest>
                                                <rdf:Description>
                                                    <rdf:type rdf:resource="&rdf;List"/>
                                                    <rdf:first rdf:datatype="&xsd;string">partial sun</rdf:first>
                                                    <rdf:rest rdf:resource="&rdf;nil"/>
                                                </rdf:Description>
                                            </rdf:rest>
                                        </rdf:Description>
                                    </rdf:rest>
                                </rdf:Description>
                            </rdf:rest>
                        </rdf:Description>
                    </rdf:rest>
                </rdf:Description>
            </owl:oneOf>
        </rdfs:Datatype>
    </owl:equivalentClass>
</rdfs:Datatype>

<rdfs:Datatype rdf:about="&re;SunsetGardenZone">
    <rdfs:label>Sunset Garden Zone</rdfs:label>
</rdfs:Datatype>

<rdfs:Datatype rdf:about="&re;USDAModeratePlantHardinessZone">
    <rdfs:label>USDA moderate plant hardiness zone</rdfs:label>
    <owl:equivalentClass>
        <rdfs:Datatype>
            <owl:oneOf>
                <rdf:Description>
                    <rdf:type rdf:resource="&rdf;List"/>
                    <rdf:first rdf:datatype="&xsd;string">Zone 6a</rdf:first>
                    <rdf:rest>
                        <rdf:Description>
                            <rdf:type rdf:resource="&rdf;List"/>
                            <rdf:first rdf:datatype="&xsd;string">Zone 6b</rdf:first>
                            <rdf:rest>
                                <rdf:Description>
                                    <rdf:type rdf:resource="&rdf;List"/>
```

```
                                                <rdf:first rdf:datatype="&xsd;string">Zone 7a</rdf:first>
                                                <rdf:rest>
                                                    <rdf:Description>
                                                        <rdf:type rdf:resource="&rdf;List"/>
                                                        <rdf:first rdf:datatype="&xsd;string">Zone 7b</rdf:first>
                                                        <rdf:rest>
                                                            <rdf:Description>
                                                                <rdf:type rdf:resource="&rdf;List"/>
                                                                <rdf:first rdf:datatype="&xsd;string">Zone 8a</rdf:first>
                                                                <rdf:rest>
                                                                    <rdf:Description>
                                                                        <rdf:type rdf:resource="&rdf;List"/>
                                                                       <rdf:first rdf:datatype="&xsd;string">Zone 8b</rdf:first>
                                                                        <rdf:rest rdf:resource="&rdf;nil"/>
                                                                    </rdf:Description>
                                                                </rdf:rest>
                                                            </rdf:Description>
                                                        </rdf:rest>
                                                    </rdf:Description>
                                                </rdf:rest>
                                            </rdf:Description>
                                        </rdf:rest>
                                    </rdf:Description>
                                </rdf:rest>
                            </rdf:Description>
                        </owl:oneOf>
                    </rdfs:Datatype>
                </owl:equivalentClass>
            </rdfs:Datatype>
```

OWL 2 provides facilities for further refinement of data ranges, as shown in Figure 14.34. The figure includes examples of data restrictions restriction, using common patterns, with tag detail shown on a number of the elements. Here, the example shows that azaleas prefer a moderate climate and full shade.

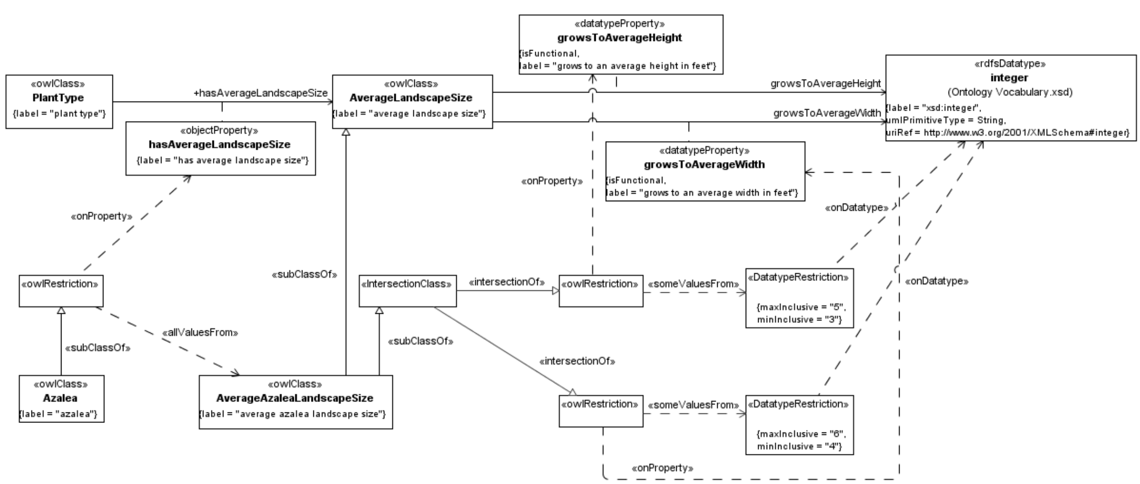

Figure 14.34 - Data property restrictions with refined data range for owl:someValuesFrom

The RDF/XML serialized OWL corresponding to Figure 14.34 is:

```
<owl:ObjectProperty rdf:about="&re;hasAverageLandscapeSize">
    <rdfs:label>has average landscape size</rdfs:label>
    <rdfs:range rdf:resource="&re;AverageLandscapeSize"/>
    <rdfs:domain rdf:resource="&re;PlantType"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:about="&re;growsToAverageHeight">
```

```
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:label>grows to an average height in feet</rdfs:label>
    <rdfs:domain rdf:resource="&re;AverageLandscapeSize"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="&re;growsToAverageWidth">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:label>grows to an average width in feet</rdfs:label>
    <rdfs:domain rdf:resource="&re;AverageLandscapeSize"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="&re;AverageAzaleaLandscapeSize">
    <rdfs:label>average azalea landscape size</rdfs:label>
    <rdfs:subClassOf rdf:resource="&re;AverageLandscapeSize"/>
    <rdfs:subClassOf>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&re;growsToAverageHeight"/>
                    <owl:someValuesFrom>
                        <rdfs:Datatype>
                            <owl:onDatatype rdf:resource="&xsd;integer"/>
                            <owl:withRestrictions rdf:parseType="Collection">
                                <rdf:Description>
                                 <xsd:minInclusive rdf:datatype="&xsd;integer">3</xsd:min-
Inclusive>
                                </rdf:Description>
                                <rdf:Description>
                                 <xsd:maxInclusive rdf:datatype="&xsd;integer">5</xsd:max-
Inclusive>
                                </rdf:Description>
                            </owl:withRestrictions>
                        </rdfs:Datatype>
                    </owl:someValuesFrom>
                </owl:Restriction>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="&re;growsToAverageWidth"/>
                    <owl:someValuesFrom>
                        <rdfs:Datatype>
                            <owl:onDatatype rdf:resource="&xsd;integer"/>
                            <owl:withRestrictions rdf:parseType="Collection">
                                <rdf:Description>
                                 <xsd:minInclusive rdf:datatype="&xsd;integer">4</xsd:min-
Inclusive>
                                </rdf:Description>
                                <rdf:Description>
                                 <xsd:maxInclusive rdf:datatype="&xsd;integer">6</xsd:max-
Inclusive>
                                </rdf:Description>
                            </owl:withRestrictions>
                        </rdfs:Datatype>
                    </owl:someValuesFrom>
                </owl:Restriction>
```

```
            </owl:intersectionOf>
        </owl:Class>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&re;AverageLandscapeSize">
    <rdfs:label>average landscape size</rdfs:label>
</owl:Class>

<owl:Class rdf:about="&re;Azalea">
    <rdfs:label>azalea</rdfs:label>
    <rdfs:subClassOf rdf:resource="&re;FloweringPlantType"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;prefersTemperatureRange"/>
            <owl:allValuesFrom rdf:resource="&re;USDAModeratePlantHardinessZone"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;prefersAmountOfLight"/>
            <owl:hasValue rdf:datatype="&xsd;string">full shade</owl:hasValue>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasAverageLandscapeSize"/>
            <owl:allValuesFrom rdf:resource="&re;AverageAzaleaLandscapeSize"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColorPattern"/>
            <owl:someValuesFrom rdf:resource="&re;ASAColorPattern"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&re;hasBloomColor"/>
            <owl:allValuesFrom>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
                        <rdf:Description rdf:about="&re;ASAColor"/>
                        <rdf:Description rdf:about="&re;RHSColor"/>
                    </owl:unionOf>
                </owl:Class>
            </owl:allValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&re;PlantType">
    <rdfs:label>plant type</rdfs:label>
</owl:Class>
```

Note that the graphical notation and use of "DatatypeRestriction" together with XML Schema facets for expressing numeric data ranges is specified in 14.3.8.

14.3.5.4 owl:intersectionOf Class Description

Description

`owl:intersectionOf` links a class to a list of class descriptions, describing an anonymous class for which the class extension contains precisely those individuals that are members of the class extension of all class descriptions in the list. «intersectionOf» is analogous to logical conjunction.

Stereotype and Base Class

«intersectionOf» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies to generalizations with a common subtype.

[2] All instances that are instances of every super type along generalizations that are stereotyped by «intersectionOf» are instances of the subtype.

[3] (Semantic) All instances of the subtype are instances of all of the super types.

Graphical Representation

Dashed line between generalization lines with stereotype label, as shown in Figure 14.35.

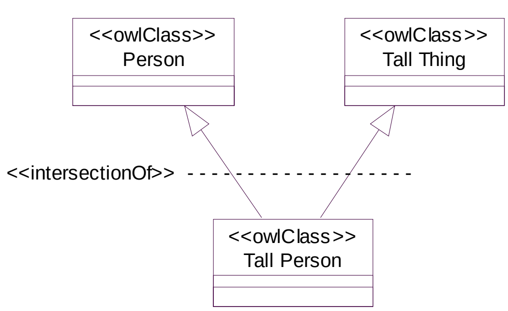

Figure 14.35 - Example Using owl:intersectionOf

The stereotype is based on UML::Generalization rather than UML::Class, so there can be other supertypes not required by the intersection. Use of UML::GeneralizationSet was prohibited in this case, because it requires one supertype – its semantics refers to the instances of the subtypes, not the supertypes.

14.3.5.5 owl:unionOf Class Description

Description

`owl:unionOf` links a class to a list of class descriptions, describing an anonymous class for which the class extension contains those individuals that occur in at least one of the class extensions of the class descriptions in the list. `owl:unionOf` is analogous to logical disjunction.

Stereotype and Base Class

No stereotype needed. Use UML::GeneralizationSet with isCovering = true, as shown in Figure 14.36. For consistency with the other class descriptions, vendors can also optionally define a «unionOf» stereotype of UML::Constraint, applied to UML::Generalization (similar to intersection, above).

Parent

None.

Properties

None.

Constraints

[1] (Semantic) All instances of the supertype are instances of at least one of the subtypes.

Graphical Representation

Dashed line between generalization lines labeled with "{complete}."

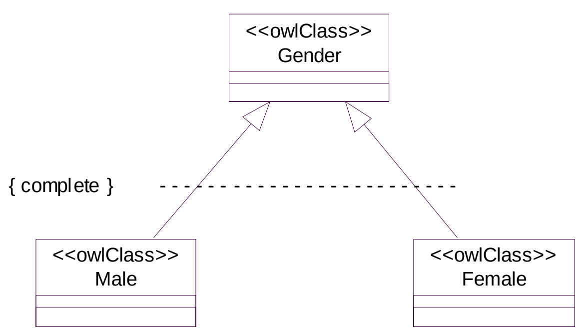

Figure 14.36 - Example Using owl:unionOf

14.3.5.6 owl:complementOf Class Description

Description

`owl:complementOf` links a class to precisely one other class, and describes a class for which the extension contains exactly those individuals that do not belong to the extension of the other class. `owl:complementOf` is analogous to logical negation: the class extension consists of those individuals that are NOT members of the extension of the complement class.

Stereotype and Base Class

«complementOf» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies to constraints between exactly two classes.

[2] (Semantic). All instances (of `owl:Thing`) are instances of exactly one of the two classes.

Graphical Representation

Dashed line between two classes with stereotype label. An arrowhead should be used opposite from the class that will have owl:complementOf in XML syntax (since all RDF, RDF Schema, and OWL graphs are unidirectional, by definition). Shorthand representation that eliminates the arrowhead may be used within an ontology, but XML production in this case should result in two instances of `owl:complementOf` – one for each "side" of the bidirectional constraint.

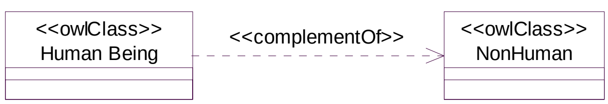

Figure 14.37 - Example Using owl:complementOf

14.3.5.7 owl:disjointWith Class Axiom

Description

`owl:disjointWith` is a built-in OWL property with a class description as domain and range. Each `owl:disjointWith` statement asserts that the class extensions of the two class descriptions involved have no individuals in common. A class axiom may also contain (multiple) `owl:disjointWith` statements. Like axioms with `rdfs:subClassOf`, declaring two classes to be disjoint is a partial definition: it imposes a necessary but not sufficient condition on the class.

Stereotype and Base Class

«disjointWith» with base class of UML::Constraint, or use disjoint UML generalizations with no stereotype.

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between classes.

[2] (Semantic). An individual can only be a member of one class participating in a particular disjoint set of classes.

Graphical Representation

Dashed line between two classes with stereotype label. An arrowhead should be used opposite from the class that will have «disjointWith» in XML syntax (since all RDF, RDF Schema, and OWL graphs are unidirectional, by definition). Shorthand representation that eliminates the arrowhead may be used within an ontology, but XML production in this case should result in two instances of «disjointWith» – one for each "side" of the bidirectional constraint.

Figure 14.38 - Example Using owl:disjointWith

In cases where there are multiple participants in the same «disjointWith» class axiom, a constraint note with stereotype label and dashed lines to more than one class should be used, as shown in Figure 14.39.

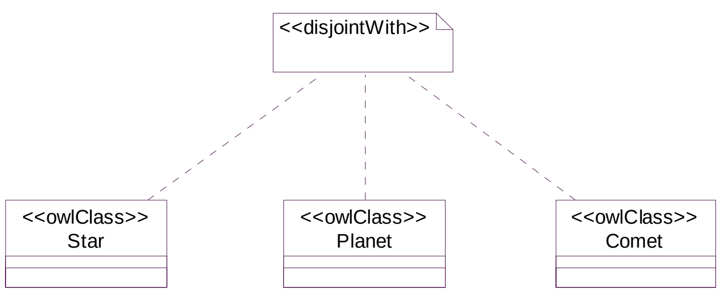

Figure 14.39 - Example Using owl:disjointWith With Multiple Participants

Alternatively, if the classes have a common supertype, use UML::GeneralizationSet with isDisjoint = true. Representation is dashed line between generalization lines labeled with "{disjoint}."

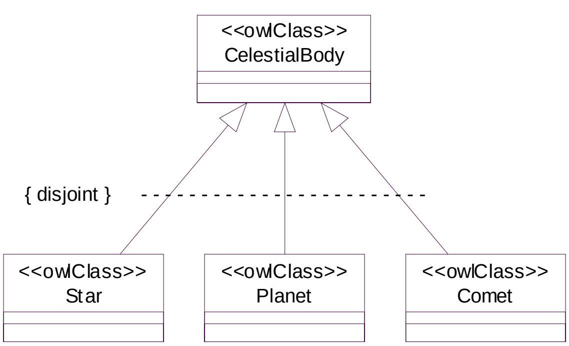

Figure 14.40 - Example Using owl:disjointWith With Common Supertype

14.3.5.8 owl:equivalentClass Class Axiom

Description

`owl:equivalentClass` is a built-in property that links a class description to another class description. The meaning of such a class axiom is that the two class descriptions involved have the same class extension (i.e., both class extensions contain exactly the same set of individuals). A class axiom may contain (multiple) `owl:equivalentClass` statements.

Stereotype and Base Class

«equivalentClass» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between classes.

[2] (Semantic). The classes have exactly the same instances.

Graphical Representation

Dashed line between two classes with stereotype label. An arrowhead should be used opposite from the class that will have owl:equivalentClass in XML syntax. Shorthand notation that eliminates the arrowhead may be used within an ontology, but XML production in this case should result in two instances of «equivalentClass» – one for each "side" of the bidirectional constraint.

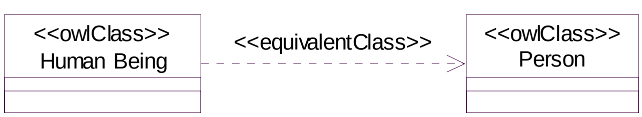

Figure 14.41 - Example Using owl:equivalentClass

Alternatively two UML::Generalizations may be used, again within a given ontology, if such circular definitions are supported by the tool (i.e., class a generalizes class b and vice versa).

In cases where there are multiple participants in the same «equivalentClass» class axiom, a constraint note with stereotype label and dashed lines to more than one class should be used, similarly to the example used for «disjointWith».

14.3.6 Properties

OWL distinguishes between two main categories of properties:

- Object properties link individuals to individuals.
- Datatype properties link individuals to data values.

Note: OWL also has the notion of annotation properties (`owl:AnnotationProperty`) and ontology properties (`owl:OntologyProperty`).

A property axiom defines characteristics of a property. In its simplest form, a property axiom just defines the existence of a property. Often, property axioms define additional characteristics of properties. OWL supports the following constructs for property axioms:

- RDF Schema constructs: `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range`
- relations to other properties: `owl:equivalentProperty` and `owl:inverseOf`
- global cardinality restrictions: `owl:FunctionalProperty` and `owl:InverseFunctionalProperty`
- logical property characteristics: `owl:AsymmetricProperty`, `owl:IrreflexiveProperty`, `owl:ReflexiveProperty`, `owl:SymmetricProperty`, and `owl:TransitiveProperty`

The relevant RDF Schema concepts are defined in 14.2.7; global cardinality restrictions and logical property characteristics are represented as UML properties on either «owlProperty» or «objectProperty», as given below.

14.3.6.1 owl:DatatypeProperty

Description

A datatype property is defined as an instance of the built-in OWL class `owl:DatatypeProperty`. The built-in class, `owl:DatatypeProperty`, is a subclass of the built-in class `rdf:Property`.

Note: In OWL Full, object properties and datatype properties are not disjoint. Because data values can be treated as individuals, datatype properties are effectively subclasses of object properties. In OWL Full `owl:ObjectProperty` is equivalent to `rdf:Property`. In practice, this mainly has consequences for the use of `owl:InverseFunctionalProperty`.

Stereotype and Base Class

«datatypeProperty» with base class of UML::Class, UML::AssociationClass, UML::Property, and UML::Association.

Parent

«owlProperty».

Properties

None.

Constraints

[1] (Semantics) The values of a property stereotyped by «datatypeProperty» must be either strings that can be represented as UML::LiteralString stereotyped by «rdfsLiteral», values corresponding to the enumerated literals represented as a UML::Enumeration stereotyped by «dataRange», or instances of a UML::Class stereotyped by «rdfsDatatype» (i.e., one of the XML Schema Datatypes or a built-in datatype).

14.3.6.2 Object Properties in OWL

Description

An object property is defined as an instance of the built-in OWL class `owl:ObjectProperty`. The built-in class, `owl:ObjectProperty`, is a subclass of the built-in class `rdf:Property`.

If a property is declared to be inverse-functional, then the object of a property statement uniquely determines the subject (some individual). More formally, if we state that P is an `owl:InverseFunctionalProperty`, then this asserts that a value y can only be the value of P for a single instance x, i.e., there cannot be two distinct instances x_1 and x_2 such that both pairs (x_1, y) and (x_2, y) are instances of P. See 4.3.2 of [OWL Reference] for additional detail, including an explanation of the notion of global cardinality restrictions and use of `owl:InverseFunctionalProperty` to represent keys in the context of a relational database.

A symmetric property is a property for which holds that if the pair (x, y) is an instance of P, then the pair (y, x) is also an instance of P.

When one defines a property P to be a transitive property, this means that if a pair (x, y) is an instance of P, and the pair (y, z) is also instance of P, then we can infer the pair (x, z) is also an instance of P.

Stereotype and Base Class

«objectProperty» with base class of UML::Class, UML::AssociationClass, UML::Property, and UML::Association.

Parent

«owlProperty».

Properties

- isInverseFunctional: Boolean [0..1] – when true, indicates that the property in question is inverse functional.
- isSymmetric: Boolean [0..1] – when true, indicates that the property in question is symmetric.
- isTransitive: Boolean [0..1] – when true, indicates that the property in question is transitive.
- isAsymmetric: Boolean [0..1] - when true, indicates that the property in question is asymmetric.
- isIrreflexive: Boolean [0..1] - when true, indicates that the property is irreflexive.
- isReflexive: Boolean [0..1] - when true, indicates that the property is reflexive.

Constraints

[1] The type of a property stereotyped by «objectProperty» must be a UML::Class stereotyped by «owlClass».

[2] The isAsymmetric, isInverseFunctional, isIrreflexive, isReflexive, isSymmetric, and isTransitive properties apply only to properties stereotyped by «objectProperty».

[3] The type of a property with isSymmetric set to true must be the same as the class on which it is defined.

[4] In OWL DL, no local or global cardinality restrictions can be declared on a property with isTransitive set to true, or on any of its super properties, nor on its inverse or on any super properties of its inverse.

14.3.6.3 owl:Property

Description

The notion of an `owl:Property`, as defined in the metamodel and redefined here in the profile is an abstract class.

A functional property is a property that can have only one (unique) value y for each instance x, i.e., there cannot be two distinct values y_1 and y_2 such that the pairs (x, y_1) and (x, y_2) are both instances of this property. Both object properties and datatype properties can be declared as “functional,” thus, we introduce it at `owl:Property`.

Stereotype and Base Class

«owlProperty» with base class of UML::Class, UML::AssociationClass, UML::Property, and UML::Association.

Parent

«rdfProperty».

Properties

- isDeprecated: Boolean [0..1] – indicates a particular property definition is deprecated.
- isFunctional: Boolean [0..1] – when true, indicates that the property in question is functional.

Constraints

[1] The isFunctional property applies only to properties stereotyped by «owlProperty», «objectProperty», or «datatypeProperty».

14.3.6.4 owl:equivalentProperty Relation

Description

`owl:equivalentProperty` can be used to state that two properties have the same property extension. Syntactically, `owl:equivalentProperty` is a built-in OWL property with `rdf:Property` as its domain and range.

Note: Property equivalence is not the same as property equality. Equivalent properties have the same "values" (i.e., the same property extension), but may have different intensional meaning (i.e., denote different concepts). Property equality should be expressed with the `owl:sameAs` construct. As this requires that properties are treated as individuals, such axioms are only allowed in OWL Full.

Stereotype and Base Class

«equivalentProperty» stereotype of UML::Constraint between classes stereotyped as «rdfProperty», «owlProperty», «objectProperty», or «datatypeProperty».

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between properties with «rdfGlobal» applied, or properties on the class at which they are introduced.

[2] (Semantic) Instances of equivalent properties (property extensions, or sets of tuples) are the same.

Graphical Representation

Dashed line between two association classes with stereotype label. An arrowhead should be used opposite from the association class that will have «equivalentProperty» in XML syntax. Shorthand notation that eliminates the arrowhead may be used within an ontology, but XML production should result in two instances of «equivalentProperty» – one for each "side" of the bidirectional constraint.

In cases where there are multiple participants in the same «equivalentProperty» relation, a constraint note with stereotype label and dashed lines to more than one association class representing the property should be used, similarly to the example for «disjointWith».

14.3.6.5 owl:inverseOf Relation

Description

OWL properties have a direction, from domain to range. In practice, people often find it useful to define relations in both directions: persons own cars, cars are owned by persons. `owl:inverseOf` can be used to define such an inverse relation between properties.

Syntactically, `owl:inverseOf` is a built-in OWL property with `owl:ObjectProperty` as its domain and range. An OWL axiom of the form `P1 owl:inverseOf P2` asserts that for every pair (x, y) in the property extension of P1, there is a pair (y, x) in the property extension of P2, and vice versa. Thus, `owl:inverseOf` is a symmetric property.

Stereotype and Base Class

«inverseOf» with base class of UML::Association, or use bidirectional associations with no stereotype.

Parent

None.

Properties

None.

Constraints

[1] Applies only to associations with «rdfGlobal» applied, or to properties on the class at which they are introduced.

[2] Applies only to binary, unidirectional associations.

[3] (UML) A property cannot be an inverse of itself (use the isSymmetric property).

Graphical Representation

We describe several options for modeling/representing inverses in UML.

A. The first is to use a simple association with properties as ends, i.e., a line between classes with properties on the ends closest to their ranges, for example, as shown in Figure 14.42.

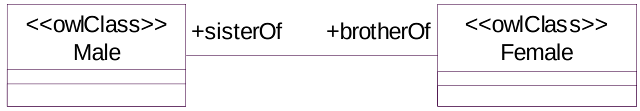

Figure 14.42 - Using owl:inverseOf With Bidirectional Representation

Additional constraint if this approach is taken:

[4] (UML) A property can have at most one inverse.

B. Alternatively, one could use an «inverseOf» stereotype of UML::Constraint between association classes for binary, unidirectional associations, as shown in Figure 14.43. An arrowhead should be used opposite from the association class that will have `owl:inverseOf` in XML syntax. Shorthand notation that eliminates the arrowhead may be used within an ontology, but XML production should result in two instances of «inverseOf» – one for each "side" of the bidirectional constraint.

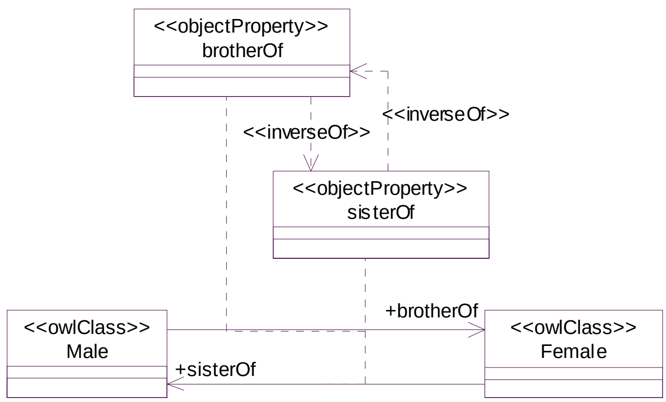

Figure 14.43 - Using owl:inverseOf Between Association Classes

C. A third notational option would be to use a stereotype «inverse» of a UML::Property with a property:

OF of type UML::Property

Using a similar representation to the approach taken in 14.3.5.3 for `owl:someValuesFrom` and `owl:hasValue`, put before the property name: "«inverse» {of = <property-name>, <property-name>}".

Additional constraint if this approach is taken:

[5] Value of OF property must refer to a property with «rdfGlobal» applied, or to properties on the class at which they were introduced.

14.3.6.6 surrogateObjectProperty

Description

A surrogate object property is a "stand-in" for an OWL object property, similar to RDF surrogates as defined in 14.2.7.3. As in the RDF case, OWL surrogates are intended for use on diagrams where one wants to show relationships between properties without "dragging all of the related baggage" (domain and range classes, for example) onto those diagrams. Examples where surrogates are useful include property hierarchies, complex restrictions, and property chains.

Stereotype and Base Class

«surrogateObjectProperty» with base class of UML::Class.

Parent

«surrogateProperty»

Properties

None.

Constraints

[1] A surrogate object property can only be defined for an existing UML property, association, or association class representing the OWL base object property.

[2] A dependency stereotyped by «surrogateFor», linking the surrogate to its base property definition is required.

Graphical Notation

Figure 14.44 provides an example of definition of a surrogate for a hasCode property, representing a common pattern used in terminology development.

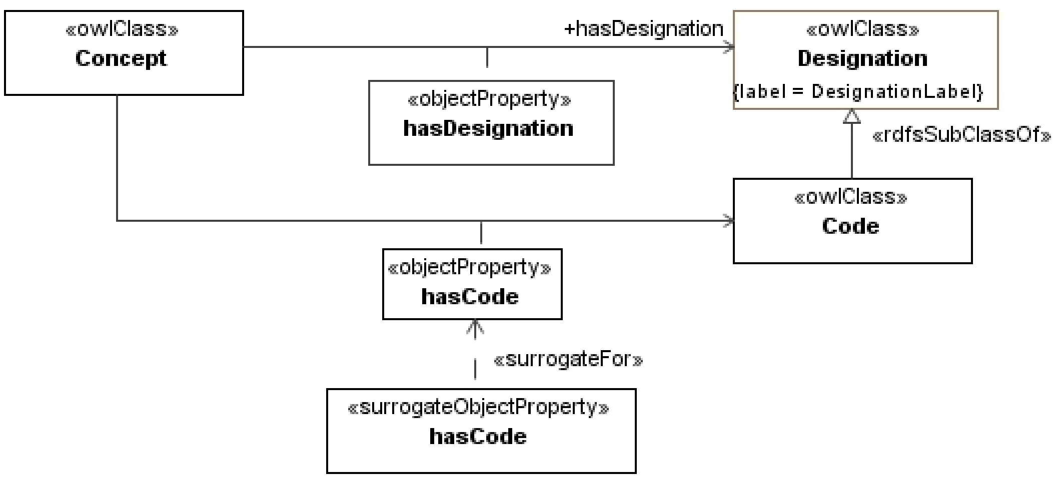

Figure 14.44 - Surrogate object property definition

The surrogate property can be used to simplify diagrams for other property/restriction definitions, shown in Figure 14.45.

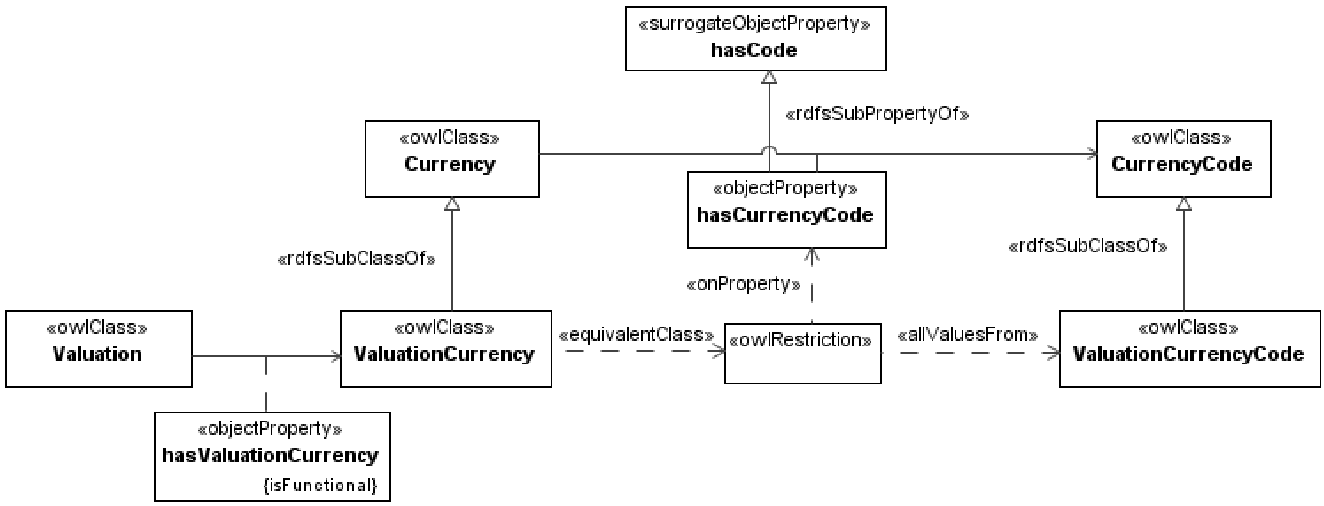

Figure 14.45 - Surrogate object property usage example

14.3.6.7 surrogateDatatypeProperty

Description

A surrogate datatype property is a “stand-in” for an OWL datatype property, similar to RDF surrogates and OWL object property surrogates.

Stereotype and Base Class

«surrogateDatatypeProperty» with base class of UML::Class.

Parent

«surrogateProperty».

Properties

None.

Constraints

[1] A surrogate datatype property can only be defined for an existing UML property, association, or association class representing the OWL base datatype property.

[2] A dependency stereotyped by «surrogateFor», linking the surrogate to its base property definition is required.

14.3.7 Individuals

Individuals are defined with individual axioms (also called “facts”). These include:

- Facts about class membership and property values of individuals
- Facts about individual identity

Many languages have a so-called "unique names" assumption: different names refer to different things in the world. On the web, such an assumption is not possible. For example, the same person could be referred to in many different ways (i.e., with different IRIs).

OWL provides three constructs for stating facts about the identity of individuals:

- `owl:sameAs` is used to state that two IRIs refer to the same individual.
- `owl:differentFrom` is used to state that two IRIs refer to different individuals.
- `owl:AllDifferent` provides an idiom for stating that a list of individuals are all different.

14.3.7.1 Class Membership and Property Values of Individuals

Description

Many facts typically are statements indicating class membership of individuals and property values of individuals. Individual axioms need not necessarily be about named individuals: they can also refer to anonymous individuals.

Stereotype and Base Class

«owlIndividual» with a base class of UML::InstanceSpecification, typed by a class having the properties desired for the individual. The class may be stereotyped by «singleton» to indicate it is for a specific individual[5]. Classes stereotyped by «singleton» are not translated to OWL, and their properties appear in OWL as properties of the individual.

Parent

None.

Properties

None.

Constraints

[1] Classes stereotyped by «singleton» have exactly one instance each.

[2] The instance specification stereotyped by «owlIndividual» has only one value, i.e., it specifies a single individual.

Graphical Representation

Instance specifications use the same symbol as classes, but their names are underlined, and have a colon separating the instance name from the class name. Singleton classes can be anonymous, omitted from the representation, and generated by tools. Instances of anonymous classes show nothing after the colon.

14.3.7.2 owl:sameAs Relation

Description

`owl:sameAs` links an individual to an individual, indicating that two IRIs actually refer to the same thing: the individuals have the same "identity." `owl:sameAs` statements are often used in defining mappings between ontologies.

Additionally, in OWL Full, where a class can be treated as instances of (meta)classes, `owl:sameAs` can be used to define class equality, thus indicating that two concepts have the same intensional meaning.

5. UML supports individuals without classes and properties on such individuals, for tools that choose to support it.

Stereotype and Base Class

«sameAs» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between instance specifications, or for modeling OWL Full, between instances or between classes.

Graphical Representation

Dashed line between two instances (or classes) with stereotype label. An arrowhead can be used opposite from the instance (or class) that will have «sameAs» in XML syntax.

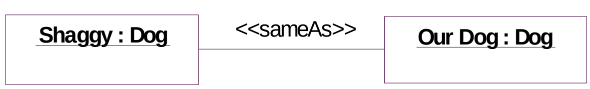

Figure 14.46 - Using owl:sameAs Between Instances

Constraint note with stereotype label and dashed lines to more than one instance (or class - translates to multiple «sameAs» statements).

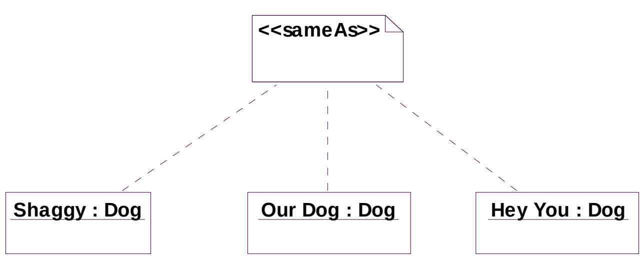

Figure 14.47 - Using owl:sameAs Between Instances

14.3.7.3 owl:differentFrom Relation

Description

The built-in `owl:differentFrom` property links an individual to an individual, indicating that two IRIs refer to different individuals.

Stereotype and Base Class

«differentFrom» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between instance specifications.

Graphical Representation

Dashed line between two instances with stereotype label. An arrowhead can be used opposite from the instance that will have «differentFrom» in XML syntax.

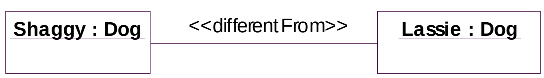

Figure 14.48 - Using owl:differentFrom Between Instances

Constraint note with stereotype label and dashed lines to more than one instance (translates to multiple «differentFrom» statements).

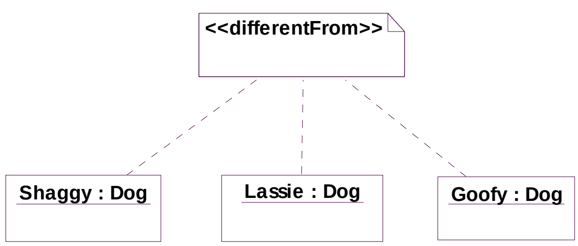

Figure 14.49 - Using owl:differentFrom Between Instances

14.3.7.4 owl:AllDifferent Construct

Description

For ontologies in which the unique-names assumption holds, the use of `owl:differentFrom` is likely to lead to a large number of statements, as all individuals have to be declared pairwise disjoint. For such situations OWL provides a special idiom in the form of `owl:AllDifferent`. `owl:AllDifferent` is a special built-in OWL class, for which the property `owl:distinctMembers` is defined, which links an instance of `owl:AllDifferent` to a list of individuals. The intended meaning of such a statement is that all individuals in the list are all different from each other.

Stereotype and Base Class

«allDifferent» with base class of UML::Constraint.

Parent

None.

Properties

None.

Constraints

[1] Applies only to constraints between instance specifications.

Graphical Representation

Constraint note with stereotype label and dashed lines to more than one instance.

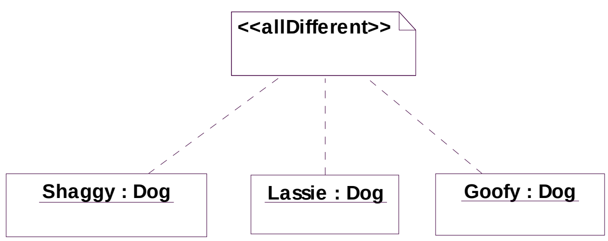

Figure 14.50 - Using owl:AllDifferent Between Instances

14.3.7.5 Individual Property Values

Description

In RDF, RDF Schema and OWL properties of individuals are accessed essentially through the *triples* (or statements), where the individual is the subject of the triple. In this profile, while we have optionally provided explicit access to the elements of the triple in a way that identifies the subject for this purpose, we also provide a more intuitive representation from a UML perspective.

Stereotype and Base Class

No stereotype, use UML::Slot to represent properties on individuals.

Parent

None.

Properties

None.

Constraints

[1] Values must conform to constraints on the property, such as type and multiplicity.

Graphical Representation

Put values after equal sign at end of property entry in instance.

14.3.8 Datatypes and Data Ranges

OWL 2 ontologies can refer to data values such as strings or integers. Similar to datatypes in UML, each of these kinds of values is called a datatype. Each datatype is identified by an IRI and is defined by the following components:

- The value space is the set of values of the datatype. Elements of the value space are called data values.
- The lexical space is a set of strings that can be used to refer to data values. Each member of the lexical space is called a lexical form, and it is mapped to a particular data value.
- The facet space is a set of pairs of the form (F , v) where F is an IRI called a constraining facet, and v is an arbitrary data value called the constraining value. Each such pair is mapped to a subset of the value space of the datatype.

Datatypes, such as `xsd:string` or `xsd:integer`, and literals such as `"1"^^xsd:integer`, can be used to express data ranges - sets of tuples of literals, where tuples consisting of only one literal are identified by the literal itself. Data ranges can be used as the range of a data property or in restrictions on data properties. Figure 14.51 depicts the set of stereotypes available for use in the OWL profile to support datatypes and data ranges.

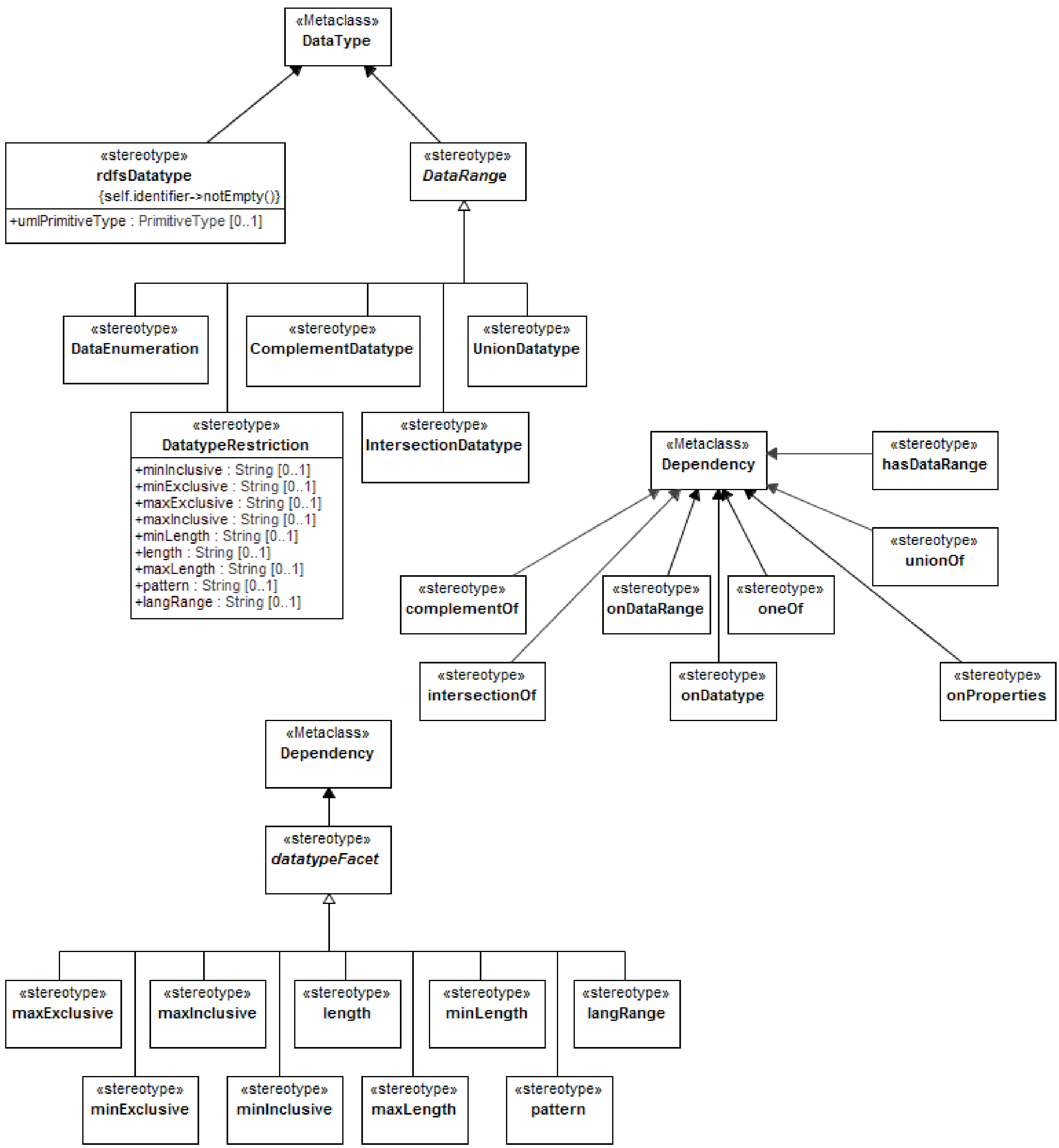

Figure 14.51 Datatypes and Data Ranges

OWL makes use of the RDF datatyping scheme, which provides a mechanism for referring to XML Schema datatypes. Data values are instances of the RDF Schema class `rdfs:Literal`. Datatypes are instances of the class `rdfs:Datatype`.

The RDF Semantics document recommends use of a subset of the simple built-in XML Schema datatypes. The set of XML Schema datatypes that are allowable for use in OWL DL are given in the model library provided in Annex A.

14.3.8.1 Datatype Definitions

Description

A datatype definition relates a custom datatype, to its data range. This provides a mechanism for naming custom data ranges, where the source for the dependency is an «rdfsDatatype» and target may include another «rdfsDatatype» or *«DataRange»*.

Stereotype and Base Class

«hasDataRange» with base class of UML::Dependency

Parent

None.

Properties

None.

Constraints

None.

14.3.8.2 Data Ranges

Description

Data ranges can be used in the range of a data property directly or in restrictions on data properties. Specializations of this stereotype provide for the standard set-theoretic operations on data ranges; in logical languages these are usually called conjunction, disjunction, and negation, respectively. They also provide facilities for enumerated literals and defining restrictions on the value space of a datatype by a constraining facet.

Note that a number of the stereotypes required for use with data ranges are defined in 14.3.5 and repeated on Figure 14.51 for convenience.

Stereotype and Base Class

«DataRange» with base class of UML::Datatype.

Parent

None.

Properties

None.

Constraints

None.

14.3.8.3 Complement Data Range

Description

A complement data range contains all tuples of literals that are not contained in its complement (i.e., it represents the logical complement, and is used to effectively support negation). In a UML diagram, the relationship between this «ComplementDatatype» and its complement data range is represented via a dependency stereotyped by «complementOf».

Stereotype and Base Class

«ComplementDatatype» with base class of UML::Datatype.

Parent

«DataRange».

Properties

None.

Constraints

None.

14.3.8.4 Datatype Facets

Description

As an alternative to representation of datatype facets using properties on a «DatatypeRestriction», the ODM provides a set of optional stereotypes that may be used to link explicitly modeled facet values («literal»s) to the restriction. Their definitions correspond to those provided for the equivalent properties defined on «DatatypeRestriction» in 14.3.8.5.

Stereotype and Base Class

«datatypeFacet», an abstract stereotype with base class of UML::Dependency.

«langRange», «length», «maxExclusive», «maxInclusive», «maxLength», «minExclusive», «minInclusive», «minLength», «pattern», all with base class of UML::Dependency and parent of *«datatypeFacet»*.

Parent

none for *«datatypeFacet»*, *«datatypeFacet»* for the others.

Properties

None.

Constraints

None.

14.3.8.5 Datatype Restrictions

Description

A datatype restriction consists of a datatype and n pairs of facets together with the appropriate facet values. The resulting data range is unary and is obtained by restricting the value space of the datatype according to the semantics of all of the pairs (multiple pairs are interpreted conjunctively). The constraining facet must be appropriate for the datatype, per the XML Schema specification.

Two approaches are taken to modeling datatype facets on restrictions in this profile: (1) using the properties provided below, or (2) by creating literals that specify the facet values and linking those to the datatype restriction via dependencies, as specified later in this sub clause. Figure 14.34 provides an example of how to construct datatype restrictions using the properties on the stereotype.

Stereotype and Base Class

«DatatypeRestriction» with base class of UML::Datatype.

Parent

«*DataRange*».

Properties

- langRange: String [0..1] - basic language ranges as specified in the OWL 2 specification (on string datatypes)
- length: String [0..1] - exact string length
- maxExclusive: String [0..1] - maximum exclusive value (on numeric datatypes)
- maxInclusive: String [0..1] - maximum inclusive value (on numeric datatypes)
- maxLength: String [0..1] - maximum string length
- minExclusive: String [0..1] - minimum exclusive value (on numeric datatypes)
- minInclusive: String [0..1] - minimum inclusive value (on numeric datatypes)
- minLength: String [0..1] - minimum string length
- pattern: String [0..1] - pattern expressions constraining string datatypes

Constraints

None.

14.3.8.6 Enumeration of Literals

Description

An enumeration of literals contains exactly the explicitly specified literals. In UML, the enumerated literals are specified as UML instance specifications or literal strings stereotyped by «literal», and linked to the DataEnumeration via a dependency stereotyped by «oneOf».

Note that enumeration literals are not used, due to the fact that their semantics in UML differ from the intended usage in OWL, in part because of the difficulty in providing alternative graphical notations that would use them (i.e., with the «oneOf» stereotype vs. as an enumeration) or for managing large numbers of them in the context of a single enumeration.

Stereotype and Base Class

«DataEnumeration» with base class of UML::Datatype.

Parent

«DataRange».

Properties

None.

Constraints

None.

14.3.8.7 Intersection Data Range

Description

An intersection data range contains all pairs of literals that are contained in each data range in the intersection. In UML, the pairs themselves are specified as data ranges that are linked to the «IntersectionDatatype» via a dependency stereotyped by «intersectionOf».

Stereotype and Base Class

«IntersectionDatatype» with base class of UML::Datatype.

Parent

«DataRange».

Properties

None.

Constraints

None.

14.3.8.8 Union Data Range

Description

A union data range contains all tuples of literals that are contained in at least one data range in the union. In UML, the pairs are specified as data ranges that are linked to the «UnionDatatype» via a dependency stereotyped by «unionOf».

Stereotype and Base Class

«UnionDatatype» with base class of UML::Datatype.

Parent

«DataRange».

Properties

None.

Constraints

None.

15 The Topic Map Profile

15.1 General

This clause defines a UML2 profile to support the usage of UML notation for the modeling of Topic Maps.

Note that the structure of Topic Maps differs considerably from UML, making the profiles somewhat misleading. UML specifies a class structure, with instances specified by a generic instances model. Topic Map constructs are largely at the individual level, in some cases gathered into classes via a type association.

In particular, the stereotype Topic extends UML Class. An instance of Class is itself a class, while an instance of Topic in the TM metamodel is generally not, although some instances of Topic serve as types for others. The profile only models instances of Topic which are types.

Further, the stereotype Association extends UML Association. An instance of UML Association specifies a set of tuples. An instance of TM Association specifies a particular link among particular individual instances of Topic. However, a TM Association is linked to an instance of Topic which serves as its type. So the stereotype models the instance of Topic which is the type of the TM Association.

15.2 Stereotypes

15.2.1 Topic Map

The Topic Map stereotype is defined as an extension of the UML Package base meta-class, as shown in Figure 15.1.

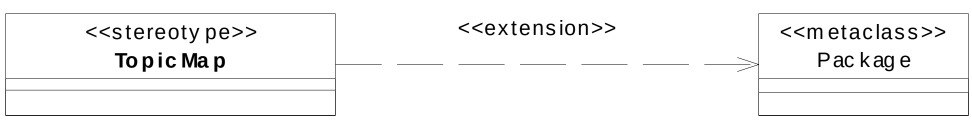

Figure 15.1 - Topic Map Stereotype

Applying this stereotype to a package requires that the UML constructs contained within the package be interpreted according to this profile definition.

Tagged Values

- itemIdentifiers - used to specify the storage location of the Topic Map, it must be a URI String.

15.2.2 Topic

The Topic stereotype extends the UML Class meta-class, as shown in Figure 15.2. Its application indicates that the UML Class is interpreted as a Topic which has been declared as a type which takes other instances of Topic as instances.

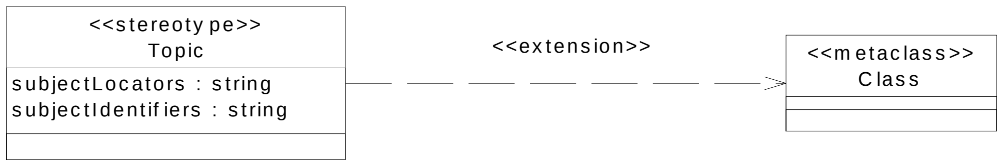

Figure 15.2 - Topic Stereotype

Tagged Values

- subjectIdentifiers - used to reference the Topic's subjectIdentifier, it must be a URI string.
- subjectLocators - used to reference the Topic's subjectLocators, it must be a URI string.
- itemIdentifiers - used to specify the storage location of the Topic Map, it must be a URI String.

15.2.3 Association

The association stereotype extends the UML Association meta-class, as shown in Figure 15.3. Its application indicates the UML Association is interpreted as a Topic Map Association. Both binary and n-ary associations are supported.

Note that the Topic Maps Association construct is an individual-level concept. The stereotype represents the topic which is the type of a set of instances of Association. An association in UML includes a set of ends which are instances of Property with associated Types. So, each instance of the Topic Map Association must be associated with the same pattern of instances of AssociationRole which have the same corresponding instances of Topic as type, corresponding to the UML association end properties. Further, each instance of Topic linked to an AssociationRole must be in a type-instance relationship with an instance of Topic corresponding to the UML type of the corresponding UML property.

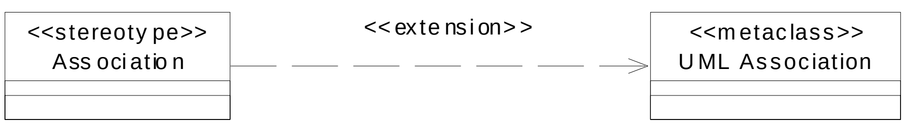

Figure 15.3 - Association Stereotype

Tagged Values

- itemIdentifiers - used to specify the storage location of the Association, it must be a URI String.

15.2.4 Characteristics

The abstract class Characteristic is not a concept from [TMDM] nor is it used in the ODM Topic Maps metamodel. It is used here to define a shared set of stereotypes that extend the UML Properties meta-class, as shown in Figure 15.4. All the metaclasses extending Characteristics are individual-level concepts, so that the stereotypes all represent topics which are either the type or scope of collections of instances.

Tagged Values

- name - used to specify the name of the Characteristic. Must be a String which can be a URI.
- datatype - used to specify the datatype of the Characteristic. Must be a String.
- value - used to specify the value of the Characteristic. Must be a String.

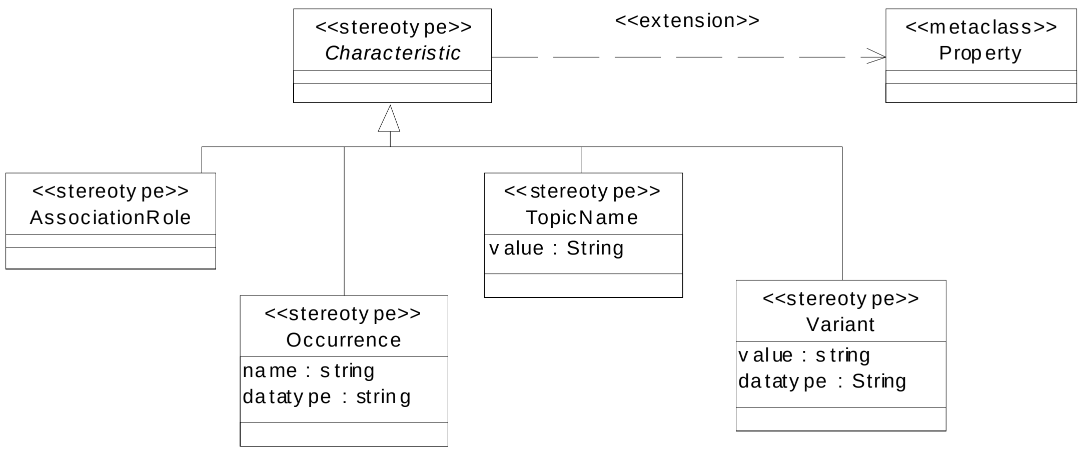

Figure 15.4 - Characteristic Stereotype

AssociationRole

The AssociationRole stereotype is used to indicate an UML Association owned end Property is a Topic Map AssociationRole. The owning UML Association must be stereotyped using the Association stereotype.The stereotype represents the instance of Topic which is the type of a collection of instances of AssociationRole.

Occurrence

The Occurrence stereotype may be applied to either a UML Attribute or a UML Association owned-end Property. Values of this property are interpreted as Topic Map Occurrence values. The property itself represents the topic which is the type of the collection of instances of Occurrence.

TopicName

The TopicName stereotype may be applied to char array or string typed attributes to indicate that values of the attribute represent Topic Names. The property itself represents the topic which is the type of the collection of instances of TopicName. The TopicName stereotyped Property may have multiple values for the tagged value 'variant' indicating a set of Variant stereotype Properties that are associated with this TopicName.

Variant

The Variant stereotype may be applied to any UML Property, including attributes and association ends, to indicate these values represent Variants. The property itself represents the topic which is the scope of the collection of instances of Variant. The Variant stereotype Property is required to have a tagged value 'parent' linking the variant to a parent TopicName.

15.3 Abstract Bases

Several abstract base meta-classes are defined in the profile. They purpose is to define shared tagged values for sets of stereotype meta-classes.

15.3.1 TopicMapElement

All stereotypes in the profile are specializations of the TopicMapELement abstract base class, as shown in Figure 15.5. This class provides all profile stereotypes with the 'itemIdentifiers' tagged value.

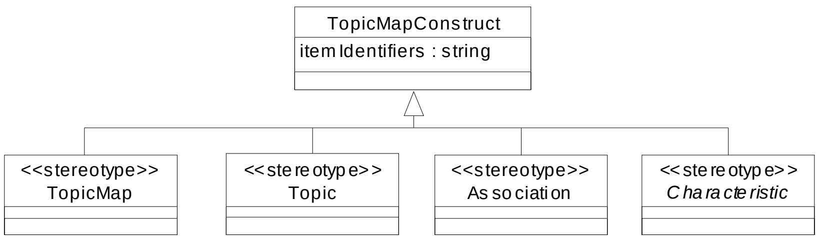

Figure 15.5 - TopicMapElement Stereotypes

Tagged Values

- itemIdentifiers - used to optionally provide an application specific unique identifier to the stereotyped elements; it must be a URI String.

15.3.2 Scoped Element

Some stereotyped elements in a profiled model, as shown in Figure 15.6, may have 'scope' tagged values that define when this scoped element is applicable.

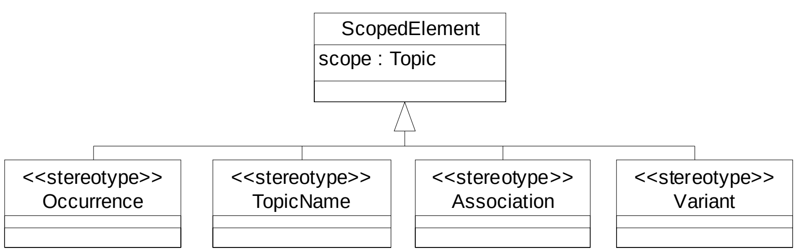

Figure 15.6 - ScopedElement Stereotypes

Tagged Values

- scope – a set of references to Topic Stereotyped elements that define the scope of the element.

15.3.3 TypedElement

Some stereotyped elements in the profiled model, as shown in Figure 15.7, may have a 'type' tagged value. This value references a Topic stereotyped class that defines the general nature of the owning element.

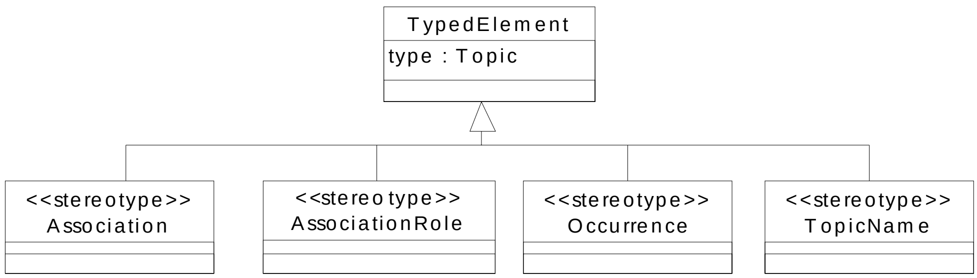

Figure 15.7 - TypedElement Stereotypes

Tagged Values

- type – a reference to a Topic stereotyped element that is the type of this element.

15.4 Example

Figure 15.8, show an example profile applied to a simple UML model of:

- A Personal Car is a Car, which may be owned by a Person.
- A Car is a Vehicle, which may have a Color.
- Carl is a person that owns one Personal Car that is red and another that is blue.

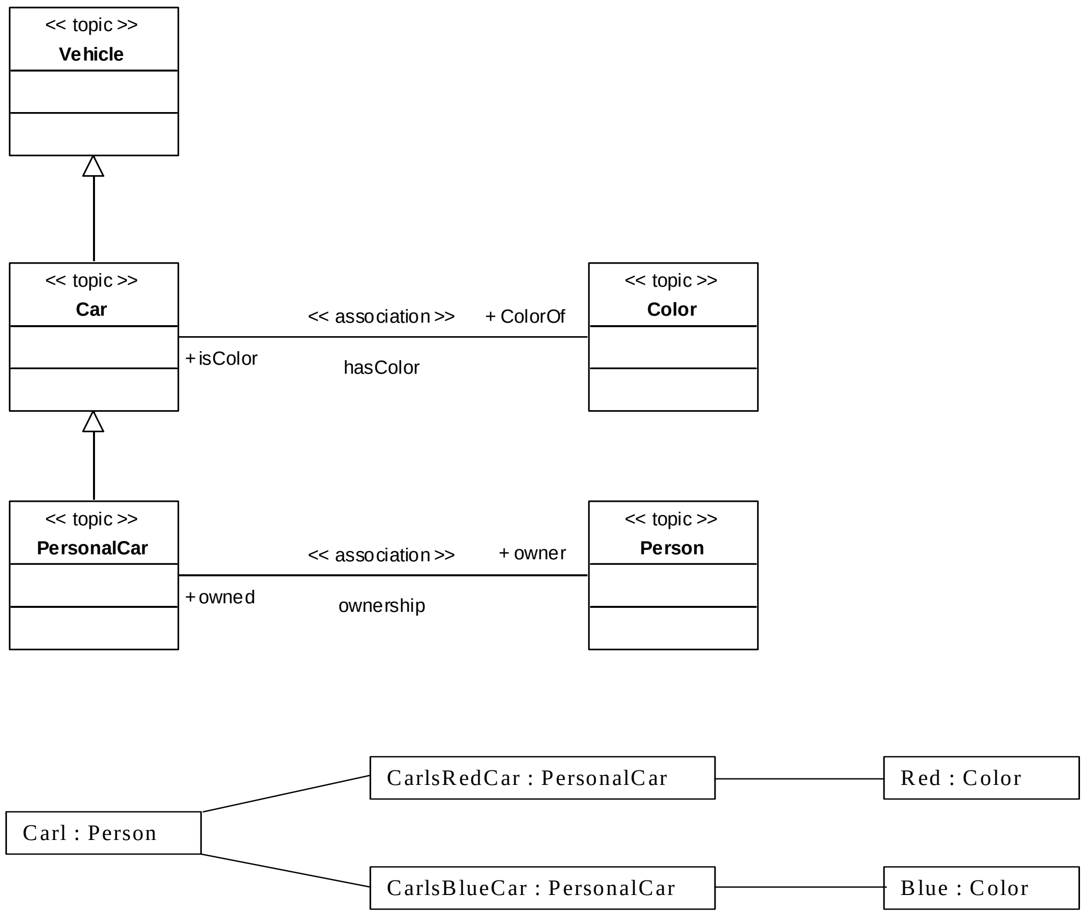

Figure 15.8 - Example Profile

16 Mapping Topic Maps to OWL

16.1 Overview

The mappings in this and the other mapping chapters of the ODM are expressed in the Relations language of QVT [MOF QVT]. A brief tutorial on this system is given in Annex H, MOF QVT: A Brief Tutorial.

The mappings in this clause are semantic mappings in terms of the W3C Semantic Web Best Practices and Deployment Group work-in-progress paper [RDF/TM], and where possible follow its suggestions. A Working Group Note or Recommendation is planned which may require that these mappings be revised during Finalization.

Note that the suggestions in [RDF/TM] are based on an earlier version of the Topic Maps Data Model than the ODM. The more recent version has many changes. Further, the mappings referred to in [RDF/TM] are all to RDF, whereas the mappings in the ODM are to OWL Full.

There is a significant structural mismatch between Topic Maps and OWL, which is at least partly mediated by the ODM OWL metamodel. In Topic Maps, as in UML, all objects are contained in packaging structures, which in Topic Maps is the metaclass TopicMap. In OWL, all objects are instances of the metaclass RDFSResource. In RDFS, a resource is conceived of as an object “in the world” which an ontology may make statements about. An ontology is a collection (graph) of statements, each of which refers to resources, but given a resource we can’t in principle navigate to the statements which refer to it. This is analogous to the fact that a web page may be linked to by many other web pages, but there is no reverse navigation. The site <omg.org> is the target of links from many sites, but there is no easy way for the OMG to know what those sites are.

However, a resource referred to in an OWL ontology is always an instance of one of the ODM OWL metaclasses. The declaration (in OWL) that a resource is an instance of one of the OWL metaclasses is a statement in a definite OWL ontology. It is possible to navigate from that declaration to the ontology it is asserted in. This situation is modeled in the OWL metamodel by the structures shown in Figure 11.8.

16.2 Topic Maps to OWL Full Mapping

16.2.1 Overview

The elements of the Topic Maps MOF meta-model are mapped into the OWL Full MOF metamodel as shown in the QVT statements below.

```
transformation TMapOntoSource  (tmap:TopicMapMetamodel, owlont:OWLMetamodel)
{
```

QVT needs to know how the various constructs are identified. The key statement can tell which model the class is in if the class’ name is unique within the two models.

16.2.1.1 RDFS/OWL Objects

```
key OWLUniverse(uriRef, ontology);
```

All objects in OWL (Full) are instances of Individual. But QVT does not support overlapping subtypes, so that the identifier is supplied by the common supertype of all OWL objects, OWLUniverse (Figure 11.8). The identifier URI is inherited from the superclass RDFSResource. URI is a unique identifier if the object has a URI. But in OWL the type of a resource depends on the ontology.

key OWLOntology(uriRef);

An OWL Ontology is identified by its base URI.

key OWLUniverse(nodeID, ontology);

A blank node does not have a URI. It is identified by a nodeID. The subclass BlankNode of OWLUniverse is defined by the presence of a nodeID attribute. The attribute nodeID is unique within a given repository population. It is the responsibility of the ontology server to main this uniqueness. But there is no requirement that the value of nodeID will be the same for different queries to the repository. Note that in OWL blank nodes occur only as restriction classes and the like, so are always used in declarations which are statements in a particular ontology.

key Statement(RDFSsubject, RDFSpredicate, RDFSobject, ontology);

16.2.1.2 Topic Map Objects

Topic Maps has a most general constuct, the abstract class TopicMapConstruct.

Topic map constructs have a complex system of identifiers. A construct has a possibly empty set of itemIdentifiers, each which is identifying. A topic has in addition a possibly empty set of subjectIdentifiers and a possibly empty set of subjectLocators, each of which is identifying. There is a constraint that an instance of TopicMapConstruct has at least one identifier, so that it is not valid for all of the properties to be empty.

Since different subclasses of TopicMapConstruct have different identification schemes, they are enumerated in the key statements. But Topic has several alternate keys. The following key declarations have the semantics that the first applies unless its property is empty. In that case, the second applies, and so on.

In each case the key is the set of property values.

```
key TopicMap(itemidentifiers);
key Topic(itemIdentifiers, parent);
key Association(itemIdentifiers, parent);
key TypeAble(itemIdentifiers, parent); //each subclass has a parent property
key Variant(itemIdentifiers, parent); //belongs to exactly one TopicName
key Topic(subjectIdentifiers); // alternative for Topic only
key Topic(subjectLocators); // alternative for Topic only
```

The basic packaging construct in Topic Maps is TopicMap, while the basic packaging construct in OWL is OWLOntology. In each case, the package can contain other packages. In TM, a TopicMap can have a reifer which belongs to another topic map, while in OWL, one OWLOntology can import another. Instance models of these are shown in Figure 16.1.

We assume the topic map to be transformed is normalized as follows:

- Topic maps are nested as a tree. Therefore there is a top topic map.
- The top topic map has an IRI among its identifiers. If not, one can be added.

Each topic map has an IRI among its itemIdentifiers. (If a topic map does not have an IRI among its identifiers, it can be merged with its parent.)

The dependency structure of the Topic Maps metamodel includes:

- Topic and Association depend on TopicMap (parent)
- Association depends on Topic (type) and TopicMap (parent)

- AssociationRole depends on Topic (player, type) and Association (parent)
- Occurrence depends on Topic (parent, type)
- TopicName depends on Topic (parent, type)
- Variant depends on TopicName (parent) and Topic (scope)

16.2.2 Packaging Construct: TopicMap

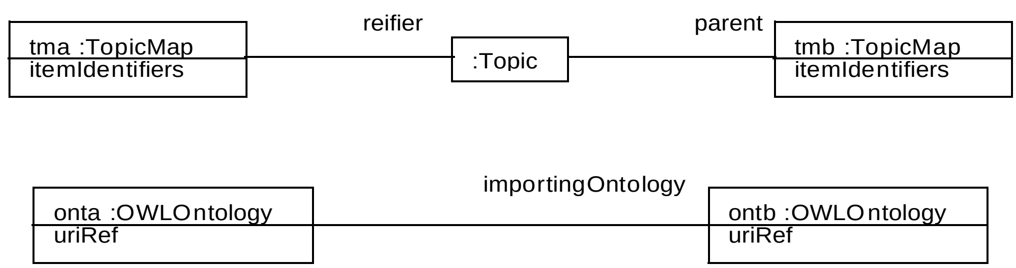

Figure 16.1 - Container structures in TM and OWL

The packaging construct TopicMap goes to OWLOntology. Topic Maps included in other topic maps are mapped to ontologies which are imported by other ontologies.

```
function TMiri(identifier : string) : boolean
// True if the identifier conforms to the syntax of a URI
{
// Details left to concrete implementations
} // TMiri

top relation TMapToOnto // Map topic map to ontology
{
    checkonly domain tmap tm:TopicMap{};
    enforce domain owlont ont:OWLOntology{};
    where{
        TopTMapToOnto(tm, ont);
        IntTMapToOnto(tm, ont);
    }
} //TMapToOnto

function TopTopicMap(tm:TopicMap): Boolean
// true if tm is a top topic map. Either no reifier or reifier is in the same topic map.
{
    if ((tm.reifer->IsEmpty) or
            (not tm.reifier.parent->exists(tmp | tmp <> tm)) then true
        else false
    endif;
} //TopTopicMap

relation TopTMapToOnto  //Map the top topic map to an ontology
{
    tmb : string;
    checkonly domain tmap tm:TopicMap {itemIdentifiers = tmb};
```

```
    enforce domain owlont ont:OWLOntology {uriRef = :URIReference{uri=:UniformResourceIdentifier{name=tmb}}};
    when {
        TopTopicMap(tm );
    }
} //TopTMapToOnto

relation IntTMapToOnto  //Map other than top topic map to an ontology
{
    tma : string;
    checkonly domain tmap tm:TopicMap {itemIdentifiers = tma,
        reifier = top: Topic {parent = tmb : TopicMap}};
    enforce domain owlont ont:OWLOntology {uriRef = :URIReference{uri=:UniformResourceIdentifier{name=tma}}
            importingOntology = ontb : OWLOntology};
    when {
        not TopTopicMap(tm);
        tm.reifer.parent->exists(tmb:TopicMap | TMapToOnto(tmb, ontb);
    }
} //IntTMapToOnto
```

16.2.3 Most General Structure: TopicMapConstruct

Instances of TopicMapConstruct can all map to instances of OWLUniverse. Mapping is constructed on demand when mapping more specific constructs.

```
relation TMCToOntoObjs
// Map links from topic map constructs in a TM to objects in an ontology.
// Modeled on AttributeToColumn [MOF QVT], p 172.
{
    tm : TopicMap;
    checkonly domain tmap tmc:TopicMapConstruct {};
    enforce domain owlont ind:OWLUniverse {uriRef = ref : URIReference,
        RDFtype = c : OWLClass{uriRef = :URIReference{uri = :UniformResourceIdentifier{name = 'owl:Thing'}}},
        ontology = ont:Ontology};
    when {
    // The semantics of QVT provide that there are multiple executions of these relations, each of which generates an
alternative binding for ref. The OR connector results in the three relations being executed in separate instantiations, each
generating a binding for ref.
        (TMCToOntoURIRef(tmc, ref) OR
        TopicSIToOntoRef(tmc, ref) OR // Applies only if tmc is a Topic
        TopicSLToOntoRef(tmc, ref)); // Applies only if tmc is a Topic
         not OWLUniverse.allInstances-> exists(i | i <> ind and TMCToOntoObjs(tmc, i));
    // prevents more than one target object from being constructed. So in this case, the target object is constructed with the
first instantiation of ref, which can come from any of the alternatives. See the key declarations for TopicMapConstruct.
        tm = TMCToTM(tmc);
        TMapToOnto (tm, ont); // Assumes ontolory target already exists. OK because relation is a top relation
    }
    where {
        MultItemIDsToSameAs(tmc, ind2); // Map the rest of identifiers to resources
    }
} // TMCToOntoObjs

function TMCToTM(tmc:TopicMapConstruct) : TopicMap
// Returns the topic map containing the topic map construct.
// Every topic map construct is either a Topic, Association, Occurrence, AssociationRole, TopicName, Variant or TopicMap
Figure 13.1, Figure 13.2.
```

```
{
      if tmc.oclIsTypeOf(Topic) or tmc.oclIsTypeOf(Association) then tmc.parent
      else if tmc.oclIsTypeOf(Occurrence) or tmc.oclIsTypeOf(AssociationRole) or
                  tmc.oclIsTypeOf(TopicName) then tmc.parent.parent
            else tmc.oclIsTypeOf(Variant) then tmc. parent.parent.parent
            else tmc // tmc is a Topic Map
            endif
      endif;
} // TMCToTm

relation IDtoURIRef
// if ident is a uri then return ident else make uri reference from base and ident
      checkonly domain tmap ident:string, base:string{};
      enforce domain owlont uriref: URIReference {};
      where {
            if TMiri(ident) then URItoURIRef(ident, uriref)
            else FragToURIRef(ident, base, uriref);
      }
} // IDtoURIRef

relation URItoURIRef
// string is already uri
checkonly domain tmap ident:string{};
enforce domain owlont uriref: UniformResourceIdentifier{name = ident};
} // URItoUIRRef

relation FragToURIRef
// construct uri from base and fragment
checkonly domain tmap ident:string, base:string {};
enforce domain owlont uriref : URIReference{uri = :UniformResourceIdentifier{name = base},
       fragmentIdentifier = :LocalName{name = ident}};
} // UFragToURIRef

top relation TMCToOntoURIRef
//Map item identifiers in a topic map construct to a URI reference in an ontology.
{
      base : string;
      tm: TopicMap;
      ont : OWLOntology;
      checkonly domain tmap tmc:TopicMapConstruct { }; //  Figure 13.1.
      enforce domain owlont ref:URIReference { };
      when {
            tm = TMCToTM(tmc);
            TMapToOnto(tm, ont);
      }
      where {
            base = ont.uriRef.uri.name;
            IDtoURIRef(tmc.itemIdentifiers, base, ref);
      }
} // TMCToOntoURIRef
```

16.2.4 Multiple Identifiers to SameAs

A topic map construct can have several instances of itemIdentifiers, all of which are identifying. A topic has also subjectIdentifiers and subjectLocators. Each gets mapped to a distinct Individual. The identifiers need to be tied together with SameAs. Subordinate to TopicMapConstruct.

```
relation MultItemIDsToSameAs
//Tie together distinct instances of itemIdentifers from an instance TopicMapConstruct with SameAs
{
    checkonly domain tmap tmc:TopicMapConstruct { }; // Figure 13.1.
    enforce domain owlont ind:OWLIndividual {uriRef = ref,
                OWLsameAs = ind1: OWLIndividual}; // Figure 11.6
    when {
    // The semantics of QVT provide that there are multiple executions of these relations, each of which generates an alternative binding for ref. The OR connector results in the three relations being executed in separate instantiations, each generating a binding for ref.
        (TMCToOntoURIRef(tmc, ref) OR
        TopicSIToOntoRef(tmc, ref) OR // Applies only if tmc is a Topic
        TopicSLToOntoRef(tmc, ref)); // Applies only if tmc is a Topic
        TMCToOntoObjs(tmc, ind1);
    // The target object is constructed with other than the first instantiation of ref, which can come from any of the alternatives. The generated object is identified with the first instantiation. See the key declarations for TopicMapConstruction.
    }
} // MultItemIDsToSameAs
```

16.2.5 Topic to OWL Class

Some topics are mapped to classes in OWL.

- Topics which are the type of an association but not a structural type
- Topics playing a type role in an association of type ‘type-instance’
- Either type in a subtype/ supertype relationship.

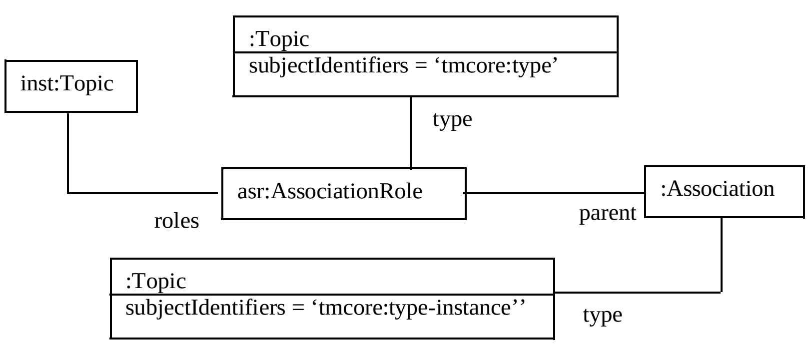

Figure 16.2 Type-instance structure in TM. Instances of Figure 13.2 and Figure 13.3

```
function TypeTopic(top: Topic) : Boolean
{
// True if a topic instance is considered a class, as an OCL expression on the TM metamodel.
    if ((top.typedConstruct ->exists(assoc: Association) and not StructuralType(top))
```

```
        // Topic is a type of an association, but not a structural type
        OR
    (top.roles->exists(asr | (asr.type.subjectIdentifiers =  'tmcore:type') and
        (asr.parent.type.subjectIdentifiers =  'tmcore:type-instance'))) OR
            // condition in Figure 16.2
    top.roles->exists(asr | asr.parent.type.subjectIdentifiers = 'tmcore:supertype-subtype')
// Topic is either a subtype or a supertype, therefore a type.
    ) then true
        else false
    endif;
} // TypeTopic

top relation TopicToClass // Map topic to a class
{
    checkonly domain tmap inst:Topic { };
    enforce domain owlont oc:OWLClass { };
    when {
        TypeTopic(inst);
    }
    where {
        TMCToOntoObjs(inst, oc);  // Create resource
    }
} // TopicToClass
```

16.2.6 Subtype to Subclass

The subtype-supertype relationship is mapped to the subclass relationship. Depends on Topic.

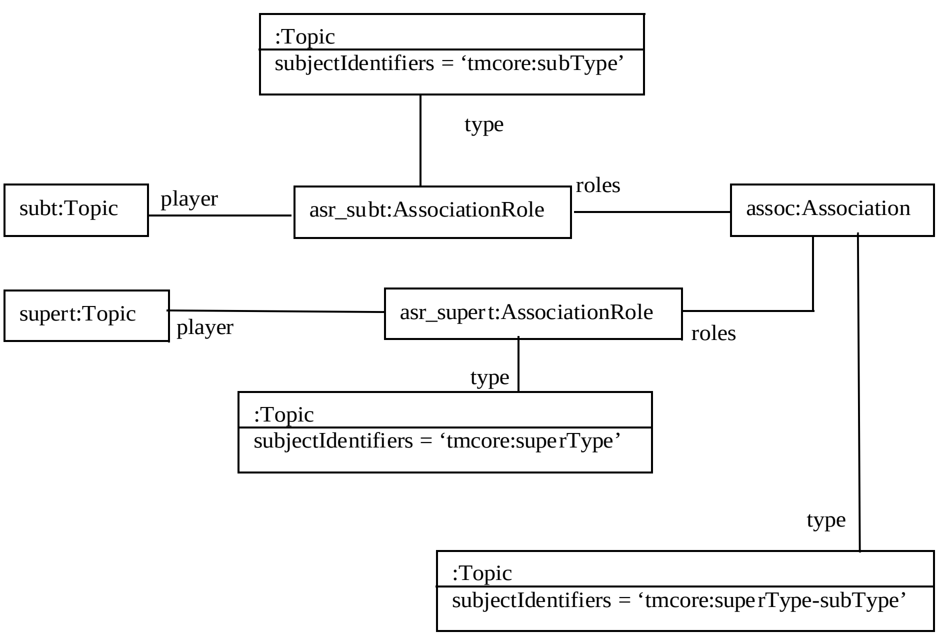

Figure 16.3 - Subtype-supertype structure in TM. Instance of Figure 13.2 and Figure 13.3

```
top relation ClassHierarchy // Map subtype-supertype to subclassOf (Figure 16.3)
{
    subID : string;
    checkonly domain tmap assoc:Association {};
    enforce domain owlont osubc : OWLClass {RDFSsubClassOf = osuperc:OWLClass};
    when {
    assoc.type.subjectIdentifiers->exists(subID | subID = 'tmcore:superType-subType');
    assoc.roles->exists(asr_subt | asr_subt.type.subjectIdentifiers = 'tmcore:subType' and
        asr_subt.player->exists(subt : Topic | assoc.roles->exists(asr_supert |
            asr_supert.type.subjectIdentifiers = 'tmcore:superType' and
                asr_supert.player-> exists(supert : Topic |
                    TopicToClass(subt, subc) and TopicToClass(supert, superc)))));
    }
} // ClassHierarchy
```

16.2.7 Topic to Property

A topic which is a type of an Occurrence, AssociationRole or TopicName is mapped to a property. Called from those constructs, so is not a top relation.

```
relation TopicToProperty // Map topic to a property
{
    checkonly domain tmap inst:Topic { };
    enforce domain owlont op:Property { };
    when {
        inst.typedConstruct->(exists(occ : Occurrence) or exists(ar : AssociationRole) or
            exists(n : TopicName));
    }
    where {
        TMCToOntoObjs(inst, op);  // Create resource
    }
} // TopicToProperty
```

16.2.8 Topic to Individual

A topic which is not a type is an individual-level object, so is mapped to an individual. The individual may belong to a class more specific than owl:Thing if the topic plays an ‘instance’ role in an association of type ‘type-instance.’ Depends on Topic (the individual’s type, if any).

```
top relation TopicToIndividual // Map topic to an individual.
{
    checkonly domain tmap inst:Topic { };
    enforce domain owlont oind:Individual { };
    when {
        not TypeTopic(inst);
    }
    where {
        TopicToTypedIndividual(inst, oind);
        TMCToOntoObjs(inst, oind);  // Create resource. Topic to untyped individual.
    }
} // TopicToIndividual
```

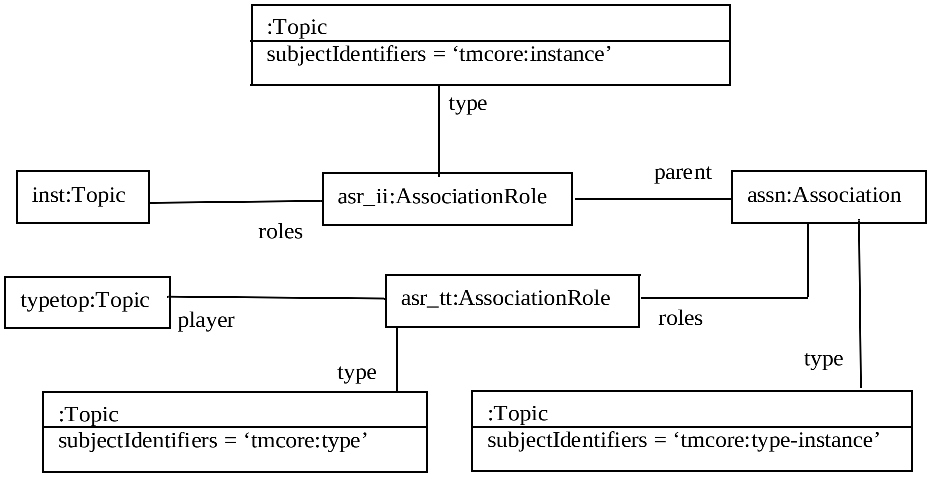

Figure 16.4 - Topic structure for typed Individual. Instances of Figure 13.2 and Figure 13.3

```
relation TopicToTypedIndividual // Map topic to a typed individual.
{
     checkonly domain tmap inst:Topic { };
     enforce domain owlont oind:Individual {RDFtype = topclass : OWLClass};
     when {
          inst.roles->exists(asr_ii : AssociationRole |
               (asr_ii.type.subjectIdentifiers =  ‘tmcore:instance) and
                    asr_ii.parent->exists(assn : Association | (assn.type.subjectIdentifiers =
                              tmcore:type-instance’) and
                         assn.roles->exists(asr_tt : AssociationRole | (asr_tt.type.
                              subjectIdentifiers = ‘tmcore:type’) and asr_tt.player->
                                   exists(typetop : Topic | TopicToClass(typetop, topclass)))));
     // Diagrammed in Figure 16.4
      //type topic is a class
     }
}  // TopicToTypedIndividual
```

16.2.9 Topic Subject Identifiers

A topic has subject identifiers which are identifying. They are mapped to URI References.

```
top relation TopicSIToOntoRef
//Map a subject identifier in a topic to URI reference in an ontology.
{
     base : string;
     tm: TopicMap;
     ont : OWLOntology;
     checkonly domain tmap top:Topic { }; // Figure 13.2.
     enforce domain owlont ref:URIReference { };
     when {
          tm = TMCToTM(top);
          TMapToOnto(tm, ont); //Ontology providing base URI
     }
     where {
```

```
        base = ont.uriRef.uri.name;
        IDtoURIRef(top.subjectIdentifiers, base, ref);
    }
} // TopicSIToOntoRef
```

16.2.10 Topic Subject Locators

A topic has subject locators which are mapped to resources. They are mapped to URI references.

```
top relation TopicSLToOntoRef
//Map a subject locator in a topic to URI reference in an ontology.
{
    base : string;
    tm: TopicMap;
    ont : OWLOntology;
    checkonly domain tmap top:Topic { }; // Figure 13.3.
    enforce domain owlont ref:URIReference { };
    when {
        tm = TMCToTM(top);
        TMapToOnto(tm, ont); //Ontology providing base URI
    }
    where {
        base = ont.uriRef.uri.name;
        IDtoURIRef(top.subjectLocators, base, ref);
    }
} // TopicSLToOntoRef
```

16.2.11 Association to Individual

An association is an individual-level object, more like an instance of an association class in UML. So an association is mapped to an OWL individual. All associations have a type, so the type is mapped to a class of which the individual is an instance. Depends on Topic.

An n-ary association is linked to n instances of AssociationRole, each of which is mapped to a property. Like the mapping of UML association classes, this strategy does not privilege binary associations. Binary associations in Topic Maps do not provide sufficient information to specify the directionality of an OWL property. On the other hand, this approach works for unary associations as well as any other.

Several association types are used for structural purposes, so are excluded from this mapping.

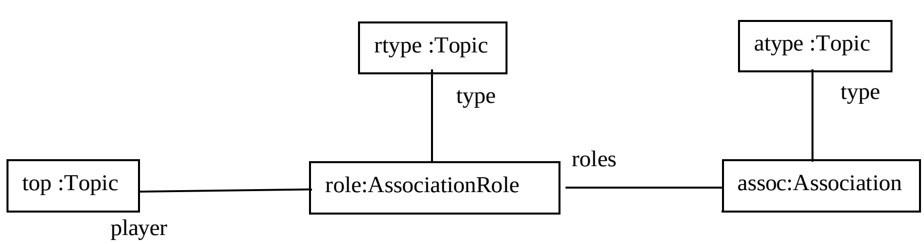

Figure 16.5 - Associations. Instance of Figure 13.2 and Figure 13.3

```
function StructuralType(top:Topic): Boolean
// True if topic is a structural type
{
```

```
	if top.subjectIdentifiers->
		exists(t | t = ‘tmcore:type-instance’ or t = ‘tmcore:superType-subType’)
	then true
	else false
	endif;
} // StructuralType

top relation AssocToTypedInd // Map association to a typed individual. Figure 16.5
{
	checkonly domain tmap assoc:Association {type = atype:Topic };
	enforce domain owlont op:Individual {RDFtype = aclass:OWLClass};
	when {
		not StructuralType(atype);
		TopicToClass(atype, aclass);
	}
	where {
		TMCToOntoObjs(assoc, op);  // Create resource
	}
} // AssocToTypedInd
```

16.2.12 Association Role to Property

Association roles are also individual-level objects, which have types. An association role must be linked both to a Topic and an Association, as in Figure 16.5. So an association role maps to a statement in RDF whose predicate is the result of a mapping of the association role’s type. Depends on Association and Topic.

Excluded are roles of structural associations.

```
top relation RoleToStatement // Map Association Roles to OWL statements.
{
	ont : Ontology;
	tm : TopicMap;
	checkonly domain tmap role:AssociationRole {
		parent = assoc : Association {type = atype : Topic},
		player  = top : Topic};
	enforce domain owlont ost:Statement
			{RDFSPredicate = prop : OWLObjectProperty,
			RDFSSubject = assocobj : Individual, RDFSObject = topobj : Individual,
			ontology = ont};
	when {
		not StructuralType(atype);
		AssocToTypedInd(assoc, assocobj);  // Created typed individual
		TopicToIndividual(top, topobj);  // Created typed individual
	}
	where {
		TopicToProperty(typetop, prop); // Create Property
		tm = TMCToTM(role); // Find ontology corresponding to AssociationRole
		TMapToOnto (tm, ont); // Assumes ontolory target already exists. OK because relation is a top relation

	}
} // RoleToStatement
```

16.2.13 Occurrence to Property

An occurrence is mapped also to a statement. An occurrence is linked to a topic and has a type, so the predicate is a property mapped from the type. If the occurrence datatype is ‘uri’ the property is an object property, otherwise a datatype property. The subject of the statement is mapped from the topic. Depends on Topic.

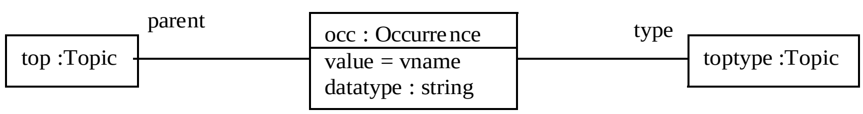

Figure 16.6 - Occurrence Instances. Instance of Figure 13.2 and Figure 13.3

```
top relation OccurInstToStatement // Occurrence to OWL statement. Figure 16.6
{
    ont : Ontology;
    tm : TopicMapl
    checkonly domain tmap occ:Occurrence {};
    enforce domain owlont ost:Statement {ontology = ont};
    where {
        OccurInstToObjStatement(occ, ost);
        OccurInstToDTStatement(occ, ost);
        tm = TMCToTM(occ); // Find ontology corresponding to Occurrence
        TMapToOnto (tm, ont); // Assumes ontolory target already exists. OK because relation is a top relation

    }
}  // OccurInstToStatement

relation OccurInstToObjStatement //Instance of object property statement
{
    turi : string;
    checkonly domain tmap occ:Occurrence {parent = top : Topic, type = toptype : Topic,
            datatype = ‘uri’, value = turi};
    enforce domain owlont ost:Statement {RDFSPredicate = op : OWLObjectProperty,
            RDFSSubject = sub : Individual,
            RDFSObject = obj : Individual {uriRef = :URIReference{uri = :UniformResourceIdentifier{name = turi}}}};
    when {
        TopicToIndividual(top, sub); // Created Individual
    }
    where {
        TopicToProperty(toptype, op); // Create Property
    }
}  // OccurInstToObjStatement

relation OccurInstToDTStatement //Instance of datatype property statement
{
    v, dt : string;
    checkonly domain tmap occ:Occurrence {parent = top : Topic, type = toptype : Topic };
    enforce domain owlont ost:Statement {RDFSPredicate = dtp : OWLDatatypeProperty,
            RDFSSubject = sub : Individual,
            RDFSObject = obj : PlainLiteral {lexicalForm = v}};
    when {
        occ.datatype->exists(dt | dt <> ‘uri’) and occ.value->exists(v);
        TopicToIndividual(top, sub); // Created Individual
```

```
    }
    where {
        TopicToProperty(toptype, dtp); // Create Property
    }
}  // OccurInstToDTStatement
```

16.2.14 Topic Names to Object Properties, Variants to Property Values

A topic name is a typed construct. It would be natural to map a topic name to a label, but a topic name can have variants, so must be able to be the subject of a property. TopicName is therefore mapped to an Individual, which has the value of the name as a label. Depends on Topic.

Variant names are mapped to values of a property whose uri is the Topic Map published subject 'tmcore:variant-name'. Some variants are resources while some are literals. But all variants have a scope, so the object of a variant is mapped to an Individual. The value of the variant whose datatype is not URI is mapped to a label of the object Individual. In this case the indiidual will be a blank node. The repository is responsible for assigning differentiating nodeID attributes to blank nodes.

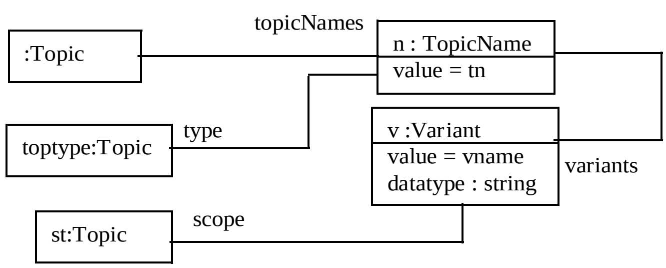

Figure 16.7 - Topic Name and Variant Name. Instances of Figure 13.2 and Figure 13.3

```
top relation TopicNamesToObjProp // Names go to object property statements. Figure 16.7
{
    tn : string;
    oind : Statement;
    ont : Ontology;
    tm : topicMap;
    checkonly domain tmap inst:Topic {topicNames = n:TopicName
        {value = tn, type = toptype}};
    enforce domain owlont ost:Statement {RDFSPredicate = op : OWLObjectProperty,
            RDFSSubject = sub : Individual,
            RDFSObject = obj : Individual {RDFSlabel = pl : PlainLiteral {lexicalForm = tn}},
            ontology = ont};
    when {
        TopicToIndividual(inst, sub); // Created Individual
    }
    where {
        TopicToProperty(toptype, op); // Create Property
        TMCToOntoObjs(n, obj); // Create Individual
        tm = TMCToTM(inst); // Find ontology corresponding to Topic
        TMapToOnto (tm, ont); // Assumes ontolory target already exists. OK because relation is a top relation
        VariantToObjProp(n, oind);
        VariantToDTProp(n, oind);
    }
```

```
} // TopicNamesToDTProp

relation VariantToObjProp
// Variant which is a resource to object property statement. Figure 13.2
{
    vname : string;
    sst : Statement;
    ont : Ontology;
    tm : TopicMap;
    checkonly domain tmap n:TopicName {
        variants = v:Variant {datatype = 'uri', value = vname}};
    enforce domain owlont ost:Statement {
        RDFSpredicate = op : OWLObjectProperty {uriRef = :UniformResourceIdentifier
            {name= 'tmcore:variant-name'}},
        RDFSsubject = sub : Individual,
        RDFSobject = obj : Individual{uriRef = :UniformResourceIdentifier{name = vname}},
        ontology = ont};
    when {
        TMCToOntoObjs(n, sub); // Created Individual
    }
    where {
        tm = TMCToTM(n); // Find ontology corresponding to TopicName
        TMapToOnto (tm, ont); // Assumes ontolory target already exists. OK because relation is a top relation
        TMCToOntoObjs(v, obj); // Create Individual
        VariantScope(v, sst : Statement);
    }
} // VariantToObjProp

relation VariantToDTProp
// Variant which is not a uri to object property statement whose object has the value as a label. Figure 13.2  Object will be
a blank node. Repository is responsible for assigning differentiating values of nodeID to blank nodes.
{
    vname : string;
    sst : Statement;
    ont : Ontology;
    tm : TopicMap;
    checkonly domain tmap n:TopicName {variants = v:Variant};
    enforce domain owlont ost:Statement {
        RDFSpredicate = op : OWLObjectProperty {uriRef = :UniformResourceIdentifier
            {name = 'tmcore:variant-name'}),
        RDFSsubject = sub : Individual,
        RDFSobject = obj : Individual {RDFSlabel = pl : PlainLiteral {lexicalForm = vname}},
        ontology = ont};
    when {
        v.datatype->exists(dt | dt <> 'uri') and
            v.value->exists(vname);
        TMCToOntoObjs(n, sub); // Created Individual
    }
    where {
        TMCToOntoObjs(v, obj); // Created Individual
        tm = TMCToTM(n); // Find ontology corresponding to TopicName
        TMapToOnto (tm, ont); // Assumes ontolory target already exists. OK because relation is a top relation
        VariantScope(v, sst : Statement);
    }
} // VariantToDTProp
```

16.2.15 Scope to Property Values

There is no concept in OWL comparable to the Topic Map concept scope, but we can map scope relationships to statements whose predicate is the uri ‘tmcore:scope.’ Depends on the subclasses of ScopeAble, Variant, and Topic.

```
top relation ScopeToStatement
{
	tm : TopicMap;
	ont : Ontology;
	checkonly domain tmap s:ScopeAble {scope = st : Topic};
	enforce domain owlont ost:Statement {
		RDFSpredicate = op : OWLObjectProperty {uriRef = :UniformResourceIdentifier{name = ‘tmcore:scope’}},
		RDFSsubject = sub : Individual,
		RDFSobject = obj : Individual,
		ontology = ont};
	when {
		TMCToOntoObjs(s, sub); // Created Individual
		TopicToIndividual(st, obj); // Created Individual
	}
	where {
		tm = TMCToTM(s); // Find ontology corresponding to ScopeAble object
		TMapToOnto (tm, ont); // Assumes ontolory target already exists. OK because relation is a top relation
	}
} // ScopeToStatement

relation VariantScope
// All variants have a scope. Mapped to a statement. Depends on Variant and Topic (scope).
{
	tm : TopicMap;
	ont : Ontology;
	checkonly domain tmap v:Variant {scope = st : Topic};
	enforce domain owlont ost:Statement {
		RDFSpredicate = op : OWLObjectProperty {uriRef = :UniformResourceIdentifier{name = ‘tmcore:scope’}},
		RDFSsubject = sub : Individual,
		RDFSobject = obj : Individual,
		ontology = ont};
	when {
		TMCToOntoObjs(v, sub); // Created Individual
		TopicToIndividual(st, obj); // Created Individual
	}
	where {
		tm = TMCToTM(v); // Find ontology corresponding to Variant
		TMapToOnto (tm, ont); // Assumes ontolory target already exists. OK because relation is a top relation
	}
} // VariantScope

} // transformation TMapOntoSource
```

16.3 OWL to Topic Maps

```
transformation OntoTMapSource  (owlont:OWLMetamodel, tmap:TopicMapMetamodel)
{
```

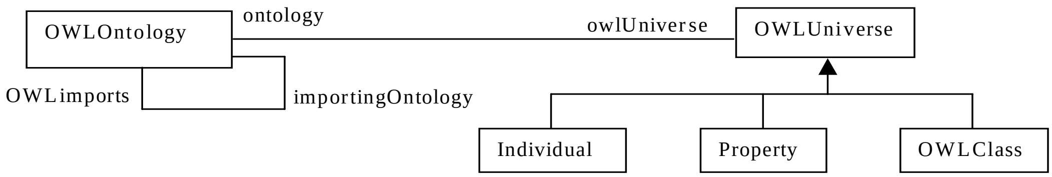

Figure 16.8 - Package structure of OWL

```
key OWLUniverse (uriRef, ontology);
// All objects in an OWL model instance are instances of OWLUniverse. Figure 16.8.
key TopicMap(itemidentifiers);
key Topic(itemIdentifiers, parent);
key Association(itemIdentifiers, parent);
key TypeAble(itemIdentifiers, parent); //each subclass has a parent property
key Variant(itemIdentifiers, parent); //belongs to exactly one TopicName
key Topic(subjectIdentifiers); // alternative for Topic only
key Topic(subjectLocators); // alternative for Topic only
```

16.3.1 Packaging Construct: OWLOntology

```
top relation OntoToTMap //Map ontology to a topic map.
{
    checkonly domain owlont ont:OWLOntology { };
    enforce domain tmap tm:TopicMap { };
    where {
        TopOntoToTMap(ont, tm);
        IntOntoToTMap(ont, tm);
    }
} // OntoToTmap

relation TopOntoToTMap  //Map top ontology to a topic map. Figure 16.1.
{
    tmi : string;
    checkonly domain owlont ont:OWLOntology {uriRef = :UniformResourceIdentifier{name = tmi}};
    enforce domain tmap tm:TopicMap {itemIdentifiers = tmi};
    when {
        ont.importingOntology->isEmpty;
    }
} //TopOntoToTMap

relation IntOntoToTMap  //Map imported ontology to a topic map. Figure 16.1.
{
    tmi : string;
    checkonly domain owlont ont:OWLOntology {uriRef = :UniformResourceIdentifier{name = tmi},
            importingOntology =  onta : OWLOntology};
    enforce domain tmap tm:TopicMap {itemIdentifiers = tmi,
                reifier = t:Topic {subjectLocators = tmi, parent = tma}};
    when {
        OntoToTmap(onta, tma);
    }
} //IntOntoToTMap
```

16.3.2 Class to Topic

Classes in OWL are identified by uri, except for restriction classes which are blank nodes. A class in Topic Maps is a topic which has a specific role in a specific association. See Figure 16.2.

```
top relation ClassToTopic
{
    checkonly domain owlont cl:OWLClass { };
    enforce domain tmap top:Topic {
        roles = asr : AssociationRole {
            type = tt : Topic {subjectIdentifiers = 'tmcore:type'}},
        parent = assn : Association {
            type = tc : Topic {subjectIdentifiers = 'tmcore:type-instance'}}
    };
    where {
        UClassToTopic(cl, top);
        RestrictionToTopic(cl, top);
    }
} // ClassToTopic
```

16.3.3 Class Identified by URI

```
relation UClassToTopic
// map an OWL class identified by uri to a topic.
{
    identifier : string;
    checkonly domain owlont cl:OWLclass {uriRef = :UniformResourceIdentifier{name = identifier},
        ontology = ont:OWLOntology};
// Note that an instance of OWLRestriction fails to have a uri, so is excluded
enforce domain tmap top:Topic {parent = tm, itemIdentifiers = identifier,
        subjectLocators = identifier};
    when {
        OntoToTMap(ont, tm);
    }
} // UClassToTopic
```

16.3.4 Restriction to Topic

```
relation RestrictionToTopic
// map an OWL restriction to a topic.
{
    ID : string;
    checkonly domain owlont res:OWLRestriction {nodeID = ID,
                ontology = ont:OWLOntology };
// Note that an instance of OWLRestriction is a blank node
    enforce domain tmap top:Topic {parent = tm, itemIdentifiers = ID,
        subjectLocators = ID};
    when {
        OntoToTMap(ont, tm);
    }
} // RestrictionToTopic
```

16.3.5 Individual to Topic

```
top relation IndividualToTopic
// map an OWL individual to a topic. An individual is an instance of at least one class, so the topic is linked to another topic
which is the class. See Figure 13.4.
{
    identifier : string;
    checkonly domain owlont ind:Individual {uriRef = :UniformResourceIdentifier{name = identifier},
        RDFType = cl:OWLClass, ontology = ont:OWLOntology};
    enforce domain tmap top:Topic {parent = tm, itemIdentifiers = identifier,
        subjectLocators = identifier
        roles = asri : AssociationRole{type = ti : Topic{subjectIdentifiers = ‘tmcore:instance’}
            parent = assn : Association
                {type = ta : Topic {subjectIdentifiers = ‘tmcore:type-instance’},
                    roles = asrt : AssociationRole
                        {type = ti : Topic {subjectIdentifiers = ‘tmcore:type}
                            player = cltop : Topic}
                }
        }
    } ;
    when {
        OntoToTMap(ont, tm);
        ClassToTopic(cl,cltop);
    }
} // IndividualToTopic
```

16.3.6 Hierarchy: RDFSsubclassOf

```
top relation SubclassToSubtype
// map the RDFSsubclassOf meta-association to a TM subclass/superclass relationship. See Figure 16.3.
{
    identifier : string;
    checkonly domain owlont subcl:OWLClass {uriRef = :UniformResourceIdentifier{name = identifier},
                        RDFSsubClassOf = supercl : OWLClass,
                        ontology = ont:OWLOntology};
    enforce domain tmap subcltop:Topic {
        roles = asrsub : AssociationRole{type =
            tsub : Topic{subjectIdentifiers = ‘tmcore:subType’},
            parent = assn : Association
                {type = ta : Topic {subjectIdentifiers = ‘tmcore:superType-subType’},
                    roles = asrsuper : AssociationRole
                        {type = tsuper : Topic {subjectIdentifiers = ‘tmcore:superType’},
                            player = supercltop : Topic}
                }
        }
    } ;
    when {
        OntoToTMap(ont, tm);
        ClassToTopic(subcl, subcltop);
        ClassToTopic(supercl, supercltop);
} // SubclassToSubtype
```

16.3.7 Object Property to Association Type

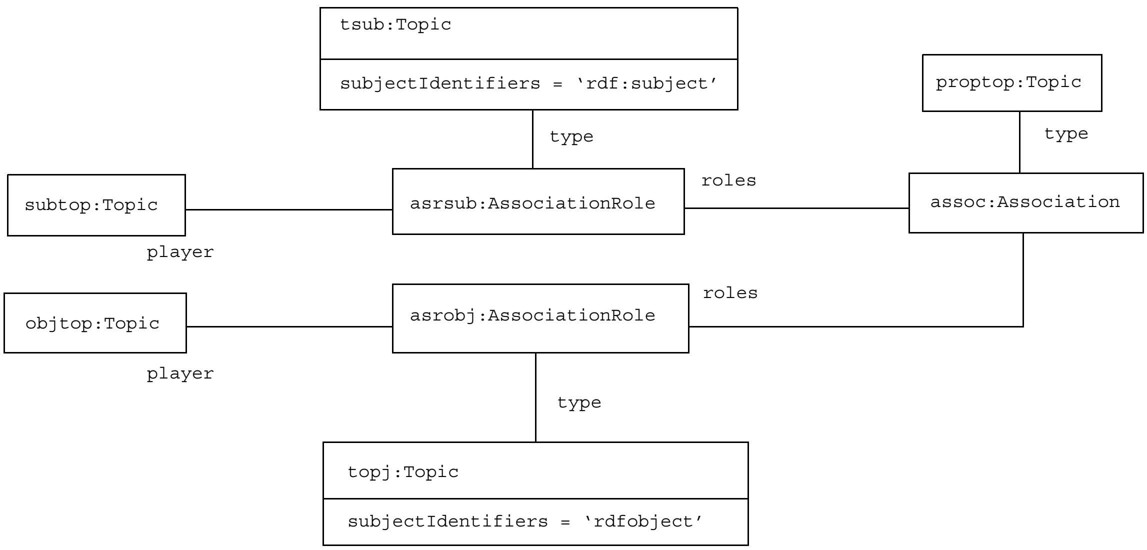

Figure 16.9 - Topic Map target of object property statement Figure 13.2 and Figure 13.3

```
top relation ObjPropToAssocType
// map an OWL object property to a topic intended to be the type of an association to which the property instance
statements are mapped. See Figure 16.9.
{
    identifier : string;
    checkonly domain owlont op:OWLObjectProperty {uriRef = :UniformResourceIdentifier{name = identifier},
                              ontology = ont:OWLOntology};
    enforce domain tmap top:Topic {parent = tm, itemIdentifiers = identifier,
        subjectLocators = identifier};
    when {
        OntoToTMap(ont, tm);
    }
} // ObjPropToAssocType
```

16.3.8 Object Property Instance Statement to Association Instance

This mapping is very different from the mapping of associations to object properties. It gives a much better representation of an object property, but with very specialized types of association roles. The types are taken from RDF.

```
top relation ObjPropStateToAssocInst
// Map an OWL statement whose predicate is an object property to an instance of an association. See Figure 16.9.
{
    ID, asrsub, asrob : string;
    checkonly domain owlont os:Statement {ontology = ont:OWLOntology,
        RDFSsubject = sub: Individual, RDFSpredicate = prop: OWLObjectProperty,
        RDFSobject = obj : Individual};
    enforce domain tmap assoc : Association {parent = tm, itemIdentifiers = ID,
            type = proptop : Topic,
            roles = : AssociationRole {itemIdentifiers =asrsub, type = : Topic
                                                {subjectIdentifiers = 'rdf:subject'},
                                            player = subtop : Topic}
```

```
            roles = : AssociationRole {itemIdentifiers =  asrobj, type = : Topic
                                                        {subjectIdentifiers = 'rdf:object'},
                                                    player = objtop : Topic}
        } ;
    when {
        ObjPropToAssocType(prop, proptop);
        IndividualToTopic(sub, subtop);
        IndividualToTopic(obj, objtop);
    }
    where {
    ID = genTMCItemidentifier(tm);
    asrsub = genTMCItemidentifier(tm);
    asrobj = genTMCItemidentifier(tm);
    }
} // ObjPropStateToAssocInst

function genTMCItemidentifier(tm : TopicMap) : string
// Generates a string which can be used as itemIndentifiers of a topic map construct, unique to topic map tm
{
} // genTMCItemidentifier
```

16.3.9 Datatype Property to Occurrence

```
top relation DTPropToOccurType
// map an OWL datatype property to a topic intended to be the type of an occurrence to which the property instance
statements are mapped. See Figure 16.6.
{
    identifier : string;
    checkonly domain owlont dtp:OWLDataTypeProperty {uriRef = :UniformResourceIdentifier{name = identifier},
                                ontology = ont:OWLOntology};
    enforce domain tmap top:Topic {parent = tm, itemIdentifiers = identifier,
        subjectLocators = identifier};
    when {
        OntoToTMap(ont, tm);
    }
} // DTPropToOccurType
```

16.3.10 Datatype Property Instance Statement to Occurrence

```
top relation DTPropStateToOccurrence
// Map an OWL statement whose predicate is a datatype property to an instance of an occurrence. See Figure 16.6.
{
    ID, dt , lexf : string;
    checkonly domain owlont os:Statement {ontology = ont:OWLOntology,
        RDFSsubject = sub: Individual, RDFSpredicate = prop: OWLDatatypeProperty,
        RDFSobject = obj : Literal{ lexicalForm = lexf}};
    enforce domain tmap occ : Occurrence {parent = subtop, itemIdentifiers = ID,
            type = proptop : Topic, value = lexf, datatype = dt} ;
    when {
        DTPropToOccurType(prop, proptop);
        IndividualToTopic(sub, subtop);
    }
    where {
        dt = datatypeOf(obj);
        ID = genTMCItemidentifier(tm);
```

```
    }
} // DTPropStateToOccurrence

function datatypeOf(obj : Literal) : string
{
    if obj.datatypeURI->exists(dt : string) then dt
        else ‘’
    endif;
} // datatypeOf
```

16.3.11 SameAs, EquivalentClass, EquivalentProperty

Individuals, classes and properties all map to Topics. SameAs, equivalentClass and equivalentProperty therefore all map to assertions that two topics are the same. Topic Maps has a normative procedure for merging topics.

```
relation mergeTopics
{
    // as specified in [TMDM]
} // mergeTopics

top relation SameAsToMerge
{
    checkonly domain owlont ind:Individual {OWLsameAs = ind1 : Individual};
    when {
        IndividualToTopic(ind, top);
        IndividualToTopic(ind1, top1);
    }
    where {
        mergeTopics(top, top1);
    }
} // SameAsToMerge

top relation EquivClassToMerge
{
    checkonly domain owlont cl : OWLClass {OWLequivalentClass = cl1 : OWLClass};
    when {
        ClassToTopic(cl, top);
        ClassToTopic(cl1, top1);
    }
    where {
        mergeTopics(top, top1);
    }
} // EquivClassToMerge

top relation EquivPropToMerge
{
    checkonly domain owlont prop:Property {OWLequivalentProperty = prop1 : Property};
    when {
        PropertyToTopic(prop, top);
        PropertyToTopic (prop1, top1);
    }
    where {
        mergeTopics(top, top1);
    }
} // EquivPropToMerge
```

```
relation PropertyToTopic
// Property is an abstract class, so any instance is an instance of one of its subtypes
{
	checkonly domain owlont prop:Property { };
	enforce domain tmap top : Topic { };
	when {
		(ObjPropToAssocType(prop, top) OR
			DTPropToOccurType(prop, top));
	}
} // PropertyToTopic

} // transformation OntoTMapSource
```

17 Mapping RDFS and OWL to CL

17.1 Overview

Mapping from the W3C Semantic Web languages, the Resource Description Framework [RDF Primer] [RDF Concepts] and the Web Ontology Language [OWL S&AS] to Common Logic (CL) is relatively straightforward, as per the draft mapping under development by Pat Hayes [SCL Translation] for incorporation in ISO 24707 [ISO 24707]. The mapping supports translation of RDF vocabularies and OWL ontologies from the RDFS and OWL metamodels, respectively, to the CL metamodel, in the spirit of the language mapping. Users are encouraged to familiarize themselves with the original translation specification and to recognize that the overarching goal is to preserve not only the abstract syntax of the source languages but their underlying semantics, such that CL reasoners can accurately represent and reason about content represented in knowledge bases that reflect those models. The mapping, as it stands, is intended to take an RDFS/OWL ontology as input and map it directly to CL from the triple format. Additional work, including (1) a direct mapping from an RDFS/OWL ontology represented solely in a UML/MOF environment using the metamodels and profiles contained herein, (2) representation of the mappings using MOF QVT, (3) a lossy, reverse mapping from CL to RDFS/OWL using MOF QVT to preserve lossy information, and (4) bi-directional mappings from CL to and from UML 2, again using MOF QVT to preserve lossy information, are planned.

Note that we have not attempted to address the issues raised in [SCL Translation] regarding the distinction between an embedded or translation approach to determining how to map language constructs – such decisions are left to the vendor, depending on the target application(s).

17.2 RDFS to CL Mapping

The separation between RDF and RDF Schema given in [SCL Translation] is not maintained in the ODM, which supports RDF Schema by design. As discussed in the Design Rationale, maintaining that separation from a MOF/UML perspective did not make sense, since (1) it is difficult, at best, to separate the abstract syntax of RDF from that of RDF Schema, and (2) the goal of ODM is to support ontology definition in MOF and UML tools, which is most commonly done using RDF Schema, OWL, or another knowledge representation language. Basic RDF graphs can be translated to CL using the mapping described herein, however.

17.2.1 RDF Triples

Any simple RDF triple (expressed as *subject predicate object*), can be embedded in CL as `(rdf_triple subject predicate object)`[1] and/or translated to an CL atomic sentence directly `(predicate subject object)`[2]. These mappings can be expressed in terms of the metamodel elements shown in Table 17.1.

Table 17.1 - RDF Triple to CL Mapping

RDFS Metamodel Element	RDFS Metamodel Property	CL Metamodel Element	CL Metamodel Property
RDFStatement		Relation[1]	`predicate: rdf_triple`
	RDFsubject		`arguments [1]`
	RDFpredicate		`arguments [2]`
	RDFobject		`arguments [3]`

Table 17.1 - RDF Triple to CL Mapping

RDFStatement	RDFpredicate	Relation[2]	`predicate`
	RDFsubject		`arguments [1]`
	RDFobject		`arguments [2]`

These two approaches are completely compatible, and the relationship between them can be expressed through the axiom:

```
(forall (x y z)(iff (rdf_triple y x z)(x y z)))
```

The translation extends this notion further through to ensure that the predicate expressed by the triple is indeed a valid RDF property, the "cautious translation approach."

RDF Property Axiom

```
(forall (x y z)(iff (rdf_triple y x z)(and (rdf:Property x)(x y z))))
```

RDF Promiscuity Axiom

```
(forall (x)(rdf:Property x))
```

For the purposes of this specification, any RDF or RDFS predicate that is not explicitly mapped to CL can be translated directly using this method.

17.2.2 RDF Literals

Literals in RDF can be defined as either "plain literals" or "typed literals," corresponding to classes of the same names in the RDFS metamodel. Plain literals translate into CL quoted strings, possibly paired with a language tag, and in both RDF and CL they are understood to denote themselves. The function `stringInLang` is used to indicate the pair consisting of a plain literal and a language tag. Typed literals in RDFS and OWL have two parts: a character string representing the lexical form of the literal, and a datatype name that indicates a function from a lexical form to a value. In RDFS/OWL these two components are incorporated into a special literal syntax; in CL, the datatype is represented as a function name applied to the lexical form as an argument. Table 17.2 provides the corresponding metamodel mappings.

17.2.3 RDF URIs and Graphs

URIs and URI references can be used directly as CL names. Blank nodes in an RDF graph translate to existentially bound variables with a quantifier whose scope is the entire graph. A graph is the conjunction of the triples it contains. Basic translation for the corresponding metamodel elements is given in Table 17.2.

Table 17.2 - Basic RDF to CL Mapping

RDFS Metamodel Element	RDFS Metamodel Property	CL Metamodel Element	CL Metamodel Property
RDFSResource	URI reference *"aaa"* –or– namespace and local name *"aaa"* –or– blank node ID *"_:aaa"*	LogicalName	`string:` *aaa*
PlainLiteral	lexicalForm: *"aaa"*	SpecialName	`specialNameKind: quotedString, string:` *aaa*

Table 17.2 - Basic RDF to CL Mapping

PlainLiteral	lexicalForm: *"aaa"* languageTag: *"tag"*	Function	`operator: stringInLang, arguments [1]: aaa, arguments [2]: tag`
TypedLiteral	lexicalForm: *"aaa"* datatypeURI: *"ddd"*	Function	`operator: ddd arguments [1]: aaa`
RDFDescription	contains: RDFSResource	CLModule	`CLText: Phrase`
RDF graph (set of triples) $\{ttt_1,...,ttt_n\}$		Sentence	(exists($bbb_1...bbb_m$) (and ttt_1 ... ttt_n) where _: bbb_1 ... _: bbb_m are all the blank node IDs in the graph.

For example, the RDF graph

```
_:x ex:firstName "Jack"^^xsd:string .
_:x rdf:type ex:Human .
_:x Married _:y .
_:y ex:firstName "Jill"^^xsd:string .
```

maps into the CL sentence:

```
(exists (x y)(and
    (ex:firstName x (xsd:string 'Jack'))
    (rdf:type x ex:Human)
    (Married x y)
    (ex:firstName y (xsd:string 'Jill'))
))
```

The RDF vocabularies for reification, containers and values have no special semantic conditions, so translate uniformly into CL using the above conversion methods.

17.2.4 RDF Lists

[SCL Translation] includes a discussion relevant to both RDFS and OWL ontologies regarding the mapping of lists that represent relations between multiple arguments to CL. Since RDF triple syntax can directly express only unary and binary relations, relations of higher arity must be encoded, and OWL in particular uses lists to do this encoding. Axioms for translating such lists, derived from [Fikes & McGuinness], are provided in [SCL Translation] and are incorporated herein by reference.

17.2.5 RDF Schema

As discussed in [SCL Translation], RDF Schema extends RDF through semantic constraints that impose additional meaning on the RDFS vocabulary. In particular, it gives a special interpretation to *rdf:type* as being a relationship between a 'thing' and a 'class,' which approximates the set-membership relationship in set theory. This relationship is captured in several axioms, repeated here due to their importance with regard to streamlining the mapping.

RDFS Class Axiom

```
(forall (x y)(iff (rdf:type x y) (and (rdfs:Class y)(y x))))
```

RDFS Promiscuity Axiom

```
(forall (x) (rdfs:Class x))
```

RDFS Universal Resource Axiom

```
(forall (x) (rdfs:Resource x))
```

Taken together, these axioms justify the more efficient mapping of RDFS triples to CL given in Table 17.3, to be used in place of Table 17.1.

Table 17.3 - RDFS Triple to CL Mapping

RDFS Metamodel Element	RDFS Metamodel Property	CL Metamodel Element	CL Metamodel Property
(1) RDFStatement	RDFsubject *(aaa)* RDFpredicate *(rdf:type)* RDFobject *(bbb)*	Relation	`predicate:` *bbb* `arguments [1]:` *aaa*
(2) RDFStatement, (any other triple)	RDFsubject *(aaa)* RDFpredicate *(ppp)* RDFobject *(bbb)*	Relation	`predicate:` *ppp* `arguments [1]:` *aaa* `arguments [2]:` *bbb*

The translations are ordered, with the second used only when the first does not apply.

The above example now translates into the more intuitive form

```
(exists (x y)
    (and
    (ex:firstName x (xsd:string 'Jack'))
    (ex:Human x)
    (Married x y)
    (ex:firstName y (xsd:string 'Jill'))
))
```

where `ex:Human` is a genuine predicate.

Similarly to the case for RDF, this assumes that *every* unary predicate corresponds to an RDFS class; to be more cautious, one would omit the promiscuity axiom and insert an extra assumption explicitly as part of the translation process: if (1), add axiom `(rdfs:Class bbb)`; otherwise, (2) add axiom: `(rdf:Property ppp)`.

17.2.6 RDFS Semantics

In [RDF Semantics], several of the constraints are expressed as RDFS assertions (“axiomatic triples”), but others are too complex to be represented in RDFS and so must be stated explicitly as external model-theoretic constraints on RDFS interpretations. All of these can be expressed directly as CL axioms, however. A CL encoding of RDFS is obtained by following the translation rules and adding a larger set of axioms. RDFS interpretations of a graph can be identified with CL interpretations of the translation of the graph with the RDF and RDFS axioms added.

A series of tables encoding numerous axioms is provided in [SCL Translation] which reflect the axiomatic triples, RDFS “semantic conditions,” and extensional axioms, as well as axioms for interpreting datatypes, which are incorporated herein by reference. Some of these are summarized in an RDFS extensional logical form translation table, which may be more efficient than deriving the translation from the embedding and axioms.

These are provided in Table 17.4, mapped to the appropriate metamodel elements.

Table 17.4 - RDFS Extensional Logical Form Translation

RDFS Metamodel Element	RDFS Metamodel Property	CL Metamodel Element	CL Metamodel Property	'Cautious' Axiom(s)
RDFStatement	RDFsubject *(aaa)* RDFpredicate (*rdf:type)* RDFobject *(bbb)*	Relation	`predicate:` *bbb* `arguments [1]:` *aaa*	`(rdfs:Class bbb)`
RDFStatement	RDFsubject *(aaa)* RDFpredicate (*rdfs:domain)* RDFobject *(bbb)*	Universal Quantification	`Binding: Term:` *u* `Term:` *y*	`(rdfs:Class bbb)` `(rdf:Property aaa)`
		Implication	`antecedent: (`*aaa u y*`)` [3] `consequent: (`*bbb u*`)`	
		[3]Relation	`predicate:` *aaa* `arguments [1]:` *u* `arguments [2]:` *y*	
		Relation	`predicate:` *bbb* `arguments [1]:` *u*	
RDFStatement	RDFsubject *(aaa)* RDFpredicate (*rdfs:range)* RDFobject *(bbb)*	Universal Quantification	`Binding: Term:` *u* `Term:` *y*	`(rdfs:Class bbb)` `(rdf:Property aaa)`
		Implication	`antecedent: (`*aaa u y*`)` `consequent: (`*bbb y*`)`	
		Relation	`predicate:` *aaa* `arguments [1]:` *u* `arguments [2]:` *y*	
		Relation	`predicate:` *bbb* `arguments [1]:` *y*	
RDFStatement	RDFsubject *(aaa)* RDFpredicate (*rdfs:subClassOf)* RDFobject *(bbb)*	Universal Quantification	`Binding: Term:` *u*	`(rdfs:Class bbb)` `(rdfs:Class aaa)`
		Implication	`antecedent: (`*aaa u*`)` `consequent: (`*bbb u*`)`	
		Relation	`predicate:`*aaa* `arguments [1]:` *u*	
		Relation	`predicate:` *bbb* `arguments [1]:` *u*	
RDFStatement	RDFsubject *(aaa)* RDFpredicate (*rdfs:subPropertyOf)* RDFobject *(bbb)*	Universal Quantification	`Binding: Term:` *u* `Term:` *y*	`(rdf:Property bbb)` `(rdf:Property aaa)`

Table 17.4 - RDFS Extensional Logical Form Translation

		Implication	antecedent: (*aaa u y*) consequent: (*bbb u y*)	
		Relation	predicate: *aaa* arguments [1]: *u* arguments [2]: *y*	
		Relation	predicate: *bbb* arguments [1]: *u* arguments [2]: *y*	
RDFStatement (any other triple)	RDFsubject *(aaa)* RDFpredicate *(ppp)* RDFobject *(bbb)*	Relation	predicate: *ppp* arguments [1]: *aaa* arguments [2]: *bbb*	(rdf:Property ppp)

Where possible clauses included in sentences, such as the antecedent and consequent of an implication, are expanded for further clarification. The translations are ordered, with the final one used only when the others do not apply.

17.3 OWL to CL Mapping

As described in the relevant specifications, the Web Ontology Language (OWL) is actually three closely related dialects rather than a single language, which share a common set of basic definitions but differ in scope and by the degree to which their syntactic forms are restricted. The OWL metamodel given in Clause 11 of this specification is intended to represent the abstract syntax for OWL Full, but can also represent the abstract syntax for OWL DL, as long as restrictions to support the more constrained semantics of OWL DL are applied.

The discussion provided in [SCL Translation] provides additional insight into the variations among OWL dialects. It then provides an unrestricted translation from the OWL vocabulary to CL, and further refines it for each dialect given a common starting point. There are a number of important considerations provided in that discussion, including a series of axioms applicable to any CL reasoning environment designed to support OWL ontologies as input.

Table 17.5 provides a summary translation from RDFS/OWL triples, as represented in the metamodel triple constructs, mapped to the appropriate high-level CL metamodel sentence constructs. We've taken this approach in keeping with the translation, but also due to the fact that what is mapped in some cases is actually a subgraph consisting of multiple RDFS/OWL statements as well as for increased clarity. Further refinement of some of the CL sentences will be accomplished during the finalization phase of the specification, along with inclusion of examples. The translation assumes the axioms stated in 17.1 and 17.2, as well as the following identity axioms:

```
(forall ((x owl:Thing)(y owl:Thing))(iff (owl:differentFrom x y)(not (= x y)) ))

(forall ((x owl:Thing))(not (owl:Nothing x)))
```

Note that OWL assertions involving annotation and ontology properties are not covered explicitly, and should be simply transcribed as atomic assertions in CL, using the same mechanisms described for RDF triples.

To use the table below to translate an OWL/RDF graph, simply generate the corresponding CL for every subgraph that matches the pattern specified in the leftmost two columns. The notation `ALLDIFFERENT` is used as a shorthand for conjunction of n(n-1) "inequations" which assert that the terms are all distinct:

```
[ALLDIFFERENT x1 ... xn]
```

means:

```
(and
```

```
            (not (= x1 x2)) (not (=x1 x3)) ... (not (= x1 xn))
            (not (= x2 x3)) ... (not (= x2 xn))
            (not (= x3 xn)) ...
            ...
            (not (= xn-1 xn))
        )
```

Note that the negation of this is a disjunction of equations. *owl_Property* should be read as shorthand for the union of *owl:DatatypeProperty* and *owl:ObjectProperty.*

Unlike the RDFS translation, this translates entire RDF subgraphs into logical sentences. To achieve a full translation, *all* matching subgraphs must be translated, and then any remaining triples rendered into logical atoms using the RDF translation. Note that a triple in the graph may occur in more than one subgraph; in particular, the *owl:onProperty* triples will often occur in several subgraph patterns when cardinality and value restrictions are used together.

Table 17.5 - RDFS/OWL to CL Metamodel Translation

RDFS/OWL Metamodel Element	RDFS/OWL Metamodel Property	CL Metamodel Element	CL Metamodel Property	Assumption(s)
Subgraph: RDFStatement RDFStatement	 RDFsubject *(rrr)* RDFpredicate *(owl:onProperty)* RDFobject *(ppp)* RDFsubject *(rrr)* RDFpredicate *(owl:minCardinality)* RDFobject *(n)*	Universal Quantification Implication	`Binding: (Term:` *(x owl:Thing))* `antecedent:` *(rrr x)* `consequent:` *(exists ((*x_1 *owl:Thing) ... (*x_n *owl:Thing)) (and* *[ALLDIFFERENT* x_1 *...* x_n*]* *(ppp x* x_1*) ...(ppp x* x_n*)* *))*	*(owl:Restriction rrr)* *(rdf:Property ppp)*
subgraph: RDFStatement RDFStatement	 RDFsubject *(rrr)* RDFpredicate *(owl:onProperty)* RDFobject *(ppp)* RDFsubject *(rrr)* RDFpredicate *(owl:maxCardinality)* RDFobject *(n)*	Universal Quantification Implication	`Binding:` *(* *Term:(x owl:Thing)* *Term:(*x_1 *owl:Thing)* *...* *Term: (*x_{n+1} *owl:Thing)* *)* `antecedent:` *(and (rrr x)* *(ppp x* x_1*) ...(ppp x* x_{n+1}*)* *)* `consequent:` *(not [ALLDIFFERENT* x_1 *...* x_{n+1}*])*	*(owl:Restriction rrr)* *(rdf:Property ppp)*

Table 17.5 - RDFS/OWL to CL Metamodel Translation

subgraph RDFStatement RDFStatement	 RDFsubject *(rrr)* RDFpredicate *(owl:onProperty)* RDFobject *(ppp)* RDFsubject *(rrr)* RDFpredicate (*owl:cardinality)* RDFobject *(n)*	Universal Quantification Implication	`Binding: (Term:` *`(x owl:Thing))`* `antecedent:` *`(rrr x)`* `consequent:` *`(exists`* $((x_1$ *`owl:Thing) ...`* $(x_n$ *`owl:Thing)) (and`* *`[ALLDIFFERENT`* $x_1 \ldots x_n]$ *`(ppp x`* $x_1)$ *`...(ppp x`* $x_n)$ *`(forall ((z owl:Thing))(implies`* *`(ppp x z)`* *`(or (= z`* $x_1)$ *`... (= z`* $x_n))$ *`))`* *`))`*	*`(owl:Restriction rrr)`* *`(rdf:Property ppp)`*
subgraph RDFStatement RDFStatement	 RDFsubject *(rrr)* RDFpredicate *(owl:onProperty)* RDFobject *(ppp)* RDFsubject *(rrr)* RDFpredicate (*owl:allValuesFrom)* RDFobject *(ccc)*	Universal Quantification Equivalence	`Binding: (Term:` *`(x owl:Thing))`* `lvalue:` *`(rrr x)`* `rvalue:` *`(forall (y)(implies (ppp x y)(ccc y)))`*	*`(owl:Restriction rrr)`* *`(rdf:Property ppp)`*
subgraph RDFStatement RDFStatement	 RDFsubject *(rrr)* RDFpredicate *(owl:onProperty)* RDFobject *(ppp)* RDFsubject *(rrr)* RDFpredicate *(owl:someValuesFrom)* RDFobject *(ccc)*	Universal Quantification Equivalence	`Binding: (Term: (x` *`owl:Thing))`* `lvalue:` *`(rrr x)`* `rvalue:` *`(exists (y)(and (ppp x y)(ccc y)))`*	*`(owl:Restriction rrr)`* *`(rdf:Property ppp)`*
subgraph RDFStatement RDFStatement	 RDFsubject *(rrr)* RDFpredicate *(owl:onProperty)* RDFobject *(ppp)* RDFsubject *(rrr)* RDFpredicate *(owl:hasValue)* RDFobject *(vvv)*	Universal Quantification Equivalence	`Binding: (Term:` *`(x owl:Thing))`* `lvalue:` *`(rrr x)`* `rvalue:` *`(ppp x vvv)`*	*`(owl:Restriction rrr)`* *`(rdf:Property ppp)`*

Table 17.5 - RDFS/OWL to CL Metamodel Translation

RDFStatement	RDFsubject *(ppp)*
	RDFpredicate
	(rdf:type)
	RDFobject
	(owl:Functional Property)
	
	-or-
		Conjunction
		
		Universal Quantification
		
		Implication
			`UniversalQuantification:`
			`Binding: (`
			`Term: `*`(x owl:Thing)`*
			`Term: `*`(y owl:Thing)`*
			`Term: `*`(z owl:Thing)`*
			`)`
			
			`Implication: (`
			`antecedent: `*`(and (ppp x y)(ppp x z))`*
			`consequent: `*`(= y z)`*
			`)`
			
			`Binding: (`
			`Term: `*`(x owl:Thing)`*
			`Term: `*`(y rdfs:Literal)`*
			`Term: `*`(z rdfs:Literal)`*
			`)`
			
			`antecedent: `*`(and (ppp x y)(ppp x z))`*
			`consequent: `*`(= y z)`*
				`(owl_Property ppp)`

RDFStatement	RDFsubject *(ppp)*
	RDFpredicate
	(rdf:type)
	RDFobject*(owl: InverseFunctional Property)*
		Universal Quantification
		
		Implication
			`Binding: (`
			`Term: `*`(x owl:Thing)`*
			`Term: `*`(y owl:Thing)`*
			`Term: `*`(z owl:Thing)`*
			`)`
			
			`antecedent: `*`(and (ppp y x)(ppp z x))`*
			`consequent: `*`(= y z)`*
				`(owl:Object Property ppp)`

RDFStatement	RDFsubject *(ppp)*
	RDFpredicate
	(rdf:type)
	RDFobject
	(owl:Symmetric Property)
		Universal Quantification
		
		Implication
			`Binding: (`
			`Term: `*`(x owl:Thing)`*
			`Term:`*`(y owl:Thing)`*
			`)`
			
			`antecedent: `*`(ppp x y)`*
			`consequent: `*`(ppp y x)`*
				`(owl:Object Property ppp)`

RDFStatement	RDFsubject *(ppp)*
	RDFpredicate
	(rdf:type)
	RDFobject
	(owl:Transitive Property)
		Universal Quantification
		
		Implication
			`Binding: (`
			`Term: `*`(x owl:Thing)`*
			`Term: `*`(y owl:Thing)`*
			`Term: `*`(z owl:Thing)`*
			`)`
			
			`antecedent: `*`(and (ppp x y)(ppp y z))`*
			`consequent: `*`(ppp x z)`*
				`(owl:Object Property ppp)`

Table 17.5 - RDFS/OWL to CL Metamodel Translation

RDFStatement	RDFsubject *(ppp)*
	RDFpredicate*(owl:*
	equivalentProperty)
	RDFobject *(qqq)*
		Universal Quantification
		
		Implication
			`Binding: (`
			`Term: (x owl:Thing)`
			`Term: (y owl:Thing)`
			`)`
			
			`antecedent: (ppp x y)`
			`consequent: (qqq x y)`
				(owl_Property ppp)
				(owl_Property qqq)

RDFStatement	RDFsubject *(ppp)*
	RDFpredicate
	(owl:inverseOf)
	RDFobject *(qqq)*
		Universal Quantification
		
		Implication
			`Binding: (`
			`Term: (x owl:Thing)`
			`Term: (y owl:Thing)`
			`)`
			
			`antecedent: (ppp x y)`
			`consequent: (qqq y x)`
				(owl_Property ppp)
				(owl_Property qqq)

RDFStatement	RDFsubject *(ccc)*
	RDFpredicate
	(owl:equivalentClass)
	RDFobject *(ddd)*
		Universal Quantification
		
		Implication
			`Binding: (Term: (x owl:Thing))`
			
			`antecedent: (ccc x)`
			`consequent: (ddd x)`
				(owl:Class ccc)
				(owl:Class ddd)

RDFStatement	RDFsubject *(ccc)*
	RDFpredicate
	(owl:disjointWith)
	RDFobject *(ddd)*
		Universal Quantification
		
		Negation
			`Binding: (Term: (x owl:Thing))`
			
			`Sentence: (`
			`Conjunction: (`
			`Sentence: (ccc x)`
			`Sentence: (ddd x)`
			`))`
				(owl:Class ccc)
				(owl:Class ddd)

RDFStatement	RDFsubject *(ccc)*
	RDFpredicate
	(owl:complementOf)
	RDFobject *(ddd)*
		Universal Quantification
		
		Implication
			`Binding: (Term: (x owl:Thing))`
			
			`antecedent: (ccc x)`
			`consequent: (not(ddd x))`
				(owl:Class ccc)
				(owl:Class ddd)

Table 17.5 - RDFS/OWL to CL Metamodel Translation

RDFStatement RDFStatement … RDFStatement	RDFsubject *(ccc)* RDFpredicate *(owl:intersectionOf)* RDFobject *(lll-1)* RDFsubject *(lll-1)* RDFpredicate *(rdf:first)* RDFobject *(aaa-1)* RDFsubject *(lll-1)* RDFpredicate *(rdf:rest)* RDFobject *(lll-2)* … RDFsubject *(lll-n)* RDFpredicate *(rdf:first)* RDFobject *(aaa-n)* RDFsubject *(lll-n)* RDFpredicate *(rdf:rest)* RDFobject *(rdf:nil)*	Universal Quantification Implication	`Binding: (Term:` *`(x owl:Thing))`* `antecedent:` *`(ccc x)`* `consequent:` *`(and (aaa-1 x) ... (aaa-n x) )`*	*`(owl:Class ccc)`*
RDFStatement RDFStatement … RDFStatement	RDFsubject *(ccc)* RDFpredicate *(owl:unionOf)* RDFobject *(lll-1)* RDFsubject *(lll-1)* RDFpredicate *(rdf:first)* RDFobject *(aaa-1)* RDFsubject *(lll-1)* RDFpredicate *(rdf:rest)* RDFobject *(lll-2)* … RDFsubject *(lll-n)* RDFpredicate *(rdf:first)* RDFobject *(aaa-n)* RDFsubject *(lll-n)* RDFpredicate *(rdf:rest)* RDFobject *(rdf:nil)*	Universal Quantification Implication	`Binding: (Term:` *`(x owl:Thing))`* `antecedent:` *`(ccc x)`* `consequent:` *`(or (aaa-1 x) ... (aaa-n x) )`*	*`(owl:Class ccc)`*

Table 17.5 - RDFS/OWL to CL Metamodel Translation

RDFStatement	RDFsubject *(ccc)* RDFpredicate *(owl:oneOf)* RDFobject *(lll-1)*	Universal Quantification Implication	`Binding: (Term:` *`(x owl:Thing))`* `antecedent:` *`(ccc x)`* `consequent:` *`(or (= aaa-1 x) ... (= aaa-n x) )`*	*`(owl:Class ccc)`*
RDFStatement	RDFsubject *(ccc)* RDFpredicate *(rdf:type)* RDFobject *(owl:Class)*			
RDFStatement	RDFsubject *(lll-1)* RDFpredicate *(rdf:first)* RDFobject *(aaa-1)* RDFsubject *(lll-1)* RDFpredicate *(rdf:rest)* RDFobject *(lll-2)*			
…	…			
RDFStatement	RDFsubject *(lll-n)* RDFpredicate *(rdf:first)* RDFobject *(aaa-n)* RDFsubject *(lll-n)* RDFpredicate *(rdf:rest)* RDFobject *(rdf:nil)*			

Table 17.5 - RDFS/OWL to CL Metamodel Translation

RDFStatement	RDFsubject *(ccc)* RDFpredicate *(owl:oneOf)* RDFobject *(lll-1)*	Universal Quantification Implication	`Binding: (Term:` *`(x rdfs:Literal))`* `antecedent:` *`(ccc x)`* `consequent:` *`(or (= aaa-1 x) ... (= aaa-n x) )`*	*`(owl:DataRange ccc)`*
RDFStatement	RDFsubject *(ccc)* RDFpredicate *(rdf:type)* RDFobject *(owl:DataRange)*			
RDFStatement	RDFsubject *(lll-1)* RDFpredicate *(rdf:first)* RDFobject *(aaa-1)* RDFsubject *(lll-1)* RDFpredicate *(rdf:rest)* RDFobject *(lll-2)*			
…	…			
RDFStatement	RDFsubject *(lll-n)* RDFpredicate *(rdf:first)* RDFobject *(aaa-n)* RDFsubject *(lll-n)* RDFpredicate *(rdf:rest)* RDFobject *(rdf:nil)*			

Table 17.5 - RDFS/OWL to CL Metamodel Translation

RDFStatement	RDFsubject *(ccc)* RDFpredicate *(rdf:type)* RDFobject *(owl:AllDifferent)*	Universal Quantification Implication	`Binding: (Term: `*`(x owl:Thing))`* `antecedent: `*`(ccc x)`* `consequent: `*`(or (= aaa-1 x) ... (= aaa-n x) )`*	*`(owl:Class ccc)`*
RDFStatement	RDFsubject *(ccc)* RDFpredicate*(owl:distinctMembers)* RDFobject *(lll-1)*	Sentence	`[ALLDIFFERENT aaa-1 ... aaa-n]`	
RDFStatement	RDFsubject *(lll-1)* RDFpredicate *(rdf:first)* RDFobject *(aaa-1)* RDFsubject *(lll-1)* RDFpredicate *(rdf:rest)* RDFobject *(lll-2)*			
…	…			
RDFStatement	RDFsubject *(lll-n)* RDFpredicate *(rdf:first)* RDFobject *(aaa-n)* RDFsubject *(lll-n)* RDFpredicate *(rdf:rest)* RDFobject *(rdf:nil)*			
RDFStatement	RDFsubject *(aaa)* RDFpredicate *(rdf:type)* RDFobject *(bbb)*	Relation	`predicate: `*`bbb`* `arguments [1]: `*`aaa`*	*`(owl:Class bbb)`*

RDFStatement	RDFsubject *(aaa)*
	RDFpredicate *(rdfs:domain)*
	RDFobject *(bbb)*
		Universal Quantification
		
		Implication
			`Binding: (`
			`Term: `*`(u rdfs:Resource)`*
			`Term: `*`(y rdfs:Resource)`*
			`)`
			
			`antecedent: `*`(aaa u y)`*
			`consequent: `*`(bbb u)`*
				`(owl:Class bbb)`
				`(rdf:Property aaa)`

RDFStatement	RDFsubject *(aaa)*
	RDFpredicate *(rdfs:range)*
	RDFobject *(bbb)*
		Universal Quantification
		
		Implication
			`Binding: (`
			`Term: `*`(u rdfs:Resource)`*
			`Term: `*`(y rdfs:Resource)`*
			`)`
			
			`antecedent: `*`(aaa u y)`*
			`consequent: `*`(bbb y)`*
				`(owl:Class bbb)`
				`(rdf:Property aaa)`

Table 17.5 - RDFS/OWL to CL Metamodel Translation

RDFStatement	RDFsubject *(aaa)* RDFpredicate *(rdfs:subClassOf)* RDFobject *(bbb)*	Universal Quantification Implication	`Binding: Term:` *`(u rdfs:Resource)`* `antecedent:` *`(aaa u)`* `consequent:` *`(bbb u)`*	*`(owl:Class bbb)`* *`(owl:Class aaa)`*

RDFStatement	RDFsubject *(aaa)*
	RDFpredicate*(rdfs:subPropertyOf)*
	RDFobject *(bbb)*
		Universal Quantification
		
		Implication
			`Binding: (`
			`Term:` *`(u rdfs:Resource)`*
			`Term:` *`(y rdfs:Resource)`*
			`)`
			
			`antecedent:` *`(aaa u y)`*
			`consequent:` *`(bbb u y)`*
				`(rdf:Property bbb)`
				`(rdf:Property aaa)`

any other triple RDFStatement	RDFsubject *(aaa)* RDFpredicate *(ppp)* RDFobject *(bbb)*	Relation	`predicate:` *`ppp`* `arguments [1]:` *`aaa`* `arguments [2]:` *`bbb`*	*`(rdf:Property ppp)`*

In addition, depending on the dialect of OWL (OWL DL or OWL Full) in question, certain hierarchical axioms are assumed, which enforce the distinction between owl:ObjectProperty and owl:DatatypeProperty, for example. For OWL DL, they also enforce the strict segregation between classes, properties, and individuals. These are summarized below for comparison purposes.

OWL Hierarchy Axioms

```
(forall ((x owl:Thing)(y owl:Thing))(iff (owl:sameAs x y)(= x y) ))
(forall (x) (implies (rdfs:Class x) (rdfs:Resource x))
(forall (x) (implies (rdf:Property x) (rdfs:Resource x))
(forall (x) (implies (rdfs:Datatype x) (rdfs:Class x))
(forall (x) (implies (owl:Thing x) (rdfs:Resource x))
(forall (x) (implies (owl_Property x) (rdf:Property x))
(forall (x) (implies (owl:Class x) (rdfs:Class x))
(forall (x) (implies (owl:DataRange x) (rdfs:Class x))
(forall (x) (implies (owl:Restriction x) (owl:Class x))
(forall (x) (implies (owl:ObjectProperty x) (owl_Property x))
(forall (x) (implies (owl:DatatypeProperty x) (owl_Property x))
(forall (x) (implies (owl:Thing x) (rdfs:Resource x))
(forall (x) (not (and (owl:Thing x)(rdfs:Literal x))))
(forall (x) (not (and (owl:Thing x)(owl:Ontology x))))
(forall (x) (not (and (owl:ObjectProperty x)(owl:DatatypeProperty x))))
```

OWL-DL Specific Hierarchy Axioms

```
(forall (x) (not (and (owl:Thing x)(owl_Property x))))
(forall (x) (not (and (owl:Thing x)(owl:Class x))))
(forall (x) (not (and (owl:Class x)(owl_Property x))))
(forall (x) (not (and (owl:OntologyProperty x)(owl_Property x))))
(forall (x) (not (and (owl:AnnotationProperty x)(owl_Property x))))
```

17.4 RDFS to CL Mapping in MOF QVT

```
transformation RDFS2CL(in src:RDFS,out dest:CL);

configuration property EMBED_TRIPLES := false; --  default value

-----------------------------------------------------------------
------------- Conversion of RDF Triples -------------------------
-----------------------------------------------------------------
query RDFRessource::isSchemaType() : Boolean =
  if self.RDFpredicate.isTypeOf(TypedLiteral)
    and self.lexicalForm='xsd:type' then true
    else false
  endif;


mapping RDFStatement::convertTriple() : AtomicSentence {
  init {
    result := if EMBED_TRIPLES=true then
                  self.convertTripleEmbedded()
              else
                 if not self.RDFpredicate.isSchemaType()
                   then self.convertTripleDirect()
                   else self.convertTripleDirectWithSchemaType()
                 endif
              endif;
  }
}

mapping RDFStatement::convertTripleDirect() : AtomicSentence  {
  predicate := self.RDFpredicate.map convertRessource();
  arguments := {
    self.RDFsubject.map convertRessource();
    self.RDFobject.map convertRessource();
    };
}

mapping RDFStatement::convertTripleDirectWithSchemaType() : AtomicSentence  {
  predicate := self.RDFobject.map convertRessource();
  arguments := self.RDFsubject.map convertRessource();
  end { -- TODO: Add here the implied axioms
  }
}

mapping RDFStatement::convertTripleEmbedded() : AtomicSentence  {
  predicate := new LogicalName('rdf_triple');
  arguments := {
    self.RDFpredicate.map convertRessource();
    self.RDFsubject.map convertRessource();
    self.RDFobject.map convertRessource();
    };
  end { -- axioms
    result.parent().phrase += {
      self.map addPropertyAxiom();
      self.map addPromiscuityAxiom();
    };
  }
```

```
}

-----------------------------------------------------------------
------------- Conversion of RDF Ressources ----------------------
-----------------------------------------------------------------

mapping RDFRessource::convertRessource() : Sentence
 disjuncts PlainLiteral::convertLiteral,
           TypedLiteral::convertLiteral,
           Graph::convertGraph
{}


mapping PlainLiteral::convertLiteral() : FunctionalTerm  {
  operator := new LogicalName("stringInLang");
  arguments := {
     new LogicalName(self.lexicalForm);
     if self.language then new LogicalName(self.language);
  };
}

mapping TypedLiteral::convertLiteral() : FunctionalTerm {
  operator := new LogicalName(self.datatypeURI);
    -- TODO: is datatypeURI a string?
  arguments := new LogicalName(self.lexicalForm);
}

mapping Graph::convertGraph() : Sentence {
  init {
    result := new ExistentialQuantification(
      self.statement[#BlankNode]->nodeID->asOrderedSet(),
      new Conjunction(self.statement->map convertTriple())
    );
  }
}

-------------------------------------------------------------------------
---------------- Axioms when embedding triples --------------------------
-------------------------------------------------------------------------



mapping RDFStatement::addPropertyAxiom() : Sentence {
  -- (forall (x y z)(iff (rdf_triple y x z)(and (rdf:Property x)(x y z))))
  init {
    result := new UniversalQuantification(
     Sequence{'x','y','z'},
     new Biconditional(
       new AtomicSentence('rdf_triple',Sequence{'y','x','z'}),
       new Conjunction(
            Sequence {
              new AtomicSentence('rdf:Property',Sequence{'x'}),
              Sequence{'x','y','z'}
            }
          )
```

```
      )
    );
 }
}

mapping RDFStatement::addPromiscuityAxiom() : Sentence {
  -- (forall (x)(rdf:Property x))
  init {
    result := new UniversalQuantification(
     Sequence{'x'},
     new AtomicSentence(new LogicalName('rdf:Property'),Sequence{'x'})
     );
 }
}

------------------------------------------------------------------
--- Constructor operations for main CL concepts -------------------
------------------------------------------------------------------

constructor LogicalName(lname:String) {
  name := lname;
}

constructor UniversalQuantification(names:Sequence(String),s:Sentence) {
   boundName := names->object(n) LogicalName {name:=n;};
   body := s;
}

constructor Biconditional(left:Sentence,right:Sentence) {
  lvalue := left;
  rvalue := right;
}

constructor AtomicSentence(pName:String,args:Sequence(Any)) {
  predicate := new LogicalName(pName);
  arguments := args->collect(a |
                if a.isKindOf(String) then new LogicalName(name=a)
                else a endif);
}

constructor Conjunction(lsentence : Sequence(Sentence)) {
  conjunct := lsentence;
}
```

18 References (non-normative)

[BCMNP] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider editors;*The Description Logic Handbook: Theory, Implementation and Applications;* Cambridge University Press, January 2003

[CGS] Sowa, J.F., *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, Reading, MA, 1984

[DOLCE] A. Gangemi, N. Guarino, C. Masolo, A. Oltramari and L. Schneider, Sweetening Ontologies with DOLCE, 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02), 1-4 October 2002, Siguenza, Spain

[Fikes & McGuinness] *An Axiomatic Semantics for RDF, RDF Schema and DAML+OIL* <http://www.ksl.stanford.edu/people/dlm/daml-semantics/abstract-axiomatic-semantics-august2001.html> Fikes, R., McGuinness, D. L., KSL Technical Report KSL-01-01, August 2001

[GE] J. Paul, S. Withanachchi, R. Mockler, M. Gartenfeld, W. Bistline and D. Dologite, Enabling B2B Marketplaces: the case of GE Global Exchange Services, in *Annals Of Cases On Information Technology*, Hershey, PA : Idea Group, 2003

[GuarWel] N. Guarino and C. Welty, Identity, Unity and Individuality: Towards a Formal Toolkit for Ontological Analysis, in: W. Horn (ed) Proceedings of ECAI-2000: The European Conference on Artificial Intelligence IOS Press, Amsterdam, 2000

[KIF] M. R. Genesereth & R. E. Fikes, Knowledge Interchange Format, Version 3.0 Reference Manual. KSL Report KSL-92-86, Knowledge Systems Laboratory, Stanford University, June 1992

[MSDW] R. Colomb, A. Gerber and M. Lawley, Issues in Mapping Metamodels in the Ontology Definition Metamodel, 1st International Workshop on the Model-Driven Semantic Web (MSDW 2004) Monterey, California, USA. 20-24 September, 2004

[Nary] W3C Defining N-ary Relations on the Semantic Web: Use with Individuals. Working Draft 21 July 2004. http://www.w3.org/TR/2004/WD-swbp-n-aryRelations-20040721/

[ODM RFP] Ontology Definition Metamodel Request for Proposal, OMG Document ad/2003-03-40

[OntoClean] N. Guarino and C. Welty. Evaluating Ontological Decisions with OntoClean, Communications of the ACM, 45(2) (2002) 61-65

[OWL OV] OWL Web Ontology Language Overview, W3C Recommendation 10 February 2004. Deborah L. McGuinness and Frank van Harmelen eds. Latest version is available at: http://www.w3.org/TR/owl-features/

[OWL Reference] OWL Web Ontology Language Reference. W3C Recommendation 10 February 2004, Mike Dean, Guus Schreiber, eds. Latest version is available at http://www.w3.org/TR/owl-ref/

[OWL XML Syntax] OWL Web Ontology Language XML Presentation Syntax. Masahiro Hori, Jérôme Euzenat, and Peter F. Patel-Schneider, Editors. W3C Note, 11 June 2003. Latest version is available at http://www.w3.org/TR/owl-xmlsyntax/

[PartWhole] W3C Simple part-whole relations in OWL Ontologies. W3C Working Draft 15 January 2005. http://www.cs.man.ac.uk/~rector/swbp/simple-part-whole/simple-part-whole-relations-v0-2.html

[RDF/TM] W3C A survey of RDF/Topic Maps Interoperability Proposals W3C Working Draft 29 March, 2005. Latest version is available at http://www.w3.org/TR/2005/WD-rdftm-survey-20050329

[Rose] IBM Rational Rose, http://www-130.ibm.com/developerworks/rational/products/rose

[SCL Translation] *Translating Semantic Web Languages into SCL*, Patrick Hayes, IHMC, November 2004. Latest version available at http://www.ihmc.us/users/phayes/CL/SW2SCL.htm

[UML2.1 Infra] Unified Modeling Language: Infrastructure, version 2.1. OMG Specification, ptc/06-04-03. Latest version (convenience document) is available at http://www.omg.org/docs/ptc/06-04-03.pdf

[UML2.1] Unified Modeling Language: Superstructure, version 2.1. OMG Specification, ptc/06-04-02. Latest version (convenience document) is available at http://www.omg.org/docs/ptc/06-04-02.pdf

[WinChafHerr] M. Winston, R. Chaffin, and D. Herrmann, (1987). A taxonomy of part-whole relations. Cognitive Science 11, 417-444

[XSCHD] XML Schema Datatypes in RDF and OWL, W3C Working Group Note 14 March 2006, Jeremy J. Carroll, Jeff Z. Pan, eds. Latest version is available at http://www.w3.org/TR/swbp-xsch-datatypes/

Annex A
Foundation Library (M1) for RDF and OWL

(normative)

A.1 Introduction

This annex includes several libraries: (1) an M1 library to be used with the RDF metamodel, (2) an M1 library to be used with the OWL metamodel in addition to the RDF library, (3) a model library to be used with the RDF profile, and (4) a model library for use with the OWL profile in addition to the RDF profile model library.

A.2 RDF Metamodel Library Elements

An M1 instance of either the RDF or OWL metamodels will generally require the use of some built-in resources as M1 instances of some M2 classes. Table A.1 gives a foundation library containing resources that may be required for use with the RDF Metamodel.

Table A.1 - Foundation Library (M1) for Use with the RDF Metamodel

M1 Object	Metaclass/Classifier	Properties	Description, Constraints
RDF Model Library	UML::Package	n/a	Contains the set of M1 model elements defined in this table
nil	RDF::RDFS::RDFList	n/a	The value of RDFfirst is nil; the value of RDFrest is nil; Special instance of RDFList - the empty list
label	RDF::RDFS::RDFProperty	language: String [0..1]	label is an optional property of RDFS Resource; the value of the label property must be a UML::LiteralString; A human-readable label for a resource
comment	RDF::RDFS::RDFProperty	language: String [0..1]	comment is an optional property of RDFSResource; the value of the comment property must be a UML::LiteralString; A human-readable comment associated with a resource
seeAlso	RDF::RDFS::RDFProperty	n/a	seeAlso is an optional property of RDFSResource; its type must also be RDFSResource; a reference providing further information about the resource that classifies it
isDefinedBy	RDF::RDFS::RDFProperty	n/a	isDefinedBy is a subproperty of seeAlso; a reference providing definitional information about the resource that classifies it

Table A.1 - Foundation Library (M1) for Use with the RDF Metamodel

M1 Object	Metaclass/Classifier	Properties	Description, Constraints
_1, _2, _3, _4, _5, _6, _7, _8, _9, _10, etc.	RDF:RDFS::RDFSContainer MembershipProperty	n/a	*Ordered* properties (meaning that the properties themselves are ordered by their number, essentially indices) indicating that their object is a member of the container that is their subject; every instance of RDFSContainerMembershipProperty is a subPropertyOf member
value	RDF::RDFBase::RDF Property	n/a	Distinguished instance of RDFProperty with no specific interpretation. Intended to be used to, for example, indicate the principal value in a complex of properties others of which provide context (units, for example). Deprecated in the ODM.
XMLLiteral	RDF::RDFS::RDFSDatatype	The value of datatypeURI for XMLLiteral is http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral.	A distinguished RDF datatype, which is a subclass of TypedLiteral, used for embedding XML in RDF.

A subset of the XML Schema Datatypes, specified in, are also available for use in RDF. We provide these in a separate M1 package. *All others are considered unsuitable for use with RDF and should not be used.*

Table A.2 - Foundation Library (M1) Defining XML Schema Datatypes For Use with RDF

M1 Object	Metaclass/Classifier	Properties	Description, Constraints
XSD Model Library	UML::Package	n/a	The package containing the set of M1 model elements defined in this table
string	RDF::RDFS::RDFSDatatype	The value of datatypeURI for string is http://www.w3.org/TR/xmlschema-2/#string	string is a subclass of TypedLiteral
boolean	RDF::RDFS::RDFSDatatype	The value of datatypeURI for boolean is http://www.w3.org/TR/xmlschema-2/#boolean	boolean is a subclass of TypedLiteral
decimal	RDF::RDFS::RDFSDatatype	The value of datatypeURI for decimal is http://www.w3.org/TR/xmlschema-2/#decimal	decimal is a subclass of TypedLiteral
float	RDF::RDFS::RDFSDatatype	The value of datatypeURI for float is http://www.w3.org/TR/xmlschema-2/#float	float is a subclass of TypedLiteral
double	RDF::RDFS::RDFSDatatype	The value of datatypeURI for double is http://www.w3.org/TR/xmlschema-2/#double	double is a subclass of TypedLiteral

Table A.2 - Foundation Library (M1) Defining XML Schema Datatypes For Use with RDF

dateTime	RDF::RDFS::RDFSDatatype	The value of datatypeURI for dateTime is http://www.w3.org/TR/xmlschema-2/#dateTime	dateTime is a subclass of TypedLiteral
time	RDF::RDFS::RDFSDatatype	The value of datatypeURI for time is http://www.w3.org/TR/xmlschema-2/#time	time is a subclass of TypedLiteral
date	RDF::RDFS::RDFSDatatype	The value of datatypeURI for date is http://www.w3.org/TR/xmlschema-2/#date	date is a subclass of RDFSLiteral
gYearMonth	RDF::RDFS::RDFSDatatype	The value of datatypeURI for gYearMonth is http://www.w3.org/TR/xmlschema-2/#gYearMonth	gYearMonth is a subclass of RDFS Literal
gYear	RDF::RDFS::RDFSDatatype	The value of datatypeURI for gYear is http://www.w3.org/TR/xmlschema-2/#gYear	gYear is a subclass of RDFSLiteral
gMonthDay	RDF::RDFS::RDFSDatatype	The value of datatypeURI for gMonthDay is http://www.w3.org/TR/xmlschema-2/#gMonthDay	gMonthDay is a subclass of RDFS Literal
gDay	RDF::RDFS::RDFSDatatype	The value of datatypeURI for gDay is http://www.w3.org/TR/xmlschema-2/#gDay	gDay is a subclass of RDFSLiteral
gMonth	RDF::RDFS::RDFSDatatype	The value of datatypeURI for gMonth is http://www.w3.org/TR/xmlschema-2/#gMonth	gMonth is a subclass of RDFSLiteral
hexBinary	RDF::RDFS::RDFSDatatype	The value of datatypeURI for hexBinary is http://www.w3.org/TR/xmlschema-2/#hexBinary	hexBinary is a subclass of RDFS Literal
base64Binary	RDF::RDFS::RDFSDatatype	The value of datatypeURI for base64Binary is http://www.w3.org/TR/xmlschema-2/#base64Binary	base64Binary is a subclass of RDFSLiteral
anyURI	RDF::RDFS::RDFSDatatype	The value of datatypeURI for anyURI is http://www.w3.org/TR/xmlschema-2/#anyURI	anyURI is a subclass of RDFSLiteral
normalizedString	RDF::RDFS::RDFSDatatype	The value of datatypeURI for normalizedString is http://www.w3.org/TR/xmlschema-2/#normalizedString	normalizedString is a subclass of RDFSLiteral
token	RDF::RDFS::RDFSDatatype	The value of datatypeURI for token is http://www.w3.org/TR/xmlschema-2/#token	token is a subclass of RDFSLiteral

Table A.2 - Foundation Library (M1) Defining XML Schema Datatypes For Use with RDF

language	RDF::RDFS::RDFSDatatype	The value of datatypeURI for language is http://www.w3.org/TR/xmlschema-2/#language	language is a subclass of RDFSLiteral
NMTOKEN	RDF::RDFS::RDFSDatatype	The value of datatypeURI for NMTOKEN is http://www.w3.org/TR/xmlschema-2/#NMTOKEN	NMTOKEN is a subclass of RDFS Literal
Name	RDF::RDFS::RDFSDatatype	The value of datatypeURI for Name is http://www.w3.org/TR/xmlschema-2/#Name	Name is a subclass of RDFSLiteral
NCName	RDF::RDFS::RDFSDatatype	The value of datatypeURI for NCName is http://www.w3.org/TR/xmlschema-2/#NCName	NCName is a subclass of RDFSLiteral
integer	RDF::RDFS::RDFSDatatype	The value of datatypeURI for integer is http://www.w3.org/TR/xmlschema-2/#integer	integer is a subclass of RDFSLiteral
nonPositiveInteger	RDF::RDFS::RDFSDatatype	The value of datatypeURI for nonPositiveInteger is http://www.w3.org/TR/xmlschema-2/#nonPositiveInteger	nonPositiveInteger is a subclass of RDFSLiteral
negativeInteger	RDF::RDFS::RDFSDatatype	The value of datatypeURI for negativeInteger is http://www.w3.org/TR/xmlschema-2/#negativeInteger	negativeInteger is a subclass of RDFSLiteral
long	RDF::RDFS::RDFSDatatype	The value of datatypeURI for long is http://www.w3.org/TR/xmlschema-2/#long	long is a subclass of RDFSLiteral
int	RDF::RDFS::RDFSDatatype	The value of datatypeURI for int is http://www.w3.org/TR/xmlschema-2/#int	int is a subclass of RDFSLiteral
short	RDF::RDFS::RDFSDatatype	The value of datatypeURI for short is http://www.w3.org/TR/xmlschema-2/#short	short is a subclass of RDFSLiteral
byte	RDF::RDFS::RDFSDatatype	The value of datatypeURI for byte is http://www.w3.org/TR/xmlschema-2/#byte	byte is a subclass of RDFSLiteral
nonNegativeInteger	RDF::RDFS::RDFSDatatype	The value of datatypeURI for nonNegativeInteger is http://www.w3.org/TR/xmlschema-2/#nonNegativeInteger	nonNegativeInteger is a subclass of RDFSLiteral
unsignedLong	RDF::RDFS::RDFSDatatype	The value of datatypeURI for unsignedLong is http://www.w3.org/TR/xmlschema-2/#unsignedLong	unsignedLong is a subclass of RDFSLiteral

Table A.2 - Foundation Library (M1) Defining XML Schema Datatypes For Use with RDF

unsignedInt	RDF::RDFS::RDFSDatatype	The value of datatypeURI for unsignedInt is http://www.w3.org/TR/xmlschema-2/#unsignedInt	unsignedInt is a subclass of RDFSLiteral
unsignedShort	RDF::RDFS::RDFSDatatype	The value of datatypeURI for unsignedShort is http://www.w3.org/TR/xmlschema-2/#unsignedShort	unsignedShort is a subclass of RDFSLiteral
unsignedByte	RDF::RDFS::RDFSDatatype	The value of datatypeURI for unsignedByte is http://www.w3.org/TR/xmlschema-2/#unsignedByte	unsignedByte is a subclass of RDFSLiteral
positiveInteger	RDF::RDFS::RDFSDatatype	The value of datatypeURI for positiveInteger is http://www.w3.org/TR/xmlschema-2/#positiveInteger	positiveInteger is a subclass of RDFSLiteral

A.3 OWL Metamodel Library Elements

Table A.3 gives a foundation library containing resources that may be required for use with the OWL Metamodel.

Table A.3 - Foundation Library (M1) for Use with the OWL Metamodel

M1 Object	Metaclass	Properties	Description, Constraints
OWL Model Library	UML::Package		Contains the set of M1 model elements defined in this table
Nothing	OWL::OWLBase::OWLClass		[1] Nothing is an RDFSsubclassOf every instance of OWLClass; [2] Thing is the complement of Nothing.
Thing	OWL::OWLBase::OWLClass		[1] Every instance of OWLClass is an RDFSsubclassOf Thing; [2] Thing is the default domain and range of every instance of OWLObjectProperty; [3] Thing is the default domain of every instance of OWLDatatypeProperty.
versionInfo	OWL::OWLBase::OWL AnnotationProperty		
RDF Model Library::label	OWL::OWLBase::OWL AnnotationProperty		Redefines label from Table A.1
RDF Model Library::comment	OWL::OWLBase::OWL AnnotationProperty		Redefines comment from Table A.1
RDF Model Library::seeAlso	OWL::OWLBase::OWL AnnotationProperty		Redefines seeAlso from Table A.1
RDF Model Library::isDefinedBy	OWL::OWLBase::OWL AnnotationProperty		Redefines isDefinedBy from Table A.1

A.4 UML Profile for RDF Library Elements

Table A.4 gives a foundation library containing resources that may be used in addition to and with the RDF Profile. Rather than creating a separate package for these elements, they augment the RDF profile package described in 14.2.

In the table below:

- The first column, M1 Object, represents the element in the model library being described.
- The second column, Base Class & Stereotype, the base class is the UML metamodel element that the M1 Object is an instance of, and the stereotype, if any, is the stereotype applied to the M1 object.
- The third column, Parent, represents the classifier that generalizes the M1 object if the M1 object is itself a classifier.
- The Properties column provides UML properties of the M1 object if that object is a classifier.
- Finally, the Description, Constraints column describes the M1 object and identifies additional constraints on that object, if any

Table A.4 - Foundation Library (M1) for Use with the RDF Profile

M1 Object	Base Class & Stereotype	Parent	Properties	Description, Constraints
_1, _2, _3, _4, _5, _6, _7, _8, _9, _10, etc.	UML::Property, UML:Association; «rdfsContainer Membership Property»			*Ordered* properties (meaning that the properties themselves are ordered by their number, essentially indices) indicating that their object is a member of the container that is their subject
nil	UML::Instance Specification, no stereotype			[1] The classifier for the Instance Specification must be RDFList; [2] The value of the first property must be nil; [3] the value of the rest property must be nil; Special instance of RDFList - the empty list.
RDFAlt	UML::Datatype; no stereotype	RDFSContainer		This is the class of RDF "Alternative" containers. «rdfAlt» is used conventionally to indicate to a human reader that typical processing will be to select one of the members of the container. The first member of the container, i.e., the value of the `rdf:_1` property, is the default choice.
RDFBag	UML::Datatype; no stereotype	RDFSContainer		This is the class of RDF "Bag" containers. It is used conventionally to indicate that the container is intended to be unordered and allow duplicate members.

Table A.4 - Foundation Library (M1) for Use with the RDF Profile

RDFList	UML::Datatype; no stereotype		uriRef: String [1] – the URI reference(s) for the list; first: [0..1] – the resource representing the first element in the list; rest: RDFList [0..1] – a sublist excluding the first element of the original list	This class represents descriptions of RDF collections, conventionally called lists and other list-like structures, corresponding to 10.6.3 (“RDFList”). [1] The value of the uriRef property must be a UML::LiteralString that is stereotyped by «uriReference»; [2] The value of the first property must be an instance of something stereotyped by «rdfsResource»; [3] The value of the rest property must be an instance of RDFList.
RDFSContainer	UML::Datatype; no stereotype		uriRef: String [1] – the URI reference(s) for the container	This is a super-class of RDF container classes, corresponding to 10.6.4 (“RDFSContainer”). [1] The value of the uriRef property must be a UML::LiteralString that is stereotyped by «uriReference».
RDFSeq	UML::Datatype; no stereotype	RDFSContainer		This is the class of RDF “Sequence” containers. It is used conventionally to indicate that the numerical ordering of the container membership properties of the container is intended to be significant.
value	UML::Property; no stereotype			Distinguished instance of RDFProperty with no specific interpretation. Intended to be used to, for example, indicate the principal value in a complex of properties others of which provide context (units, for example). Deprecated in the ODM.

Table A.5 gives a foundation library defining the set of XML Schema Datatypes that may be used with the RDF and OWL Profiles. *All others are considered unsuitable for use with RDF and should not be used*.

Table A.5 - Foundation Library (M1) Defining XML Schema Datatypes For Use with the RDF Profile

M1 Object	Base Class & Stereotype	Parent	Properties	Description, Constraints
XSD Library	UML::Package			The package containing the set of M1 model elements defined in this table.
string	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#string."
boolean	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for boolean is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#boolean."
decimal	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for decimal is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#decimal."
float	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for float is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#float."
double	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for double is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#double."
dateTime	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for dateTime is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#dateTime."
time	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for time is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#time."
date	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for date is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#date."
gYearMonth	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for gYearMonth is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#gYearMonth."
gYear	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for gYear is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#gYear."

Table A.5 - Foundation Library (M1) Defining XML Schema Datatypes For Use with the RDF Profile

gMonthDay	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for gMonthDay is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#gMonthDay."
gDay	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for gDay is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#gDay."
gMonth	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for gMonth is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#gMonth."
hexBinary	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for hexBinary is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#hexBinary."
base64Binary	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#base64Binary."
anyURI	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for anyURI is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#anyURI."
normalizedString	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for normalizedString is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#normalizedString."
token	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for token is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#token."
language	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for language is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#language."
NMTOKEN	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for NMTOKEN is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#NMTOKEN."
Name	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for Name is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#Name."

Table A.5 - Foundation Library (M1) Defining XML Schema Datatypes For Use with the RDF Profile

NCName	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for NCName is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#NCName."
integer	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for integer is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#integer."
nonPositiveInteger	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#nonPositiveInteger."
negativeInteger	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#negativeInteger."
long	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#long."
int	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#int."
short	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#short."
byte	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#byte."
nonNegativeInteger	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#nonNegativeInteger."
unsignedLong	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#unsignedLong."
unsignedInt	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for string is an instance of UML::Datatype, with a value of the value property: "http://www.w3.org/TR/xmlschema-2/#unsignedInt."

Table A.5 - Foundation Library (M1) Defining XML Schema Datatypes For Use with the RDF Profile

unsignedShort	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for unsignedShort is an instance of UML::Datatype, with a value of the value property: “http://www.w3.org/TR/xmlschema-2/#unsignedShort.”
unsignedByte	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for unsignedByte is an instance of UML::Datatype, with a value of the value property: “http://www.w3.org/TR/xmlschema-2/#unsignedByte.”
positiveInteger	UML::Datatype;«rdfsDatatype»			[1] The value of uriRef for positiveInteger is an instance of UML::Datatype, with a value of the value property: “http://www.w3.org/TR/xmlschema-2/#positiveInteger.”

A.5 UML Profile for OWL Library Elements

Table A.6 gives a foundation library containing resources that may be used in addition to and with the OWL Profile. Rather than creating a separate package for these elements, they augment the OWL profile package described in 14.3.

Table A.6 - Foundation Library (M1) for Use with the OWL Profile

M1 Object	Base Class & Stereotype	Parent	Properties	Description, Constraints
Nothing	UML::Class; «owlClass»			[1] Nothing is generalized by every M1 class stereotyped by «owlClass»; [2] There is a «complementOf» constraint between Nothing and Thing.
Thing	UML::Class; «owlClass»			[1] Thing generalizes every M1 class stereotyped by «owlClass».

Annex B
Conceptual Entity Relationship Modeling

B.1 Overview

UML is considered a basic conceptual modeling language from an ODM perspective. It is frequently used for this purpose, and is also critical for leveraging existing artifacts as a basis for ontology modeling, either through migration to one of the ontology-specific languages represented in the ODM, or integration with other components represented in these languages. Many resources that may be leveraged for ontology development are modeled not in UML, but in one of the dialects of the Entity-Relationship system. The ODM team considered including an ER metamodel, but in the end did not do so for several reasons: (1) there is no existing ISO or other standard reference for ER (as there is for each of the other conceptual modeling languages included herein), (2) there are many "dialects" for Entity Relationship Modeling implemented by various tools, without discriminating features that might be useful if implemented in an ODM conceptual ER metamodel, and (3) we believe that the best place for developing such a metamodel is the upcoming information modeling effort that will ultimately replace the current Common Warehouse Metamodel (CWM).

Even so, the team also believes that:

- A significant percentage of conceptual modeling is done in an ER framework.
- This will continue to be the case indefinitely.
- ER-style modeling is sufficiently similar to logical modeling in UML that providing an appendix describing a general mapping strategy from ER to UML and ODM would be useful.

This informative appendix shows the relationship between common ER constructs and comparable UML constructs. The only aspect of ER modeling without a close correspondence to UML is the concept of identifiers. Here, we describe an optional package sufficient to represent ER identifiers.

B.2 Basic Constructs: Entity, Attribute, Relationship

The basic constructs of ER correspond directly to UML constructs:

- Entity or entity type corresponds to class.
- Attribute corresponds to Attribute (a property which is an ownedAttribute of a class).
- Value set or Domain of an attribute corresponds to the type of the property.
- A composite attribute has a domain corresponding to a type which itself has attributes (in UML).
- Relationship or relationship type corresponds to association.
- Role in a relationship corresponds to a property at a memberEnd.
- A relationship which can have attributes, or itself participate in other relationships, corresponds to an association class.

B.3 Cardinality Constraints

A number of constructs in ER correspond to UML multiplicity constraints.

- Multiple valued attributes correspond to properties whose maximum multiplicity is greater than 1.
- Attributes which can be null correspond to properties whose minimum multiplicity is 0.
- Total participation of an entity playing a role corresponds to a minimum multiplicity greater than 0 in the corresponding property.
- Partial participation of an entity playing a role corresponds to a minimum multiplicity of 0 in the corresponding property.
- Cardinality ratio of 'one' corresponds to a maximum multiplicity of 1 in the corresponding property.
- Cardinality ratio of 'many' corresponds to a maximum multiplicity of ‘*’ in the corresponding property.
- Min-max notation for cardinality constraints corresponds approximately to multiplicity notation in UML, with slight variation. In ER, the constraints refer to the number of instances of the entity playing the role which can appear in the relationship set. In UML, the multiplicities refer to the number of instances of the class that is the type of the property which can be associated with a fixed collection of instances for the other memberEnds. So for a binary relationship, the min-max cardinality corresponds to the multiplicities for the opposite property. For relationships with degree greater than two, there is no simple correspondence between ER min-max cardinality constraints and UML multiplicities.

B.4 Generalization

The enhanced ER (EER) system has a generalization/specialization mechanism that corresponds to that in UML. The disjointness and completeness constraints for specialization correspond respectively to the isDisjoint and isCovering attributes of a GeneralizationSet in the UML PowerTypes package.

B.5 Identity

The major feature of ER modeling lacking in UML is the concept of identity. The MOF has a primitive identity construct, the isID attribute of Property in the Identities package, but UML does not support identity at all.

Identity is supported in ER in several ways:

- An entity instance can be identified by an attribute (key), or by a combination of attributes (compound key).
- An entity instance can be identified by the key of another entity it has a relationship with. For example, in a dialect of ER in which relationships cannot have attributes, that a student is given a grade in respect of enrollment in a course can be modeled by treating the enrollment as an entity identified by the keys of the student and course entities it is related to.
- In the special case of a weak entity, an instance of an entity is identified by a compound of local attributes (partial key) and the key of another (identifying or owner) entity it is in an identifying relationship with.

The ER identity feature can be supported by the following MOF package which could be included in UML. Class and Property are both from the UML2 Classes diagram, while NamedElement is from the Namespaces diagram of the UML2 Kernel package.

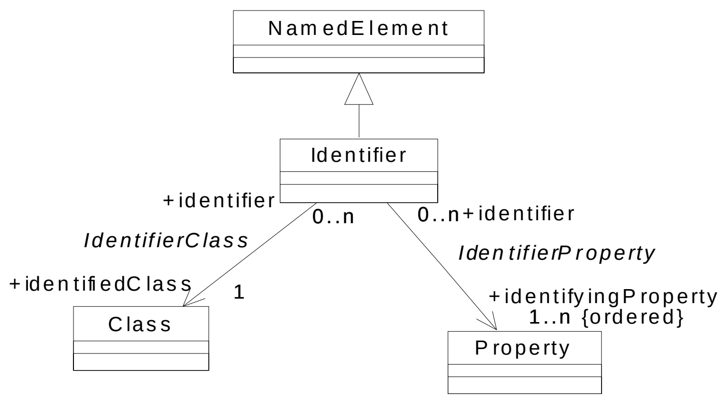

Figure B.1 - Identifiers (Keys) Diagram

B.5.1 Identifier

Description

Connects a class with one or more properties constituting a (possibly compound) key for the class.An instance of Identifier is identified by the property identifiedClass together with name from NamedElement. If there is only one instance of Identifier associated with an instance of Class, then name can be absent.

Attribute

- No additional attributes.

Associations

- identifiedClass [1] the class identified.
- identifyingProperty [1..n] the collection of one or more properties constituting the identifier.

Constraints

Identifier has a name if its identifiedClass has more than one identifier.

```
context Identifier inv:
(self.identifiedClass.identifier->sizeOf()>1)implies (self.name->
        exists(n : string | (self.identifiedClass.identifier.name->
           forAll(m: string | m = n implies
                    (self = self.identifiedClass.identifier))
           )
      )
)
```

All the identifying properties of the class must be associated with the class identified or a superclass, either as owned attributes or the type of a property at another memberEnd of an association.

```
context Identifier inv:
 (self.identifyingProperty-> forAll(p:Property |
     (self.identifiedClass.ownedAttribute->(exists (pp:Property | pp= p)) or
     (self.allparents().ownedAttribute->exists(pp:Property | pp= p)) or
     (self.identifiedClass.oclIsKindOf(p.opposite.type))
 ))
```

B.6 Profile for Identity

An optional package for UML is not very useful unless it has some way of drawing it. The easiest way to get a notation is to define a profile.

If a Class and Identifier are fixed, we have a set of Property. It therefore makes sense to profile Identifier as an extension of the metaclass Property, as shown in Figure B.2.

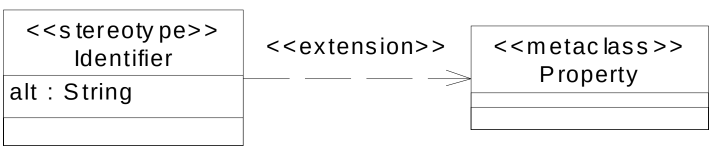

Figure B.2 - Profile for Identity

Tagged Values

- alt : string - name of instance of alternative identifier if there are more than one identifiers for a given class.

B.7 Example

We show an example of use of identifiers, employing the notation specified in Figure B.2, in Figure B.3. The application is keeping track of results in a sporting competition like the Olympics.

Team and Event are identified by single attributes: country for Team and eID for Event.

Competitor has a single composite identifier, the attribute cName and the property team. This identifier carries the tag ‘a’ for alt.

CompetesIn has no identifier. Instances are identified by OID, as is usual for UML classes. It could, however, be identified by the composite of the properties competitor and event.

The subclass MedalAwarded of CompetesIn does have an identifier, the attribute medal and the property event of its superclass CompetesIn. This identifier carries the tag ‘b’ for alt.

The subclass MedalWinner of Competitor has an additional identifier, carrying tag ‘c’ for alt. Besides the identifier inherited from its superclass Competitor, an instance of MedalWinner is identified by the property wins. So a medal winner is identified by the identifier of MedalAwarded, which is the medal and the event.

Strictly, the tag ‘b’ for alt is not needed in the identifier for MedalAwarded, as there is only one identifier for that class. The tag is used to improve readability.

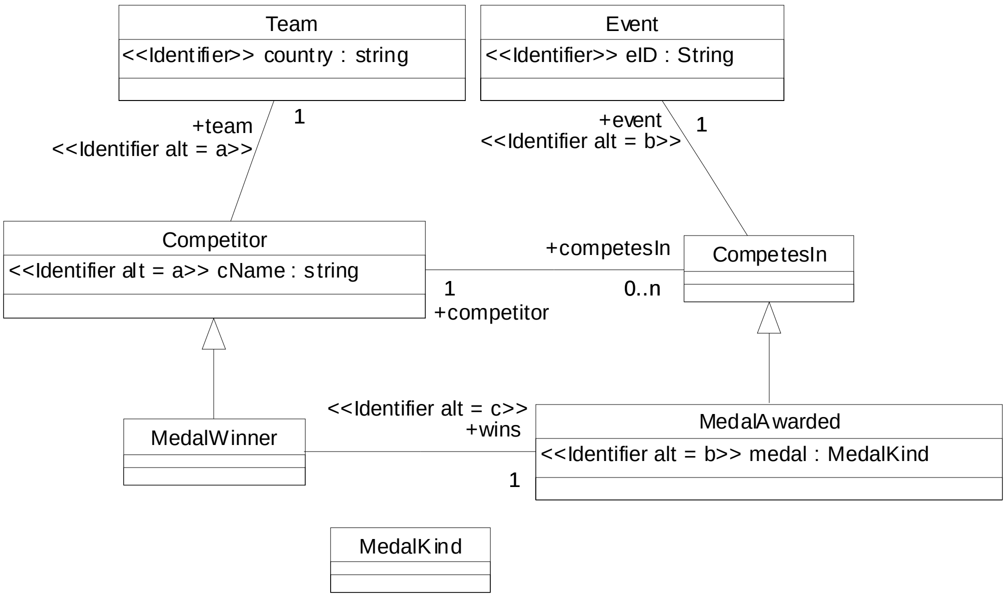

Figure B.3 - Example of Identifier Usage

Annex C
A Description Logic Metamodel

C.1 Introduction

This annex provides an introduction to Description Logics through the elaboration of an exemplar Description Logic Meta-Model.

The Description Logic (DL) meta-model defines a basic, minimally constrained DL. In use, DLs are typically found in the Knowledge-Base of a Knowledge Representation System, as shown in Figure C.1.

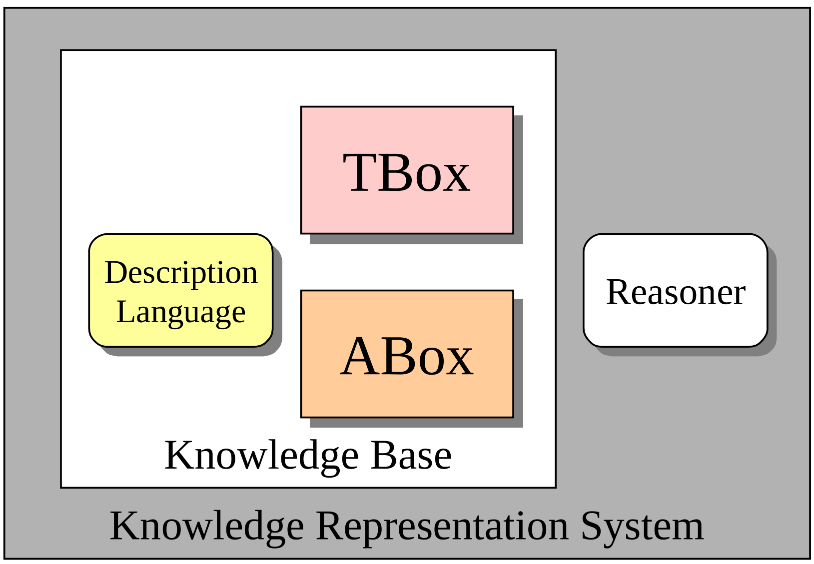

Figure C.1 - Knowledge Representation System

A DL Knowledge Base is traditionally divided into three principal parts:

1. Terminology or schema, the vocabulary of application domain, called the “TBox.”
2. Assertions, which are named individuals expressed in terms of the vocabulary, called the “ABox.”
3. Description Language that define terms and operators for build expressions.

Note that the TBox and ABox elements represent two separate meta-levels in the application domain.

C.2 Containers

Basic containment constructs of this DL meta-model, as shown in Figure C.2, are provided through the TBox and ABox elements, which correspond directly to the TBox and ABox concepts from description logics.

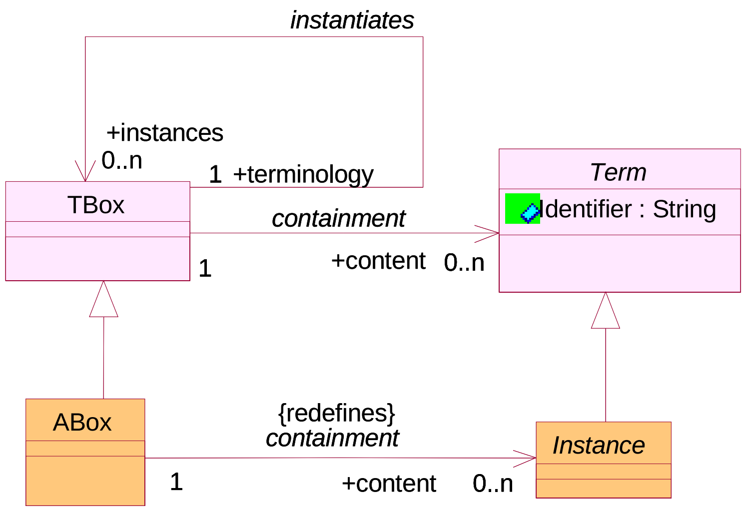

Figure C.2 - Basic Containment Constructs

C.2.1 TBox

Description

A TBox contains all of a DL model's terminology. The TBox may include Terms and any of the sub-classes of Term. Note that this includes Instances to allow supporting predefined, delineated instances as 'special terms' in the ABox. An example of this would be OWL Thing.

Associations

- Containment.content[0..n]: Term – the terminology contained in this TBox.
- instances[0..n]: ABox – the TBox that uses terms and instances from this TBox.
- terminology[1]: TBox – the TBox that contains the terminology used by this TBox (or ABox)

C.2.2 ABox

Description

An ABox contains all of a DL model's instances. The ABox extends TBox and restricts its content to be only the sub-classes Instance.

Associations

- Containment.content[0..n]: Instance – the instances contained in this ABox, redefining containment from TBox.

Semantics

All the instances in an ABox are expressed using the terminology from exactly one TBox.

C.3 Concepts and Roles

C.3.1 Element

Description

Element is an abstract base class of all atomic components in a DL as seen in Figure C.3. It defines the notion of unique identity so that references may be made to elements using that identifier.

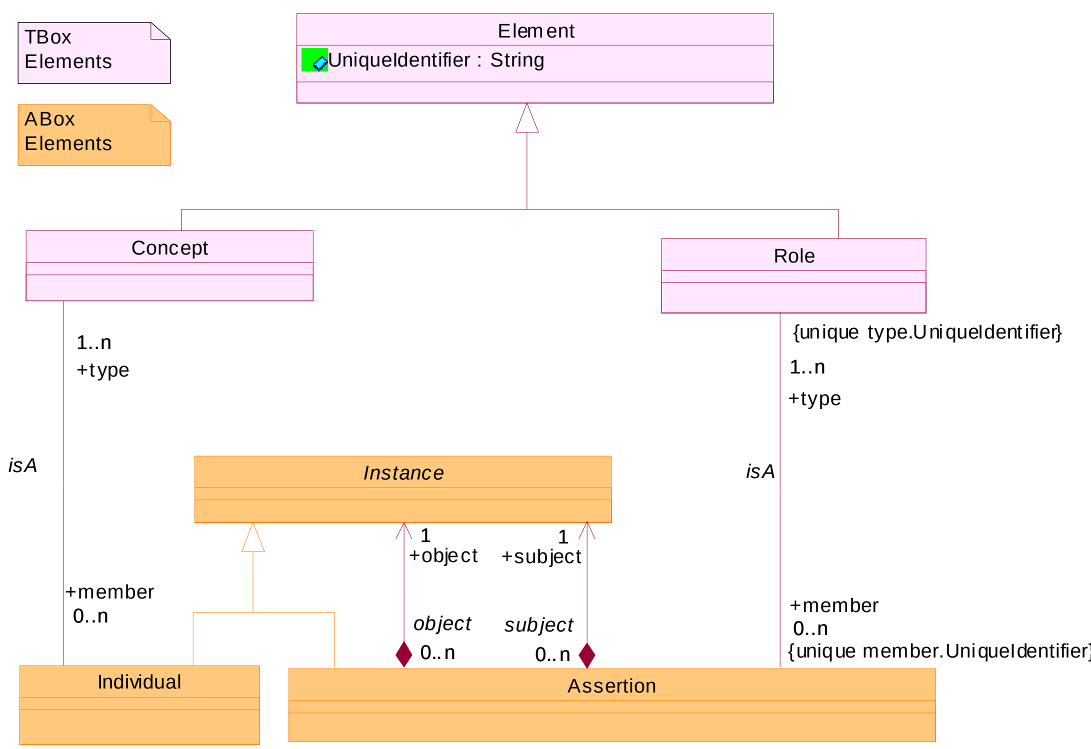

Figure C.3 - Element Model

Attributes

- UniqueIdentifier: String - Uniquely identifies a Element and all Elements are identified by a single value. That is if UniqueIdentifiers of two Elements are different, then the Elements are different. Note that this is different than URIs. UniqueIdentifier is required.

C.3.2 Concept

Description

Concept is a set of Instances which are define as having something in common.

Concept is a specialization of Element.

Similar Terms

Class, Entity, Topic, Type

Associations

- isA.member: Individual[0..n] -- The set of Individuals that are the extent of the concept.

C.3.3 Instance

Description

Instance provides an abstract base class for all ABox constructs. Instance is a specialization of Term.

Similar Terms

Object, Instantiation

C.3.4 Role

Description

A Role is a set of binary tuples, specifically (subject, object), that asset that this role for subject is satisfied by object. Role is a specialization of Element.

Similar Terms

Association, Attribute, Property, Slot

Associations

- isA.member: Assertion [0..n] -- The set of Assertions that are the extent of the concept.

C.3.5 Individual

Description

Individual is an instance of a Concept. An Individual is a specialization of Instance

Similar Terms

Object

Associations

- isA.type: Concept[1..n] – The set of concept sets that has this individual as a member.

C.3.6 Assertion

Description

Assertions are the specific binary tuples that are instances of Roles. An Assertion is a specialization of Instance.

Similar Terms

Link, Statement, Fact

Associations

- subject.subject: Instance[1] – The Instance that is the subject of the assertion.
- object.object: Instance[1] – The Instance that is the object of the assertion.
- sA.type: Concept[1..n] – The set of Roles that has this assertion as a member.
- predicate: Instance[1] – A derived reference to the Role which this assertion is an instance of. (Not shown in diagram.)

C.4 Datatypes

C.4.1 Datatype

Description

A Datatype is a specialization of Concept. Datatypes are those concepts whose members have no identity except their value, that is the members of a datatype are literals, as shown in Figure C.4. Datatype may represent primitive types, for example integer, string, or boolean; or user defined type, for example time-interval or length-in-meters.

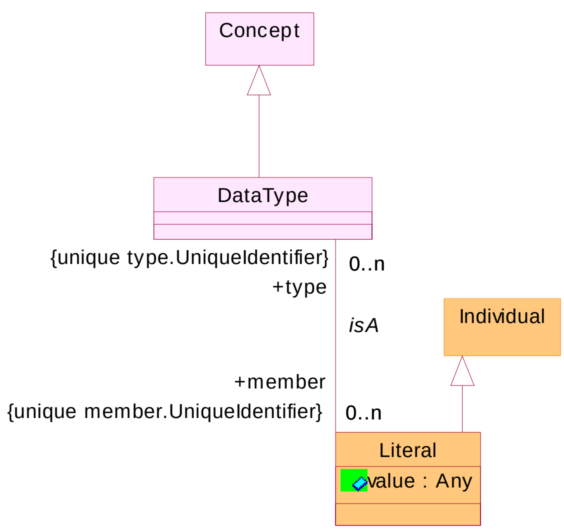

Figure C.4 - DataType Model

Associations

- isA.member: Literal[0..n] – the set of literals that are members of this datatype.

C.4.2 Literal

Description

Literals are the specification of instances of datatypes. The UniqueIdentifier inherited from Element is for a literal, uniquely defined by the literal's value itself.

Attributes

- value: Any – The implementation and Datatype dependent value of this literal.

Associations

- type: Datatype[0..n] – the possibly empty set of datatypes in which this literal is a member.

Constraints

Restricts range of Concept.type to a set of Datatypes.

Semantics

Element.UniqueIdentifier has a functional relation with Literal.value.

Literal.value has a functional relation with Element.UniqueIdentifier.

C.5 Collections

C.5.1 Collection

Description

A Collection is a specialization of Concept. Collection allows instances to be brought together as a group and referenced as a single collective. The class diagram for Collection is shown in Figure C.5.

Collection is conceptually a ‘bag’, that is un-order and allowing duplicate members..

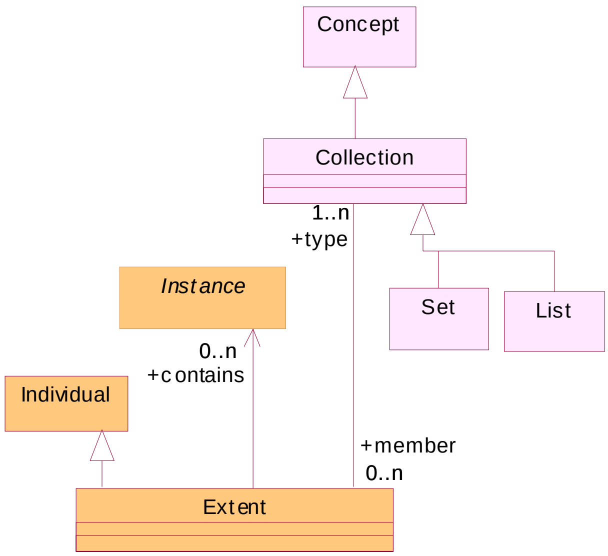

Figure C.5 - Collection Model

Similar Terms

Container; and Sequence, List, Bag, Set as specific types.

Associations

- isA.member: Extent[0..n] – The set of instances of a particular kind of collection.

C.5.2 List

Description

List is a specialization of Collection. List requires that the member instances that are in the collection are ordered in a user defined way.

Semantics

For all a_i , a_j members of the list, there is a comparator function C() such that $C(a_i) < C(a_j)$ if $i < j$

C.5.3 Set

Description

Set is a specialization of Collection. Set requires that the member instances in the collection are unique.

Semantics

For all a_i , a_j members of the list, there is a identity function I() such that $I(a_i)=I(a_j)$ iff $i = j$

C.5.4 Extent

Description

Extent is a specialization of Instance. Extent is the set of all instances of a collection of a particular type, for example the set of all Alphabetical-Lists.

Associations

- containment.contains: Instance[0..n] – Those instances that are in this instance of a collection.
- isA.type: Collection - The set of collection sets that has this extent as a member.

C.6 Expressions and Constructors

Expressions provide the mechanism for constructing class definitions and implications about TBox elements. They provide a hook for more expressive constraint and rule languages.

A number of common expression constructors, shown in Figure C.6, are provided as specializations of Constructor.

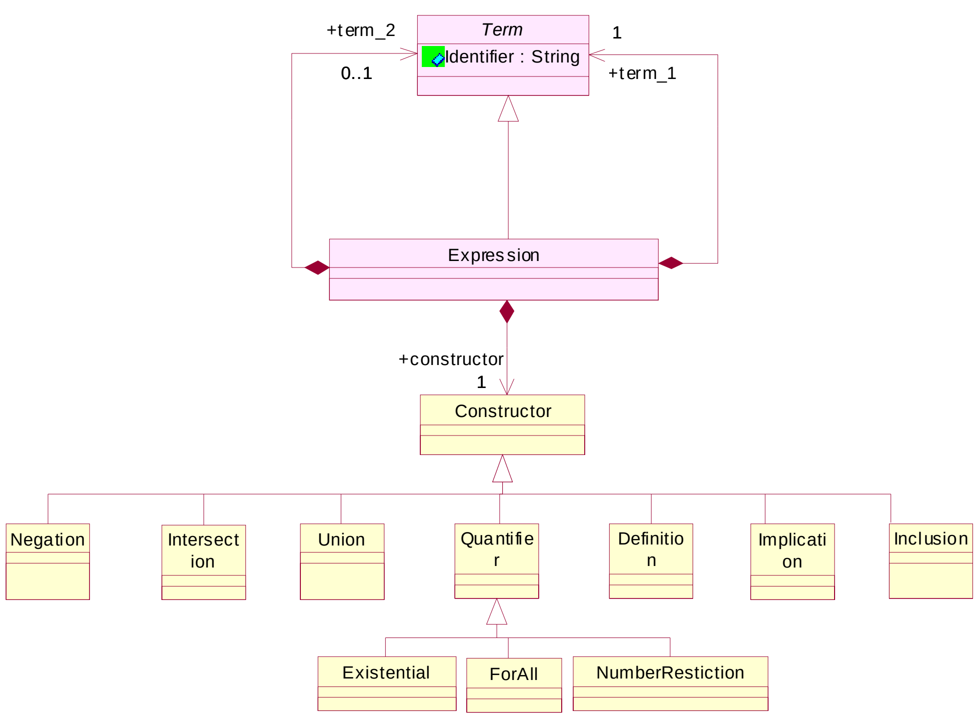

Figure C.6 - Specializations of Constructor

C.6.1 Term

Description

Terms are the components used to build expressions. They are an abstract root class of most DL classes, excluding only ABox, TBox, and Constructors.

Similar Terms

Word, Component

Attributes

- Identifier: String [0..1] – An optional identifier for this term.

C.6.2 Expression

Description

Expressions are the representation of the DL Knowledge Base Description Language, shown in Figure C.1. Expressions are an extension of Term and are also constructed from Terms using Constructors. Thus allowing arbitrarily complex expressions to be created.

Similar Terms

Statement, Formula

Associations

- term_1: Term[1] – The required term for the constructor.
- term_2: Term[0..1] – The optional term for the constructor.
- constructor: Constructor[1] – a monadic or dyadic operator applied to the terms.

C.6.3 Constructor

Description

A Constructor is an operator that is used to build expressions. A Constructor may be either monadic or dyadic.

Note that individual specializations of constructor may have additional semantics and restrictions that are not elaborated here.

Similar Terms

Operator

Semantics

- Monadic constructors have term_2.multiplicity = 0
- Dyadic constructors have term_2.multiplicity = 1

C.6.4 Intersection

Description

The Intersection constructor is a dyadic constructor. It results in the set of instances that are members of both the left-hand term and the right-hand term.

C.6.5 Negation

Description

The Negation constructor is a monadic constructor. It results in the set containing all instances not contained in the right-hand term.

C.6.6 Union

Description

The Union constructor is a dyadic constructor. It results is the set containing any instance that is a member of either the left-hand or right-hand term.

C.6.7 Quantifier

Description

A Quantifier is a specialization of Constructor. It is a monadic constructor. They are operators that bind the number of a role's assertions by specifying their quantity in a logical formula.

C.6.8 ForAll

Description

ForAll is a specialization of Quantifier. ForAll specifies that all members of term_1 must have the binding value for the specified role.

C.6.9 Existential

Description

Existential is a specialization of Quantifier. Existential specifies that at least one member of term_1 has the binding value for the specified role.

C.6.10 NumberRestriction

Description

NumberRestriction is a specialization of Quantifier. NumberRestriction specifies that a specified number of members have a value for the specified role, similar to cardinality or multiplicity.

Further specializations of NumberRestriction may include upper bound, lower bound and exact number specifications.

C.6.11 Definition

Description

Definition is a specialization of Constructor. It is dyadic. Definition is used in axioms to define the left-hand term as exactly the right-hand term.

C.6.12 Implication

Description

Implication is a specialization of Constructor. It is dyadic. Implication is a logical relationship between the term_1 and term_2, that states term_2 is true if term_1 is true.

C.6.13 Inclusion

Description

Inclusion is a specialization of Constructor. It is dyadic. Inclusion is a relation between the term_1 and the term_2 that states all members of the first are also members of the second. Inclusion is similar to sub-types, in that all members of a sub-type are included in the super-type.

C.7 Examples

The following two examples, in Figure C.7 and Figure C.8, illustrate the representation of simple statements as instance models of the DL meta-model.

C.7.1 Example One

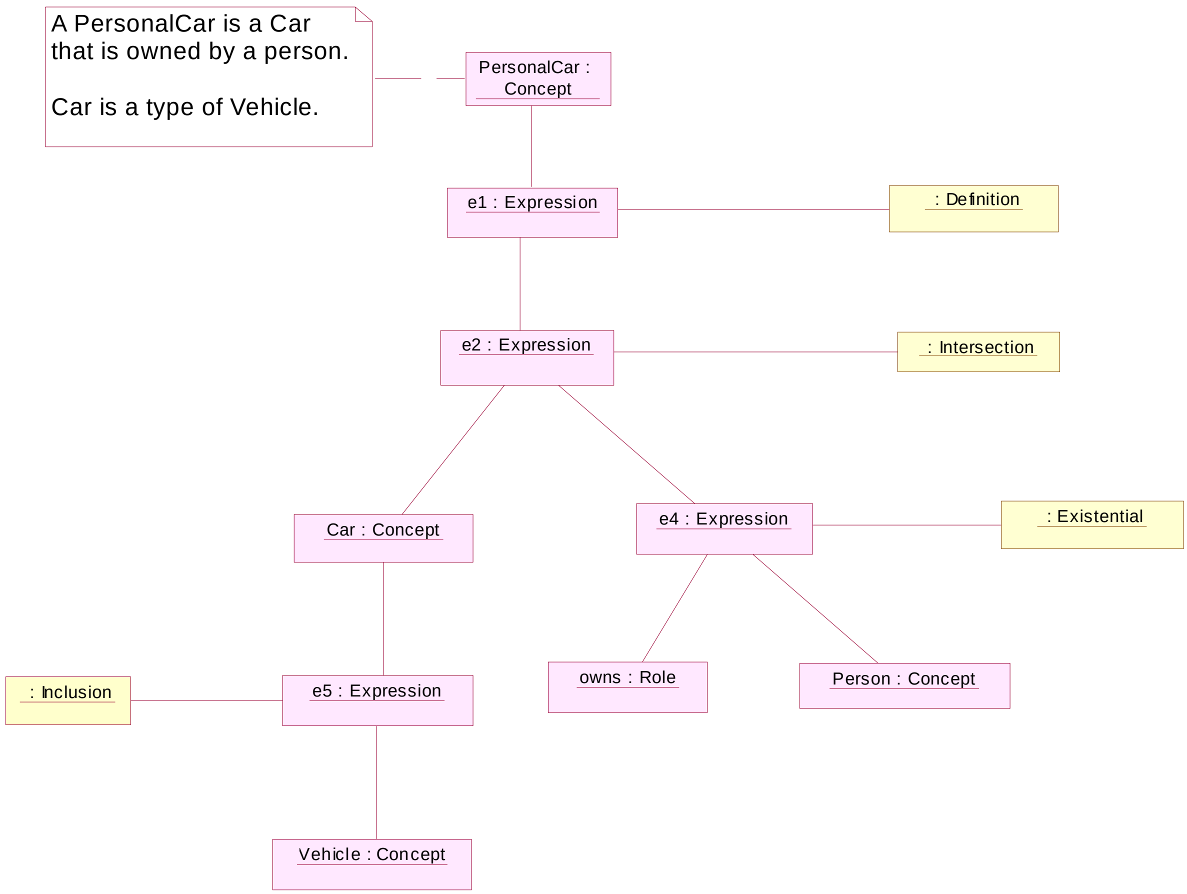

Figure C.7 - Example One

C.7.2 Example Two

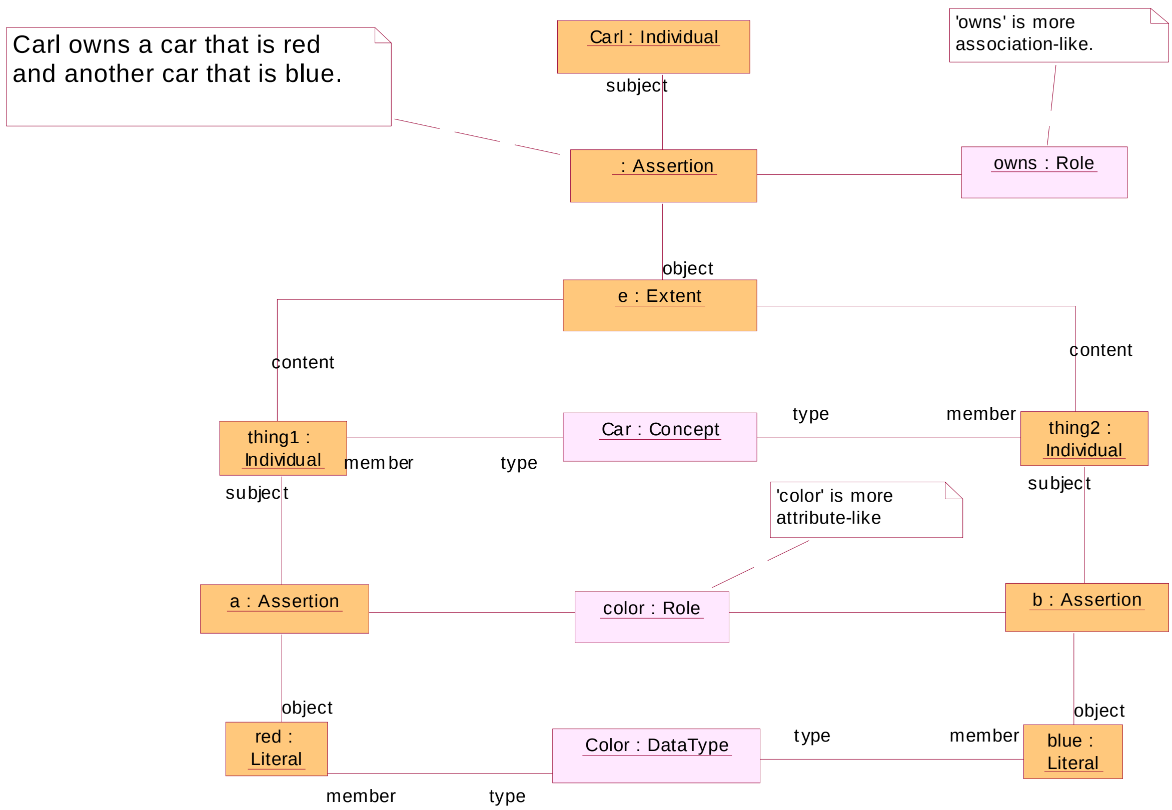

Figure C.8 - Example Two

C.8 Overview Diagram

Figure C.9 provides a overview of the complete class hierarchy and key associations in the DL meta-model.

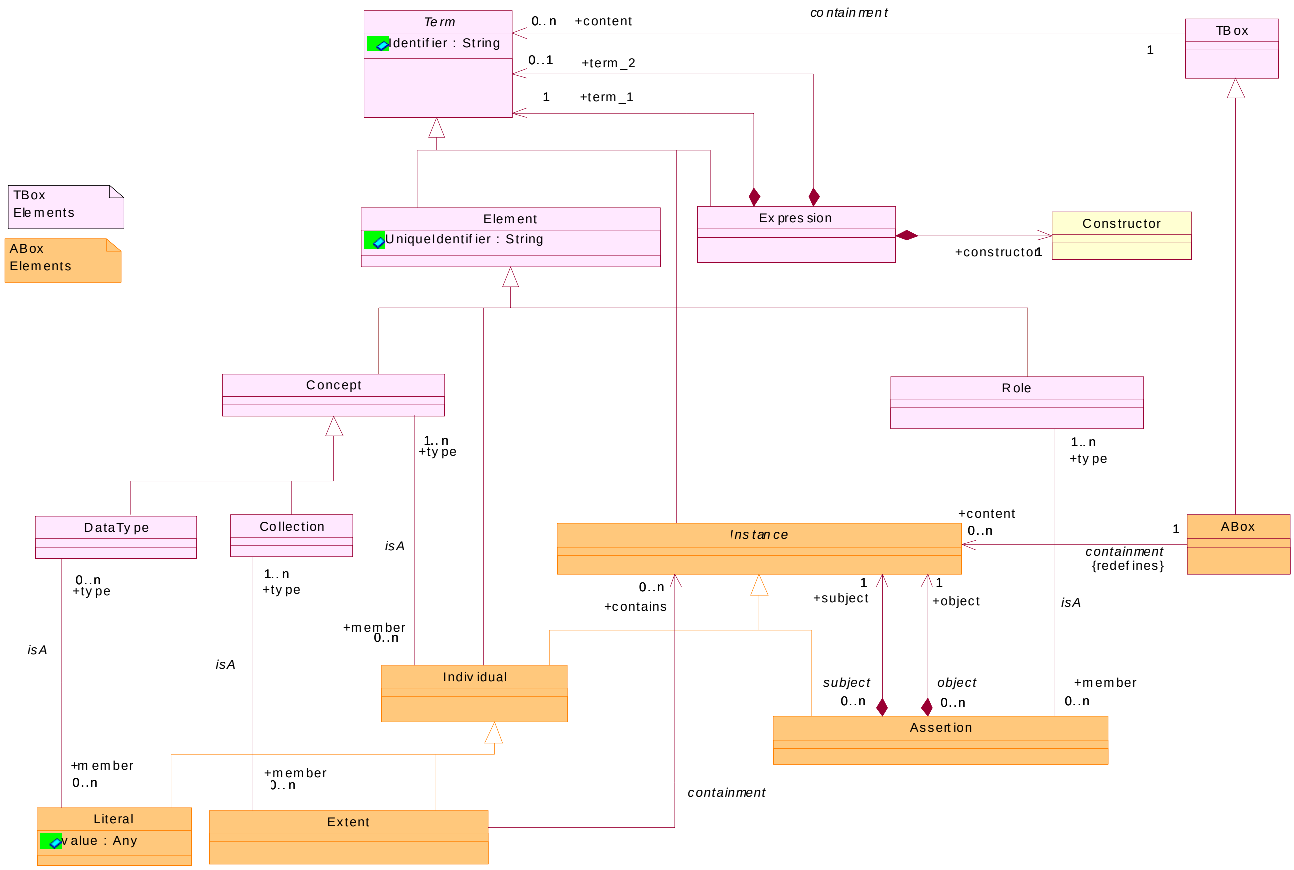

Figure C.9 - Complete DL Metamodel

Annex D
Mapping UML to OWL

D.1 Introduction

This annex provides an informative comparison between UML and the mandated ontology representation language OWL. It compares the features of OWL Full (as summarized in OWL Web Ontology Language Overview [OWL OV]) with the features of UML 2.0 [UML2]. It first looks at the features the two have in common, although sometimes represented differently, then reviews the features that are prevalent in one but not the other. Little attempt is made to distinguish the features of OWL Lite or OWL DL from those of OWL Full. This overview also ignores secondary features such as headers, comments, and version control. In the features in common, a sketch is given of the translation from a model expressed in UML to an OWL expression. In several cases, there are alternative ways to translate UML constructs to OWL constructs. This annex selects a particular way in each case, but the translation is not intended to be normative. In particular applications, other choices may be preferable.

This annex also includes informative formal mappings from UML to OWL and from OWL to UML, both expressed in QVT [MOF QVT].

UML models are defined within a larger meta-modeling framework standardized by OMG. This framework defines a series of metalevels for UML:

- M3 is the MOF, the universal modeling language in which modeling systems are specified.
- M2 is the model of a particular modeling system. The UML metamodel is an M2 construct, as it is specified in the M3 MOF.
- M1 is the model of a particular application represented in a particular modeling system. The UML Class diagram model of an order entry system is an M1 construct expressed in the M2 metamodel for the UML Class diagram.
- M0 is the population of a particular application. The population of a particular order entry system at a particular time is an M0 construct.

D.2 Features in Common (More or Less)

D.2.1 UML Kernel

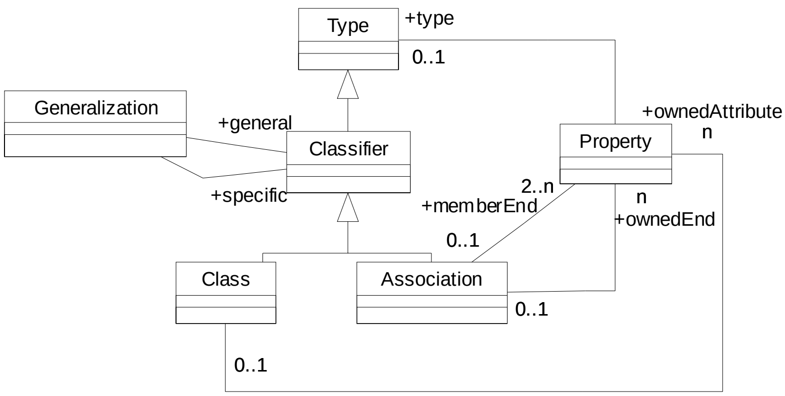

Figure D.1 - Key Aspects of UML Class Diagram

The M2 model definition is quite different from OWL. We need to understand the relationship between an M1/M0 model in UML and the equivalent model in OWL, so we need to understand how the M1 model is represented in the M2 structure shown. First, a few observations from Figure D.1.

- Most of the content of a UML model instance is in the M1 specification. The M0 model can be anything that meets the specification of the M1 model.
- There is no direct linkage between Association and Class. The linkage is mediated by Property.
- A Property is a structural feature (not shown), which is typed. The M1 model is built from structural features.
- Both Class and Association are types.
- A class can have properties that characterize instances of the class.
- A property may or may not be owned by a class. A property may be either *navigable* or *not navigable*. Associations ends are properties.

It will help if we represent a simple M1 model in this structure (Figure D.2).

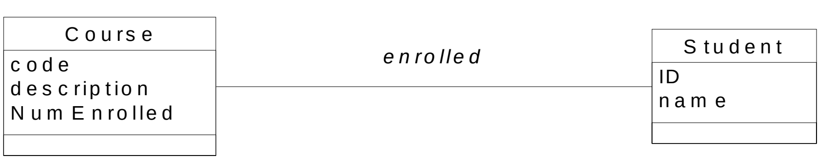

Figure D.2 - Simple M1 Model

The properties with their types are listed in Table D.1.

Table D.1 - Properties and Types in Simple Model

Property	Type
code	CourseIdentifier
description	string
NumEnrolled	integer
ID	StudentIdentifier
name	string

The classes are: Course, Student.

Classes are represented by sets of *ownedAttribute* properties.

Table D.2 - Classes and Owned Properties in Simple Model

Class	ownedAttribute Properties
Course	code, description, NumEnrolled
Student	ID, name

Associations are: enrolled.

The association can be modeled in a number of different ways, depending on how classes are implemented. If classes are implemented as in Table D.2, one way is as the disjoint union of the owned attributes of the two classes.

Table D.3 - Implementation of Association in Simple Model

Association	Implementation
enrolled	code, description, NumEnrolled, ID, name

But there are other ways to implement a class. If it is known that the property *code* identifies instances of *Course* and that the property *ID* identifies instances of *Student*, then an alternative implementation of *enrolled* is shown below:

Table D.4 - Alternative Implementation of Association in Simple Model

Association	Implementation
enrolled	code, ID

In this case, the properties *code* and *ID* would be of type *Course* and *Student* respectively.

D.2.2 Class and Property - Basics

Both OWL and UML are based on classes. A **class** in OWL is a set of zero or more **instances**. A class in UML is a more general construct, but one of its uses is as a set of instances. The set of instances associated at a particular time with a class is called the class' **extent**. There are subtle differences between OWL classes and UML classes that represent sets.

In UML the extent of a class is a set of zero or more instances of M0 objects. (Instances may be specified at the M1 level in a model library, but they specify possibly several M0 objects.) An instance consists of a set of slots each of which contains a value drawn from the type of the property of the slot. The instance is associated with one or more classifiers. An instance of the class *Course* might be as shown in Table D.5.

Table D.5 - Example Course Instance

Classifier	code	description	NumEnrolled
Course	INFS3101	Ontology and the Semantic Web	0

In OWL, the extent of a class is a set of individuals identified by URIs. Individual is defined independently of classes. There is a universal class *Thing* whose extent is all individuals in a given OWL model, and all classes are subclasses of *Thing*. The main difference between UML and OWL in respect of instances is that in OWL an individual may be an instance of Thing and not necessarily any other class, so could be outside the system in a UML model. It is of course possible to include a universal class in an M1 model library, but the concept is central to OWL.

An OWL class is declared by assigning a name to the relevant type. For example:

```
<owl:Class rdf:ID=”Course”/>
```

An individual is at bottom an RDFS resource, which is essentially a name, so the individual INFS3101 will be declared with something like

```
<owl:Thing rdf:ID=“INFS3101”/>
```

Relationships among classes in OWL are called **properties**. That the class *course* has the relationship with the class *student* called *enrolled*, which was represented in the UML model as the association *enrolled*, is represented in OWL as a property.

```
<owl:ObjectProperty rdf:ID = “enrolled”/>
```

Properties are not necessarily tied to classes. By default, a property is a binary relation between *Thing* and *Thing*.

So, in order to translate the M1 model of Figure D.2 to OWL, UML Class goes to owl:Class.

Table D.6 - Simple Model Classes Translated to OWL

Class	OWL equivalent
Course	`<owl:Class rdf:ID="Course"/>`
Student	`<owl:Class rdf:ID="Student"/>`

The relationships among classes represented in OWL by owl:ObjectProperty and owl:DatatypeProperty come from two different sources in the UML model. One source is the M2 association *ownedAttribute* between Class and Property, which generates the representation of a class as a bundle of owned attributes as in Table D.2. An M1 instance of *Class ownedAttribute Property* would translate as properties whose domain is *Class* and whose range is the type of *Property*.

The UML *ownedAttribute* instance would translate to owl:ObjectProperty if the type of *Property* were a UML Class, and owl:DatatypeProperty otherwise. The translation of Table D.2 is shown in Table D.7. Note that UML *ownedAttribute* M2 associations are distinct, even if *ownedAttributes* have the same name associated with different classes. The owl property names must therefore be unique. One way to do this is to use a combination of the class name and the owned property name. Note also that since instances of *ownedAttribute* are always relationships among types, the equivalent OWL properties all have domain and range specified.

An alternative way to give domain and range to OWL properties is to use restriction to allValuesFrom the range class when the property is applied to the domain class. This is probably a more natural OWL specification. However, since all OWL properties arising from a UML model are distinct, the method employed in this chapter is adequate. Should a translation of a UML model be intended as a base for further development in OWL, an appropriate translation can be employed (see D.23).

Table D.7 - Simple Model Associations Translated to OWL

Class	**Owned property**	**Type of owned property**	**OWL equivalent**
Course	code	CourseID	<owl:ObjectProperty rdf:ID="CourseCode"> <rdfs:domain rdf:resource="Course"/> <rdfs:range rdf:resource="CourseID"/> </owl:ObjectProperty>
	description	string	<owl:DatatypeProperty rdf:ID="CourseDescription"> <rdfs:domain rdf:resource="Course"/> <rdfs:range rdf:resource="http://www.w3.org/2001/ XMLSchema#string"/> </owl:DatatypeProperty>
	Num Enrolled	integer	<owl:DatatypeProperty rdf:ID="CourseEnrolled"> <rdfs:domain rdf:resource="Course"/> <rdfs:range rdf:resource="http://www.w3.org/2001/ XMLSchema#integer"/> </owl:DatatypeProperty>
Student	ID	StudentIdent	<owl:ObjectProperty rdf:ID="StudentID"> <rdfs:domain rdf:resource="Student"/> <rdfs:range rdf:resource="StudentIdent"/> </owl:ObjectProperty>
	name	string	<owl:DatatypeProperty rdf:ID="StudentName"> <rdfs:domain rdf:resource="Student"/> <rdfs:range rdf:resource="http://www.w3.org/2001/ XMLSchema#string"/> </owl:DatatypeProperty>

Note that the translation in Table D.7 assumes that a single name is an identifier for instances of the corresponding class. This is not always true. That is, there are cases in which a relational database implementation would use a compound key to identify an instance of a class. The translation of the UML class, when using compound keys, would construct a unitary name for an OWL individual from the value of the individual property. For example, if the association *enrolled* were treated as a class (UML association class), its representing property might be a concatenation of Course.code and

Student.id, so that the link for student 1234 enrolled in course INFS3101 might be translated to an OWL individual with name a globalized equivalent of 1234.INFS3101. Alternatively, a system-defined name could be assigned, linked to each name in the compound key by system-defined properties.

The second source of owl properties in a UML M1 model is the M1 population of the M2 class *association (see 16.2.3 for details)*.

Table D.8 - Sample Associations Translated to OWL

Association	**Assn end 1 Property Type**	**Assn end 2 Property Type**	**OWL equivalent**

enrolled	Course	Student	`<owl:ObjectProperty rdf:ID="enrolled">`
			`<rdfs:domain rdf:resource="Course"/>`
			`<rdfs:range rdf:resource="Student"/>`
			`</owl:ObjectProperty>`

Both languages support the **subclass** relationship (OWL rdfs:subClassOf, UML generalization). Both also support **subproperties** (UML generalization of association or meta-associations among properties like subsetting or redefining). UML defines generalization at the supertype *classifier*, while in OWL subtype and subproperty are separately but identically defined.

The translation from UML to OWL is straightforward. If <S, G> is an M1 instance of the UML M2 metaclass *generalization* (S is a subclassifier of G), then if both S and G are classes and TS, TG are respectively the types of the identifying owned property of S, G respectively, the OWL equivalent is the addition of the clause <rdfs:subClassOf rdf:resource="TG"/> to the definition of the OWL class TS. Similarly if S and G are both associations, the owl equivalent is the addition of the clause <rdfs:subPropertyOf rdf:resource="G"/> to the definition of the OWL object property S. Note that subassociations can be defined in a number of ways, including by OCL.

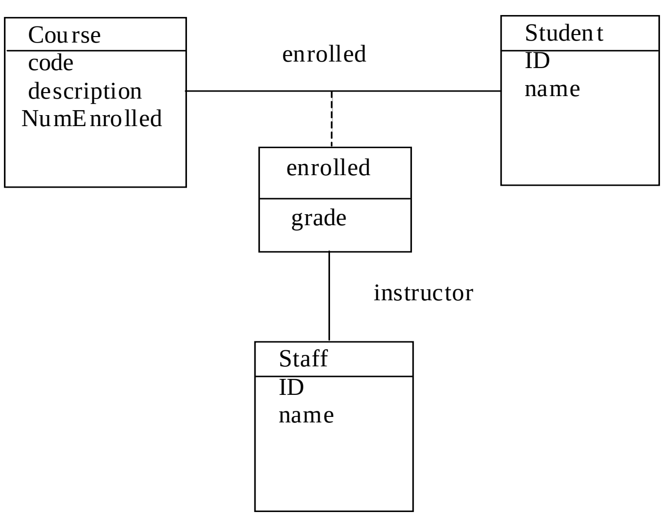

Figure D.3 - M1 Model with Association Class

An association in UML can be N-ary. It can have its own ends (*ownedEnd*). An association can also be a class (*association class*), so can participate in further associations. In OWL DL, classes and properties are disjoint, but in OWL Full they are potentially overlapping. However, there is limited syntactic mechanism in the documents so far published to support this overlap. There is an advantage in translating these more complex UML associations to structures supported by OWL DL. In any case, the translations described are not normative, so those responsible for a particular application can use more powerful features of OWL if there is an advantage to doing so.

This specification takes advantage of the fact that both an N-ary relation among types T_1 ... T_N and an association class with attributes are formally equivalent to a set R of identifiers together with N projection functions P_1, ..., P_N, where P_i:R -> T_i. Thereby both association classes and N-ary UML associations are translated to OWL classes with bundles of binary functional properties.

Figure D.3 extends the model of Figure D.2 by making *enrolled* an association class that owns an attribute *grade*. Some students enrolled in a given course may be assigned to one staff member as instructor, some as another.

The model of Figure D.3 is represented in table form in Table D.9.

Table D.9 - Sample Model Association Classes

Association	Parts	Type
enrolled	end 1	Course
	end 2	Student
	attri-bute	Grade
	Reification	enrolledR
instructor	end 1	enrolledR
	end 2	Staff

The association class *enrolled* is represented by its two end classes, *Course* and *Student*, the attribute of the association class *Grade*, and by an owned attribute *enrolledR* that implements the association class as a class, in the same way as in Table D.3 and Table D.4.

The implementation of *enrolled* and *Instructor* in Table D.9 is translated into OWL as follows:

```
<owl:Class rdf:ID="enrolled" / >
<owl:FunctionalProperty rdf:ID="enrolledCourse">
   <rdfs:domain rdf:resource="enrolled/>
   <rdfs: range rdf:resource="Course"/>
</owl:FunctionalProperty >
<owl:FunctionalProperty rdf:ID="enrolledStudent">
   <rdfs:domain rdf:resource=enrolled/>
   <rdfs: range rdf:resource="Student"/>
</owl:FunctionalProperty >
<owl:FunctionalProperty rdf:ID="enrolledGrade">
   <rdfs:domain rdf:resource=enrolled/>
   <rdfs: range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty >
<owl:FunctionalProperty rdf:ID="enrolledenrolledR">
   <rdfs:domain rdf:resource=enrolled/>
   <rdfs: range rdf:resource=enrolledR/>
</owl:FunctionalProperty >
<owl:FunctionalProperty rdf:ID="instructor">
   <rdfs:domain rdf:resource=enrolledR/>
```

```
    <rdfs: range rdf:resource=Staff/>
</owl:FunctionalProperty >
```

D.2.3 More Advanced Concepts

There are a number of more advanced concepts in both UML and OWL. In the cases where the UML concept occurs in OWL, the translation is often quite straightforward, so will not always be shown in this section (see D.23 and D.24 for full details).

Both languages support a module structure, called **package** in UML and **ontology** in OWL. The translation of package to ontology is straightforward. Both languages also support **namespaces**.

Both UML and OWL support a fixed defined extent for a class (OWL **oneOf**, UML **enumeration**). Note that in UML enumeration is a datatype rather than a class.

UML has the option for associations to have distinguished ends that can be **navigable** or **non-navigable**. A navigable property is one that is owned by a class or optionally an association, while a non-navigable is not (the end might be of type integer, say). OWL properties always are binary and have distinguished ends called **domain** and **range**. A UML binary association with one navigable end and one non-navigable end will be translated into a property whose domain is the non-navigable end. A UML binary association with two navigable ends will be translated into a pair of OWL properties, where one is **inverseOf** the other.

A key difference is that in OWL a property is defined by default as having range and domain both *Thing*. A given property therefore can in principle apply to any class. So a property name has global scope and is the same property wherever it appears. In UML the scope of a property is limited to the subclasses of the class on which it is defined. A UML association name can be duplicated in a given diagram, with each occurrence having a different semantics. It is possible, though not customary, to include a universal superclass in an M1 model library.

An OWL individual can be difficult to represent in a UML model. UML has a facility **dynamic classification** that allows an instance of one class to be changed into an instance of another, which captures some of the features of Individual, but an object must always be an instance of some class. UML models rarely include universal classes.

Both languages allow a class to be a subclass of more than one class (**multiple inheritance**). Both allow any subclasses of a class to be declared **disjoint**. UML allows a collection of subclasses to be declared to **cover** a superclass, that is to say every instance of the superclass is an instance of at least one of the subclasses. The corresponding OWL construct is the declare the superclass to be the union of the subclasses, using the construct **unionOf**.

UML has a strict separation of metalevels, so that the population of M1 classes is distinct from the population of M0 instances and also the M1 model libraries. OWL Full permits classes to be instances of other classes. UML only models classes of classes in the context of declaration of disjoint or covering powertypes.

In OWL, a property restriction applied to a class can impose a cardinality constraint giving the minimum (**minCardinality**), maximum (**maxCardinality**), or exact number of instances that can participate in a relation (of the specified type) with an instance of that class. In addition, an OWL property can be globally declared as functional (**functionalProperty**) or inverse functional (**inverseFunctional**). A functional property has a maximum cardinality of 1 on its range, while an inverse functional property has a maximum cardinality of 1 on its domain. In UML an association can have minimum and maximum cardinalities (**multiplicity**) specified for any of its ends. OWL allows individual-valued properties (objectProperty) to be declared in pairs, one the inverse of the other.

So if a binary UML association has a multiplicity on a navigable end, the corresponding OWL property will have the same multiplicity. If a binary UML association has a multiplicity on both its ends, then the corresponding OWL property will be an inverse pair, each having one of the multiplicity declarations.

For an N-ary UML association, multiplicities are more problematic to map to OWL. For example, in Figure D.4 the multiplicities show that:

- given instances of *event*, *Olympiad*, and *competitor* there is at most one instance of *result*.
- given instances of *event*, *Olympiad*, and *result* there is at most one instance of *competitor.*
- given instances of *Olympiad*, *competitor*, and *result* there may be many instances of event (an athlete may compete at several events in the same Olympiad and finish in the same place in each).
- given instances of *event*, *competitor*, and *result* there may be many instances of *Olympiad* (an athlete may compete in the same event at several Olympiads and finish in the same place in each).

For an N-ary UML association, any multiplicity associated with one of its end classes will apply to the OWL property translating the corresponding projection from the association class to the translated end class.

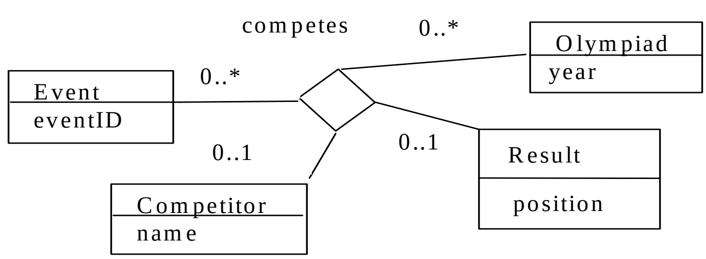

Figure D.4 - Example N-ary Association with Multiplicity

The N-ary association in Figure D.4 would be translated as a class *competes* whose instances are instances of links in the association, and four properties whose domain is *competes* and whose ranges are the classes attached to the member ends of the association. Since one instance of a link includes only one instance of the class at each member end, all the properties are functional. The multiplicities on the UML diagram do not translate to OWL in a straightforward way.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
<!ENTITY owl   "http://www.w3.org/2002/07/owl#">
<!ENTITY rdf   "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<!ENTITY rdfs  "http://www.w3.org/2000/01/rdf-schema#">
<!ENTITY xsd   "http://www.w3.org/2001/XMLSchema#">
]>
<rdf:RDF xmlns:rdf="&rdf;"
         xmlns:rdfs="&rdfs;"
    xmlns:owl="&owl;"
         xmlns:xsd="&xsd;"/>

<owl:Class rdf:ID="competes">
      <owl:subClassOf>
         <owl:Restriction>
        <owl:onProperty rdf:resource="competesEvent"/>
          <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
         </owl:Restriction>
      </owl:subClassOf>
      <owl:subClassOf>
```

```
         <owl:Restriction>
            <owl:onProperty rdf:resource="competesCompetitor"/>
          <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
         </owl:Restriction>
      </owl:subClassOf>
      <owl:subClassOf>
         <owl:Restriction>
            <owl:onProperty rdf:resource="#competesOlympiad"/>
          <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
         </owl:Restriction>
      </owl:subClassOf>
      <owl:subClassOf>
         <owl:Restriction>
            <owl:onProperty rdf:resource="#competesResult"/>
          <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
         </owl:Restriction>
      </owl:subClassOf>
      <owl:subClassOf>
         <owl:Restriction>
            <owl:onProperty rdf:resource="#competesResult"/>
          <owl:maxCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
         </owl:Restriction>
      </owl:subClassOf>
</owl:Class>
<owl:FunctionalProperty rdf:ID="competesEvent">
 <rdfs:domain rdf:resource="#competes"/>
   <rdfs:range rdf:resource="#Event"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="competesCompetitor">
  <rdfs:domain rdf:resource="#competes"/>
     <rdfs:range rdf:resource="#Competitor"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="competesOlympiad">
  <rdfs:domain rdf:resource="#competes"/>
      <rdfs:range rdf:resource="#Olympiad"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="competesResult">
  <rdfs:domain rdf:resource="#competes"/>
     <rdfs:range rdf:resource="&xsd;string> >
</owl:FunctionalProperty>
</rdf:RDF>
```

In UML, multiplicities can be defined on both ends of an association. In OWL, general multiplicities apply to the range instances associated with a given domain instance. In both cases, multiplicities can be strengthened (minimim increased or maximum decreased) when associations/properties are applied to subclasses.

Note that the class might be the domain of a property for which the individual might not have a value. This can happen if the mincardinality of the domain of the property is 0, in which case the property is optional (or partial) for that class. The same can happen in UML. An instance of a class is constrained to participate only in properties that are mandatory, minimum cardinality > 0. So an instance can lack optional properties. (The somewhat strange construct maxCardinality < minCardinality is syntactically correct in OWL and has the semantics that the property has no instances. It can occur where multiple autonomous ontologies are merged, for example.)

However, even if the property is mandatory (mincardinality > 0 and maxcardinality >= mincardinality), there may not be definite values for the property. Consider a class (K) for which a property (P) is mandatory. In this case, the individual (I) must satisfy the predicate

[M]: I instance of K -> exists X such that P(I) = X.

UML and OWL do not require that there is a constant C such that X = C, e.g., all horses have color, but we may not know what color a particular horse has. As consequence of this possible indeterminacy, it may not be possible to compute a transitive closure for a property across several ontologies, even if they share individuals.

In UML, there is a strict separation between the M1 and M0 levels. If an association is mandatory (minimum cardinality greater than 0), it exactly matches the predicate [M]. Any difference between UML and OWL must come from the treatment of the model of the M1 theory at the M0 level. In practice, implementations derived from UML models tend to be ground Herbrand models implemented by something like an SQL database manager. For these cases, if we know a horse has a color, then we know what color it has. To the extent that UML tools and modeling build this expectation into products, conflict can occur when interoperating with an OWL ontology.

But UML does not mandate Herbrand compliance. It is possible for a particular application to introduce a special constant “unknown” into a class, and define special treatment by the programs. UML does not forbid an implementation of a class model in one of these ways. There is no difference in principle between UML and OWL for properties that are declared to have minCardinality greater than 0 (and maxCardinality >= minCardinality) for a class.

An OWL property can have its range restricted when applied to a particular class such that the range is limited (subtype of the property's range if declared) (allValuesFrom). UML permits these and other restrictions using the facilities: subsets, specializes, or redefines. Often the specific restriction is defined in OCL.

OWL allows properties to be declared symmetric (**SymmetricProperty**) or transitive (**TransitiveProperty**). In both cases, if the domain and range are not type compatible, the property is empty. UML uses OCL for this purpose.

OWL permits declaration of a property whose value is the same for all instances of a class, so the property value is in effect attached to the class (OWL DL property declared as allValuesFrom a singleton set for that class). OWL full allows properties to be directly assigned to classes without special machinery. In OWL, if class A is an instance of class B, then a property P whose domain includes B will designate a value P(A) that can apply to the class A so it can be derived for all instances of A.

A classifier in UML can be declared **abstract**. An abstract classifier typically cannot be instantiated, but may be a superclass of concrete classifiers. There is no OWL equivalent for this.

Two different objects modeled in UML may have dependencies that are not represented by UML named (model) elements, so that a change in one (the supplier) requiring a change in the other (the client) will not be signaled by, for example, association links. Two such objects may be declared **dependent**. There are a number of subclasses of dependency, including abstraction, usage, permission, realization, and substitution. OWL does not have a comparable feature except as annotations, but RDF, the parent of OWL, permits an RDF:property relation between very general elements classified by RDFS:Class. Therefore, a dependency relationship between a supplier and client UML model element will be translated to a reserved name RDF:Property relation whose domain and range are both RDF:Class. Population of the property will include the individuals, which are the target of the translation of the supplier and client named elements.

D.2.4 Summary of More-or-Less Common Features

This sub clause has described features of UML and OWL that are in most respects similar. Table D.10 summarizes the features of UML in this feature space, giving the equivalent OWL features. UML features are grouped in clusters that translate to a single OWL feature or a cluster of related OWL features. The column *Package* shows the section of the UML Superstructure document [UML2] where the relevant features are documented.

Table D.10 - Common Features of UML and OWL

UML elements	Package	OWL elements	Comment
class, property ownedAttribute, type[a]	7.3.7 Classes 7.3.8 Classifiers 7.3.32 Multiplicities	class	
instance	7.3.22 Instances	individual	OWL individual independent of class
ownedAttribute, binary association	7.3.7 Classes	property	OWL property can be global
subclass, generalization	7.3.7 Classes 7.3.8 Classifiers	subclass subproperty	
N-ary association, association class	7.3.7 Classes 7.3.4 Association Classes	class, property	
enumeration	7.3.11 Datatypes	oneOf	
disjoint, cover	7.3.21 Generalization sets	disjointWith, unionOf	
multiplicity	7.3.32 Multiplicities	minCardinality maxCardinality	OWL cardinality declared only for range
package	7.3.37 Packages	ontology	
dependency	7.3.12 Dependencies	reserved name RDF:property	

a. This cell summarizes the relationship between UML class and OWL class mediated by property, own-edAttribute and type. It does not signify that the latter three are themselves translated to OWL class.

All of the UML features considered in the scope of the ODM have more-or-less satisfactory OWL equivalents. Some UML features have no OWL equivalents, as summarized in Table D.11. Some OWL features in this feature space have no UML equivalent, so are omitted from Table D.10. They are summarized in Table D.12. Besides the small differences in the features in the feature space common to UML and OWL, there are some more general differences described in D.25 and D.26.

Table D.11 - UML features with no OWL equivalent

navigable, non-navigable
derived
abstract classifier

Table D.12 - OWL features with no UML equivalent

Thing, global properties, autonomous individual
someValuesFrom
SymmetricProperty, TransitiveProperty
Classes as instances (in OWL Full)
complementOf

D.3 UML to OWL

This sub clause describes mappings from [UML2] models to ODM OWL models. The UML2 metamodel is based on ptc/04-10-02. This mapping is limited to OWL DL, which means only OWL-DL constructs will be used in mapping definitions. There are many abstract metaclasses in UML2 kernel package, so only important concrete classes are mapped to OWL constructs.

Mappings are expressed in QVT [MOF QVT]. A brief tutorial is presented in Annex H.

Mappings are shown for all constructs in Table D.10 for which there is an OWL equivalent, except for Instance. The Instance model is not part of the class model, and is intended to show partial specifications of instances rather than concrete instances.

D.3.1 Naming Issues

In OWL, all objects are identified either by uniform resource identifiers (URI) or optionally by an arbitrarily assigned identifier unique within the ontology (blank nodes). A typical method is for elements within an ontology to be identified by URI reference, which adds a local name to a base URI that identifies the ontology. It is also possible for an element to have a URI independent of that of the ontology. Blank node identifiers can be treated as local names during the course of the mapping, even though they do not persist. A URI is conceptually global. It universally identifies the same element no matter where it appears.

In UML, elements can be identified by name within a minimally disambiguating context. If there are several packages involved in a mapping, the packages themselves must have unique names within the scope of the mapping and provide scope for the elements they contain. But other packages may exist elsewhere that have the same name. Within a package, classes, associations, and other elements can be identified by names unique to the package. Lower level elements such as properties are identified by names unique within their parent element. For example, several different classes may have attributes with the same name.

Ontologies in OWL are free-standing objects that can import one another. Packages in UML can also import one another, but in addition there is a standard procedure by which several packages may be merged into one.

A critical problem in mapping between UML and OWL is the generation of appropriate identifiers for objects in the target model instance given the identifiers of the relevant objects found in the relevant pattern in the source model instance. Since the mappings proceed from the packaging constructs to their components, the first problem is generation of an identifier for the target packaging construct given the identifier of the source packaging construct. If the source is an OWL ontology, one possibility is to identify the target package with the same uri as the ontology. However, this method violates the spirit of the uri, since the same uri now identifies two different objects that could evolve independently. If the source is a package, a base uri must be constructed for the target ontology. There is not enough information available in the UML model instance to generate a globally unique uri.

Because of these incompatibilities, we have made use of two only partly specified relations, PackageNameToUriBase and URIRefToName. PackageNameToUriBase takes a package name and creates a uri suitable to be extended by fragment identifiers. URIRefToName takes a uri reference, possibly a fragment on a base uri, and generates a name unique to the uri reference. (The relation takes distinct uri references to distinct names.)

Further, in mapping from UML to OWL, the target of objects whose names are unique within packages are identified by uri references that are fragments on the uri base of the corresponding ontology. Targets of objects whose names are unique only within a narrower context are identified by fragment identifiers generated by concatenating the name of the source object with the names of its context objects starting with the object whose name is unique to a package. Thus an attribute bar of a class foo would map to an object with fragment identifier foobar.

D.3.2 Package To Ontology

Each object in both the source and target model instance must have an identification scheme. Both UML and OWL support the concept of a namespace represented as a packaging construct, called Package in UML and ontology in OWL. Individual objects are contained in packaging constructs, and identified with respect to the identifier of their packaging construct.

A package is a namespace for its members, and may contain other packages. Only packageable elements can be owned members of a package. By virtue of being a namespace, a package can import either individual members of other packages, or all the members of other packages.

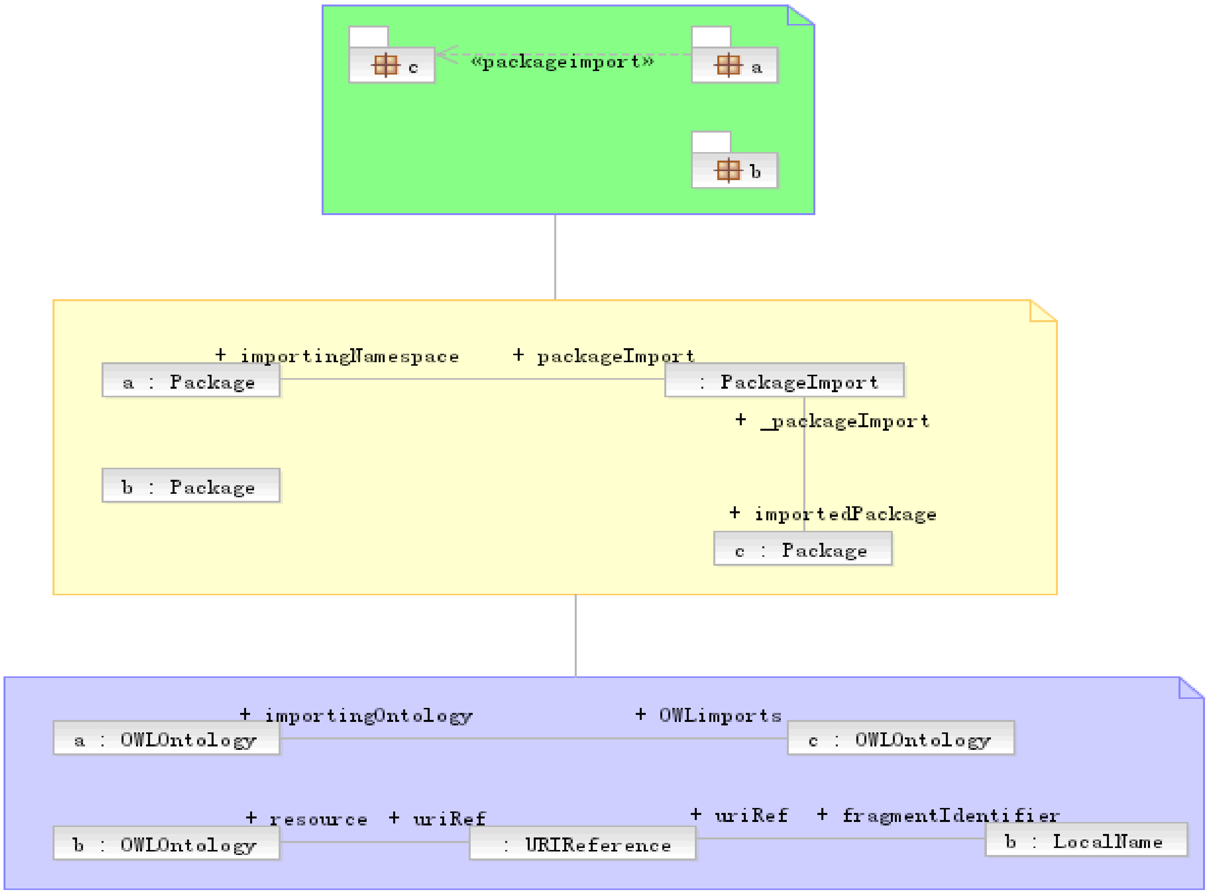

Figure D.5 - Map UML Package to OWL Ontology

```
transformation UMLToOWL (uml:UML, owl:OWL)
// transform UML model to OWL model
{

// Objects in UML have names relative to other constructs, ultimately to Package
key Package(name);
key Class(name, owningPackage);
key Association(name, owningPackage);
key UML::...::Kernel::Property(name, class);
key UML::...::Kernel::Property(name, association);

// All objects in an OWL model instance are instances of OWLUniverse. Figure D.10.
key OWLOntology(uriRef);
key OWLUniverse (uriRef, ontology);
key OWLUniverse (nodeID, ontology); // anonymous classes

top relation PackageToOntology
//map packages in UML to ontologies in OWL
{
    pn:String;
    checkonly domain uml p: Package {name=pn};
```

```
   enforce domain owl o: OWLOntology{uriRef=ref:URIReference};
   when {
          PackageNameToUriBase (pn, ref)
   }
}// PackageToOntology

relation PackageNameToUriBase
{
   primitive domain pn: string;
   enforce domain owl ref: URIReference{};
// Details of this relation are left to specific mappings
} // PackageNameToUriBase

top relation ImportedPackageToOWLImports
//map imported packages in UML to OWLImports links in OWL
{
   checkonly domain uml p: Package{packageImport = :PackageImport
          {importedPackage = imp : Package}};
   enforce domain owl o: OWLOntology{OWLImports= io:OWLOntology};
   when{
          PackageToOntology(p, o);
          PackageToOntology(imp, io);
   }
}//ImportedPackageToOWLImports
```

D.3.3 Class To Class

The mapping from Class to OWLClass includes the transformation of generalization relationships between Classes.

A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. It has the same semantics of RDFSsubClassOf in RDF Schema, and the two ends of the generalization relationships can be accessed by the source and target that are defined in DirectedRelationship.

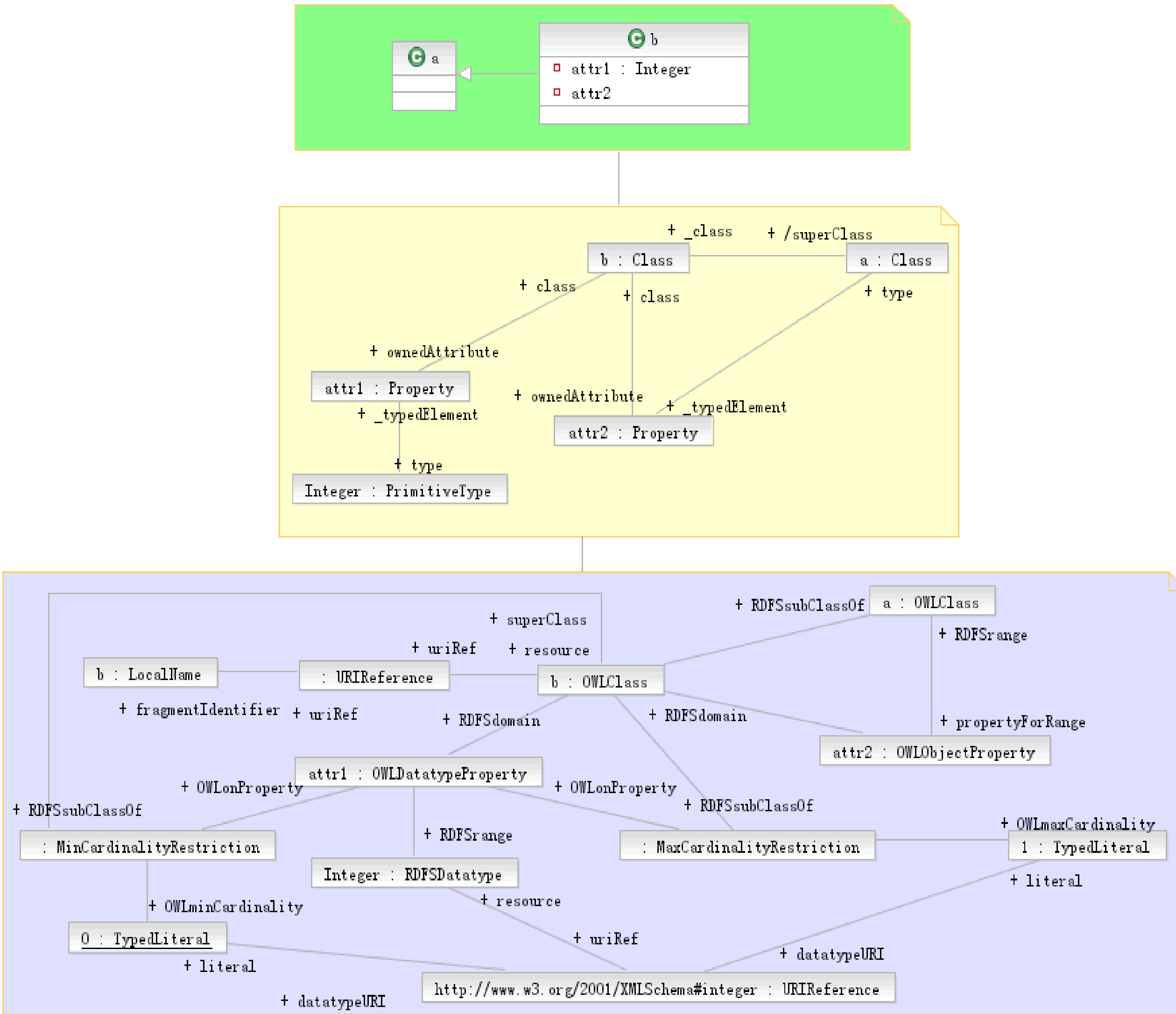

Figure D.6 - Map UML Class to OWL Class [1]

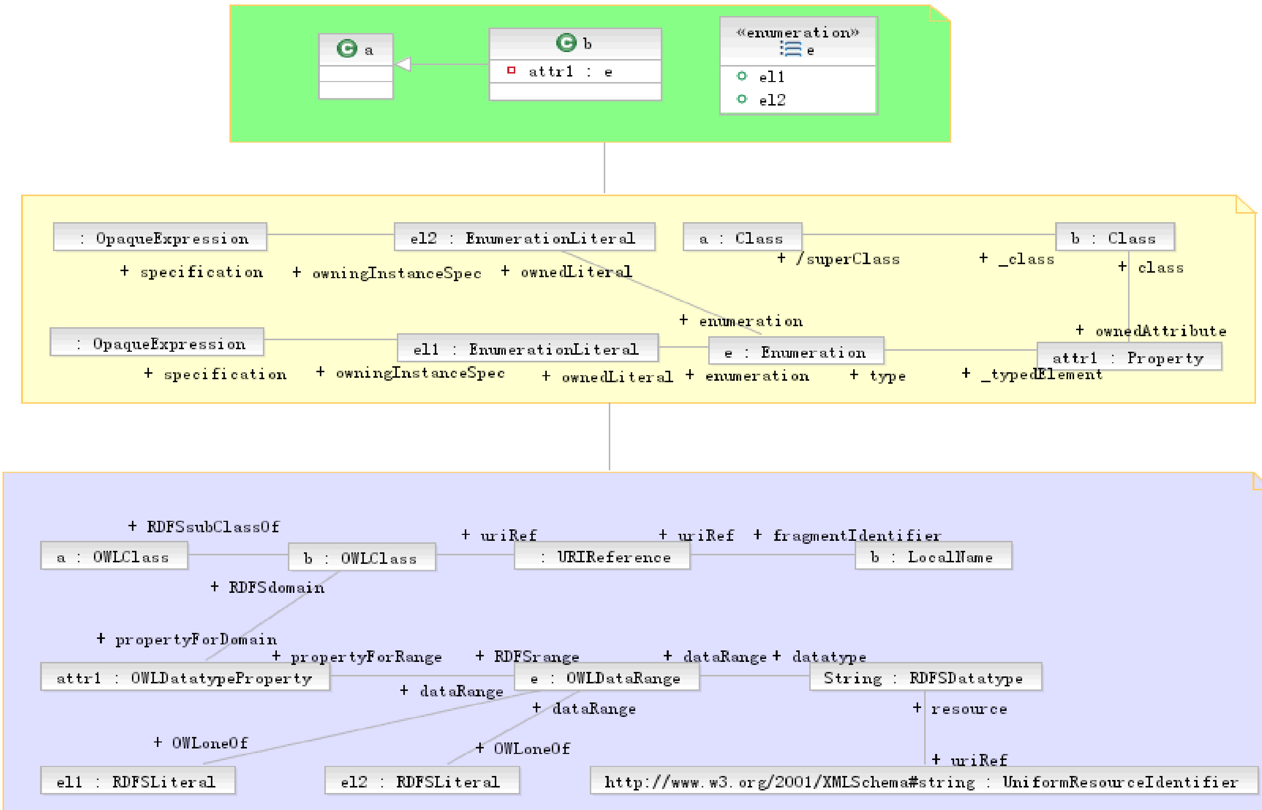

Figure D.7 - Map UML Class to OWL Class [2]

```
top relation UClassToOClass
//map UML Class to OWL Class
{
    cn:String;
    checkonly domain uml uc:Class{name=cn, owningPackage=p:Package};
    enforce domain owl oc:OWLClass{uriRef=:URIReference {uri = ref : UniformResourceIdenti-
    fier,
                    fragmentIdentifier=:LocalName{name=cn}}, ontology=o};
    // no need to name the larger constructs unless they are used or generated elsewhere
    when{
            PackageToOntology(p, o);
            ref = o.uri;  // Provides a base uri for the fragment identifier
    }
}//UClassToOClass

top relation GeneralizationToSubClassOf
//map generalization hierarchy to rdfs:subClassof
{
    checkonly domain uml uc:Class{superClass=gen:Class};
    enforce domain owl oc:OWLClass{RDFSsubClassOf=super:OWLClass};
    when{
            UClassToOClass(uc, oc);
            UClassToOClass(gen, super);
```

```
    }
}// GeneralizationToSubClassOf
```

D.3.4 Attribute to Property

The ownedAttribute defines the attributes owned by the class. It is an ordered set of Properties, which can be mapping to either OWLDatatypeProperty or OWLObjectProperty. If a property is part of the memberEnds of an Association, the mapping of it will be discussed in Association mapping sub clauses [defined in D.23.5 and D.23.6].

If the type of the property is a PrimitiveType, the property is mapped to the OWLDatatypeProperty. If the type of the property is an Enumeration, and the ownedLiteral of the Enumeration has specification as ValueSpecification, then the property is OWLDatatypeProperty.

If the type of the property is Class, or the ownedLiteral of an Enumeration type has at least one classifier, (which does not map to a member of the XSD model library or some user defined type), the property can be mapped to OWLObjectProperty.

```
top relation AttributeToObjectProperty
// maps ownedAttribute where property's type is a class to an object property
// notice that the enforce creates the more specific object OWLObjectProperty, then the more
   general structure Property is created in the where clause. The when clause both forces the
   relation to wait until the classes have been mapped, and gives the mappings for the
   classes.
{
   checkonly domain uml prop : Property{class = cl : Class, type = tp : Class};
   enforce domain owl op : OWLObjectProperty{RDFSdomain = dom : OWLClass, RDFSrange = ran :
   OWLClass};
   when {
          UClassToOClass(cl, dom);
          UClassToOClass(tp, ran);
   }
   where {
          PropertyToProperty(prop, op);
   }
} // AttributeToObjectProperty

relation PropertyToProperty
// not a top relation. Intended to be called in a where clause of the relation mapping one of
   the more specific metaclasses. It fills in the structures relevant to the superclass Prop-
   erty. Assumes the naming conventions for a property used as an attribute.
{
   cn, pn : string;
   checkonly domain uml prop : Property{name = pn, class = :Class{
          name = cn, owningPackage=p:Package}};
   enforce domain owl op : Property{uriRef=:URIReference
          {uri = ref : UniformResourceIdentifier, fragmentIdentifier=:LocalName{name=cnpn}},
   ontology=o};
   when{
          PackageToOntology(p, o);
          ref = o.uri;  // Provides a base uri for the fragment identifier
   }
   where {
          cnpn = cn + pn; // Follows naming conventions to disambiguate property name
```

```
    }
} // PropertyToProperty

top relation AttributeToDatatypeProperty
// maps ownedAttribute where property's type is a primitive type to datatype property
{
    checkonly domain uml prop : Property{class = cl : Class, type = tp : PrimitiveType};
    enforce domain owl op : OWLDatatypeProperty{RDFSdomain = dom : OWLClass, RDFSrange = ran :
    Literal};
    when {
            UClassToOClass(cl, dom);
            UMLPrimTypeToLiteral(tp, ran};
    }
    where {
            PropertyToProperty(prop, op);
    }
} // AttributeToDatatypeProperty

top relation UMLPrimTypeToLiteral
// maps UML primitive type names to OWL literal type names
// To be implemented
{} // UMLPrimTypeToLiteral
```

D.3.5 Binary Association To Object Property

A binary association specifies a relationship that can occur between typed instances. It has exactly two ends represented by properties, each of which is connected to the type of the end. The AssociationToObjectProperty relation is used to set OWLinverseOf relationships between inverse properties.

Note that in this strategy the UML association name is not mapped, so even though the OWL to UML mapping recognizes inverse pairs and maps them to associations, the association name is not recoverable.

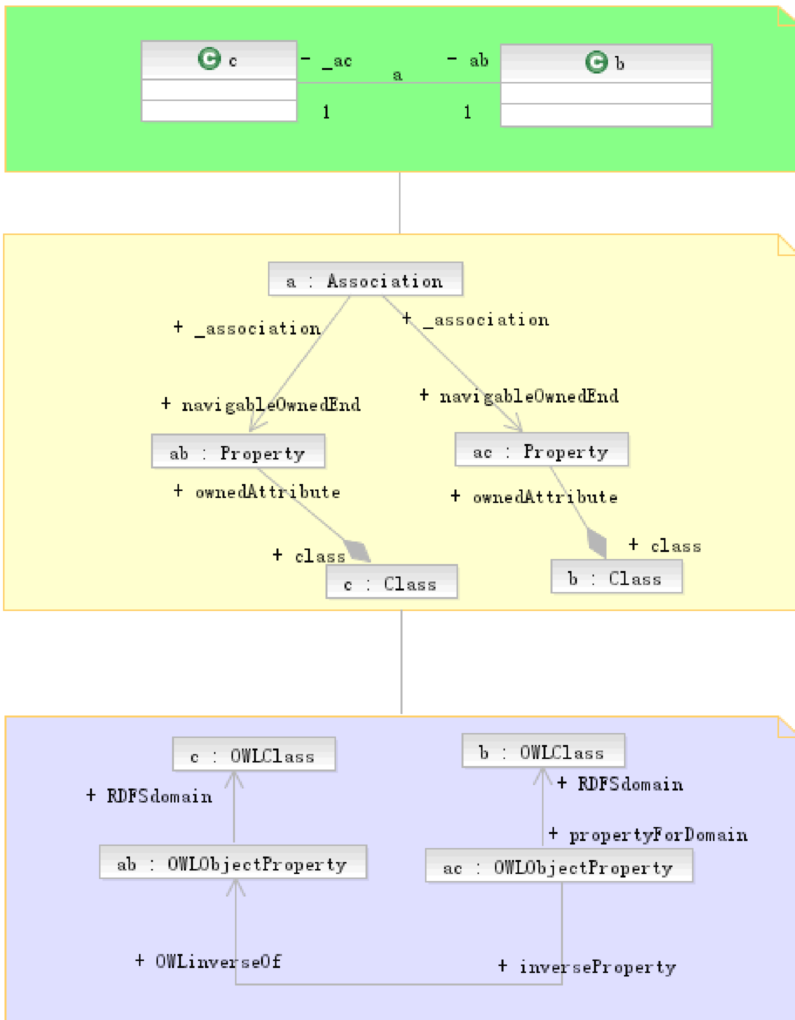

Figure D.8 - Map UML Association to OWL ObjectProperty

```
top relation AssociationToPropertyPair
// Association whose ends are of different types goes to pair of inverse properties
{
    checkonly domain uml assn : Association{
            memberEnd = ps : Sequence(Property){
                    prop1 : Property {type = tp1:Class},
                    prop2 : Property{type = tp2 : Class}}};
// Note checkonly clause succeeds only when there are exactly two memberEnds
    enforce domain owl oprop1 : OWLObjectProperty{
            RDFSdomain = cl2:OWLClass, RDFSrange = cl1 : Class, OWLinverseOf = oprop2 : OWLOb-
    jectProperty{
                     RDFSdomain = cl1:OWLClass, RDFSrange = cl2 : Class}};
    when {
            not prop1.type = prop2.type;  // ends are of different type
            UClassToOClass(tp1, cl1);
```

```
        UClassToOClass(tp2, cl2);
    }
    where {
        EndPropertyToProperty(prop1, oprop1);
        EndPropertyToProperty(prop2, oprop2);
    }
} // AssociationToPropertyPair


relation EndPropertyToProperty
// not a top relation. Intended to be called in a where clause of the relation mapping one of
    the more specific metaclasses. It fills in the structures relevant to the superclass Prop-
    erty. Assumes the naming conventions for a property used as an association end.
{
    an, pn : string;
    checkonly domain uml prop : Property{name = pn, association = :Association{
        name = an, owningPackage=p:Package}};
    enforce domain owl op : Property{uriRef=:URIReference
        {uri = ref : UniformResourceIdentifier, fragmentIdentifier=:LocalName{name=anpn}},
    ontology=o};
    when{
        PackageToOntology(p, o);
        ref = o.uri;  // Provides a base uri for the fragment identifier
    }
    where {
        anpn = an + pn; // Follows naming conventions to disambiguate property name
    }
} // EndPropertyToProperty

top relation SymAssociationToSymProperty
// Association where both properties are of the same type to symmetric property
{
    checkonly domain uml assn : Association{
        memberEnd = ps : Sequence(Property) {
            prop1 : Property{type = tp :Class},
            prop2 : Property{type = tp :Class}}};
// checkonly clause succeeds only if there are exactly two member ends.
    enforce domain owl oprop1:SymmetricProperty{RDFSdomain = cl:OWLClass, RDFSrange = cl:OWL-
    Class};
    when {
        UClassToOClass(tp, cl);
    }
    where {
        EndPropertyToProperty(prop1, oprop1);
    }
} // SymAssociationToSymProperty
```

D.3.6 Association Classes and N-ary Associations

An AssociationClass can be seen as an association that also has class properties, or as a class that also has association properties. It not only connects a set of classifiers but also defines a set of features that belong to the relationship itself and not to any of the classifiers.

Both association classes and N-ary associations are mapped to a class that is the domain of properties derived from each of its ends.

Association Class

```
top relation AssociationClassToOWLClass
{
   checkonly domain uml asc : AssociationClass{};
   enforce domain owl cl : OWLClass{};
   where {
         UClassToOClass(asc, cl);
   }
} // AssociationClassToOWLClass

top relation AssociationClassToOWLProps
// Generates the properties coming from the member ends.
{
   checkonly domain uml asc : AssociationClass{memberEnd = uprop : Property{type = tp}};
   enforce domain owl oprop : OWLObjectProperty{RDFSdomain = cl : OWLClass,
         RDFSrange = rancl : OWLClass};
   when {
         AssociationClassToOWLClass(asc,cl);
         UClassToOClass(tp, rancl);
         EndPropertyToProperty(uprop, oprop);
   }
} // AssociationClassToOWLProps

top relation OwnedEndAttributeToObjectProperty
// maps ownedEnd property where property's type is a class to an object property
{
   checkonly domain uml asc : AssociationClass{ownedEnd = uprop : Property{type = tp}};
   enforce domain owl op : OWLObjectProperty{RDFSdomain = dom : OWLClass, RDFSrange = ran :
   OWLClass};
   when {
         AssociationClassToOWLClass(asc,dom);
         UClassToOClass(tp, ran);
   }
   where {
         PropertyToProperty(uprop, op);
   }
} // OwnedEndAttributeToObjectProperty

top relation OwnedEndAttributeToDatatypeProperty
// maps ownedAttribute where property's type is a primitive type to datatype property
{
   checkonly domain uml asc : AssociationClass{ownedEnd = uprop : Property{type = tp : Primi-
   tiveType}};
   enforce domain owl op : OWLDatatypeProperty{RDFSdomain = dom : OWLClass, RDFSrange = ran :
   Literal};
   when {
         AssociationClassToOWLClass(asc,dom);
         UMLPrimTypeToLiteral(tp, ran};
   }
   where {
         PropertyToProperty(prop, op);
   }
} // OwnedEndAttributeToDatatypeProperty
```

N-ary Association

```
top relation NaryAssociationToOWLClass
// Maps n-ary association where n > 2 to an OWL class
{
    checkonly domain uml asc : Association{};
    enforce domain owl cl : OWLClass{};
    when {
        asc.memberEnd->size() > 2; // only associations with more than two ends
    }
    where {
        UClassToOClass(asc, cl);
    }
} // NaryAssociationToOWLClass

top relation NaryAssociationToOWLProps
// Generates the properties coming from the member ends.
{
    checkonly domain uml asc : Association{memberEnd = uprop : Property{type = tp}};
    enforce domain owl oprop : OWLObjectProperty{RDFSdomain = cl : OWLClass,
        RDFSrange = rancl : OWLClass};
    when {
        NaryAssociationToOWLClass(asc,cl);
        UClassToOClass(tp, rancl);
        EndPropertyToProperty(uprop, oprop);
    }
} // NaryAssociationToOWLProps
```

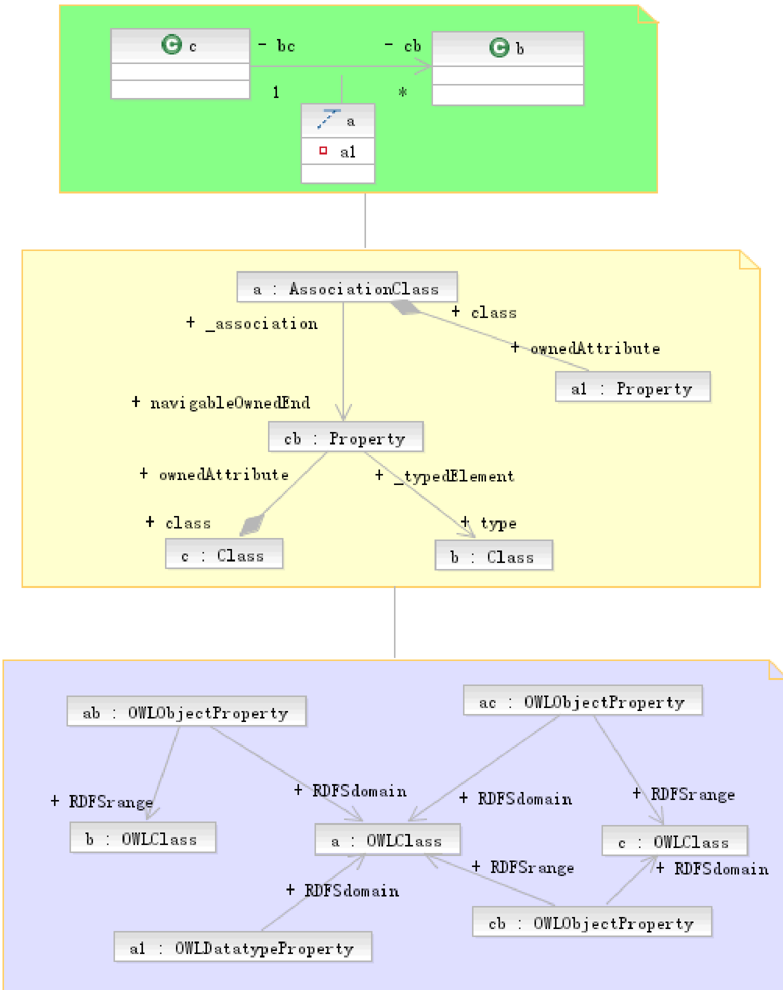

Figure D.9 - Map UML AssociationClass to OWL

D.3.7 Multiplicity

In UML, property is a MultiplicityElement, which defines upperValue and lowerValue to express cardinality. However, OWL uses Restrictions to represent Cardinality. So in addition to map Class to OWLClass, some OWLRestrictions will be generated based on multiplicity definitions of the ownedProperties and corresponding RDFSsubClassOf relationships between OWLClass and OWLRestriction will also be created.

The relation mapping multiplicity can be a top relation, since having PropertyToProperty in the when clause delays its execution until the RDFSdomain = structure in the enforce clause already exists. It is possible that there are several domains for the property. This relation will force them all to be subclasses of the restriction class.

```
top relation UpperMultToMaxCard
// Upper multiplicity
{
    m, n : string;
    checkonly domain uml p : Property{upperValue = :ValueSpecification{value = m}};
```

```
    enforce domain owl owlp : Property{propertyRestriction = rest:MaxCardinalityRestriction{
        OWLmaxCardinality = :TypedLiteral{lexicalForm = m, datatypeURI = :URIRef{
            uri = :UniformResourceIdentifier{name = "xsd:integer"}}},
        RDFSdomain = :OWLClass{RDFSSubClassOf = rest}};
    when {
        not m = '*';  // Excludes infinitity, which is no restriction
        PropertyToProperty(p, owlp);
    }
    where {
        BlankNodeID(rest);
    }
} // UpperMultToMaxCard

relation BlankNodeID
// Make sure Blank node has a nodeID
{
    n : string;
    checkonly domain owl rest : OWLRestriction{};
    enforce domain owl rest : OWLRestriction{nodeID = n};
    where {
        if rest.nodeID->isEmpty n = genAnonNodeID();
// Generates a unique blank node identifier
    }
} // BlankNodeID

top relation LowerMultToMinCard
// Lower multiplicity
{
    m, n : string;
    checkonly domain uml p : Property{lowerValue = :ValueSpecification{value = m}};
    enforce domain owl owlp : Property{propertyRestriction = rest:MinCardinalityRestriction{
        OWLminCardinality = :TypedLiteral{lexicalForm = m, datatypeURI = :URIRef{
            uri = :UniformResourceIdentifier{name = "xsd:integer"}}},
        RDFSdomain = :OWLClass{RDFSSubClassOf = rest}};
    when {
        not m = 0;  // Excludes zero, which is no restriction
        PropertyToProperty(p, owlp);
    }
    where {
        BlankNodeID(rest);
    }
} // LowerMultToMinCard
```

D.3.8 Association Generalization

Several kinds of generalizations of properties and associations in UML are mapped to subproperty and subclass relationships in OWL:

- subsetted properties to subPropertyOf
- properties at member ends of a generalized association to subPropertyOf
- generalized n-ary associations or association classes to subClassOf

```
top relation SubsetsPropertyToSubproperty
```

```
// Map a subsetted property to subPropertyOf
{
   checkonly domain uml prop:Property{subsettedProperty = superprop : Property};
   enforce domain owl oprop : Property{RDFSsubPropertyOf = osuperprop : OWLProperty};
   when {
          PropertyToProperty(prop, oprop);
          PropertyToProperty(superprop, osuperprop);
   }
} // SubsetsPropertyToSubproperty

top relation AssocGeneralToSubProp
// Maps each member end of a generalized association to a subproperty
// Steps through the member ends by successive instantiations of the set comprehension pattern
{
   p1, p2 : OWL::...::OWLBase::Property;
   checkonly domain uml assn : Association{general = superassn : Association{
                 memberEnd = supSeq : Sequence(Property){usuper | true}},
          memberEnd = subSeq : Sequence(Property){uprop | true}}
          {supSeq->indexOf(usuper) = subSeq->indexOf(uprop)}; // steps through both sequences
   in tandem
   enforce domain owl oprop:Property{RDFSsubPropertyOf = osuper : Property};
   when {
          (AssociationToPropertyPair(assn, p1);  AssociationToPropertyPair(superassn, p2))OR
          (SymAssociationToSymProperty(assn, p1); SymAssociationToSymProperty(superassn,
   p2))OR
          (NaryAssociationToOWLProps(assn, p1); NaryAssociationToOWLProps(superassn, p2))OR
          (AssociationClassToOWLProps(assn, p1); AssociationClassToOWLProps(superassn, p2));
// Makes sure associations are both mapped
          EndPropertyToProperty(usuper, osuper); // Extracts mapping of end properties
          EndPropertyToProperty(uprop, oprop); // Corresponding ends in super and sub
   }
} // AssocGeneralToSubProp

top relation GeneralizesNaryToSubclass
// Creates subclass relationship for mapped n-ary associations
{
   checkonly domain uml assn : Association{general = superassn : Association};
   enforce domain owl oclass : OWLClass{RDFSsubclassOf = osuper : OWLClass};
   when {
          NaryAssociationToOWLClass(assn, oclass);
          NaryAssociationToOWLClass(superassn, osuper);
   }
} // GeneralizesNaryToSubclass

top relation GeneralizesAssocClassToSubclass
// Creates subclass relationship for mapped association class generalization
{
   checkonly domain uml assn : AssociationClass{general = superassn : AssociationClass};
   enforce domain owl oclass : OWLClass{RDFSsubclassOf = osuper : OWLClass};
   when {
          AssociationClassToOWLClass(assn, oclass);
          AssociationClassToOWLClass(superassn, osuper);
   }
} // GeneralizesAssocClassToSubclass
```

D.3.9 Enumeration

An enumeration in UML is typically a designated collection of literals, and in that case corresponds to an enumerated datatype in OWL.

```
top relation EnumerationToEnumeratedDatatype
//Created an enumerated datatype from an enumeration
(
   checkonly domain uml enum:Enumeration{ownedLiteral = ul : EnumerationLiteral};
   enforce domain owl edt:OWLDataRange{OWLoneOf = ol:RDFSLiteral};
   where {
         UMLLiteralToOWLLiteral(ul, ol); // not supplied
         UClassToOClass(enum, edt);
   }
) // EnumerationToEnumeratedDatatype
```

D.3.10 Powertypes

If a generalization set is covering, the general classifier is the union of the specific classifiers. If a generalization set is disjoint, then the specific classifiers are pairwise disjoint. OWL does not support the equivalent for properties, so generalization sets involving Associations are not mapped.

```
top relation IsCoveringToUnion
// Covering generalization set of classes goes to union
{
   checkonly domain uml genset : GeneralizationSet{isCovering = true,
         powertype = super : Class,
         generalization = :Generalization{specific = speccl : Class}};
   enforce domain owl ucl: UnionClass{OWLunionOf = osc : OWLClass};
   when {
         UClassToOClass(super, ucl);
         UClassToOClass(speccl, osc);
   }
}// IsCoveringToUnion

top relation IsDisjointToDisjoint
// Disjoint generalization to disjoint subclasses. Will generate sufficient pairs to make the
   mappings all pairwise disjoint
{
   checkonly domain uml genset : GeneralizationSet{isDisjoint = true,
         powertype = super : Class,
         generalization = :Generalization{specific = scl1 : Class},
         generalization = :Generalization{specific = scl2 : Class}};
// Succeeds if there are more than one specific class
   enforce domain owl cl1 : OWLClass{OWLdisjointWith = cl2 : OWLClass};
   when {
         scl1.name < scl2.name;
         UClassToOClass(scl1, cl1);
         UClassToOClass(scl2, cl2);
   }
} // IsDisjointToDisjoint

} // transformation UMLToOWL
```

D.4 OWL to UML

D.4.1 Problematic Features of OWL

D.4.1.1 Mapping for Individuals

In the profile, Individual is represented as a singleton class. This works well in a profile, because a profile is a tool of the OWL ontology creator. The ontology creator would only model individuals in a profile if there were some special reason to. They could, and probably would, choose not to represent the vast bulk of individuals in this way.

The mappings on the other hand have to treat all individuals uniformly. It would be possible to map individuals to singleton classes, but then the mapping would have to deal with the RDFtype association. If an object is modeled as a singleton class, then the subclass relationship is equivalent to the instance relationship, so it would be necessary to map the RDFtype associations to subclass relationships in UML. This is probably not at all what one would generally want. In terms of Gruber's ontology quality measures, this is enormous encoding bias.

In these informative mappings, mapping Individual is not specified.

D.4.1.2 Mapping for Enumerated Classes

Enumerated classes are represented in OWL as owl:oneOf class restrictions, where the enumeration is either a data range (literals) or a set of individuals. Since individuals are not mapped, only the data range version of oneOf class restriction is mapped.

D.4.1.3 Mapping for complementOf and disjointWith

UML has constructions corresponding to the OWL complementOf and disjointWith constructions in the PowerSets package. These have been mapped in the UML to OWL mappings. However, in OWL these constructs are pairwise rather than applying to an entire generalization set. Mapping pairwise restrictions like this is complicated, leading to a difficult-to-read UML model. In these informative mappings, mappings for OWLcomplementOf and OWLdisjointWith are not specified.

D.4.1.4 Multiple Domains or Ranges for Properties

It is possible for a property to have multiple classes specified as either the domain or range of a property. In this case, OWL specifies that the domain or range is the intersection of all. The mappings specified here take this into account.

A problem is that multiple instances of domain or range specifications can come from a number of sources.

- Domain of the inverse of a property is range of the property, the range of the inverse is domain of a property
- Domain and range can be specified on superproperties of a property
- Domain and range can be derived from other equivalent properties
- Combinations of these

The mappings assume that the OWL model instance mapped includes the deductive closure of all domain and range specifications.

D.4.2 Transformation Header

```
transformation OntoUMLSource  (owl:OWLMetamodel, uml:UMLMetamodel)
{
```

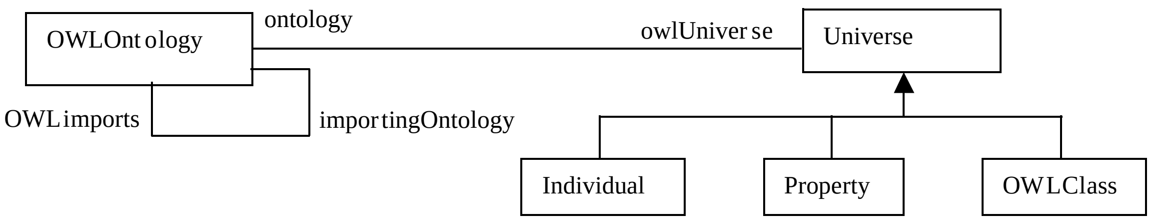

Figure D.10 - Package structure of OWL

```
key OWLOntology(uriRef);
key OWLUniverse (uriRef, ontology);
key OWLUniverse (nodeID, ontology); // anonymous classes
// All objects in an OWL model instance are instances of OWLUniverse. Figure D.10.
// Objects in UML have names relative to other constructs, ultimately to Package
key Package(name);
key Class(name, owningPackage);
key Association(name, owningPackage);
key UML::...::Kernel::Property(name, class);
key UML::...::Kernel::Property(name, association);
```

D.4.3 Packaging Construct: OWLOntology

D.4.3.1 Ontology to Package

```
top relation OntoToPackage //Map ontology to Package.
{
   checkonly domain owl ont:OWLOntology { };
   enforce domain uml pack:Package { };
   where {
          TopOntoToPackage(ont, pack);
          IntOntoToPackage(ont, pack);
   }
} // OntoToPackage

relation TopOntoToPackage  //Map top ontology to Package (Figure D.5 ).
{
   n, un : string;
   checkonly domain owl ont:OWLOntology {uriRef = :UniformResourceIdentifier{name = n}};
   enforce domain uml pack:Package {name = un};
   when {
          ont.importingOntology->isEmpty;
   }
   where {
          URIRefToName(n, un);
   }
} //TopOntoToPackage

relation IntOntoToPackage  //Map imported ontology to Package (Figure D.5).
{
   n, un : string;
   checkonly domain owl ont:OWLOntology {uriRef = :UniformResourceIdentifier{name = n},
```

```
                importingOntology =  onta : OWLOntology};
    enforce domain uml pack:Package {name = un,
                        _packageImport = :PackageImport{importingNamespace = packa:Package}};
    when {
            OntoToPackage(onta, packa);
    }
    where {
            URIRefToName(n, un);
    }
} //IntOntoToPackage
```

D.4.3.2 Ontology Properties to Comments

The Package construct in UML cannot be the source of properties. The only related structure is the facility to attach comments inherited from NamedElement. Other than the mapping between importingOntology and packageImport, ontology properties are therefore mapped to comments. Note that OWLversionInfo is not an Ontology property, but an annotation property.

```
top relation PriorVersionInfoToComment
{
    v : string;
    checkonly domain owl ont : OWLOntology{OWLpriorVersion = :RDFSLiteral{lexicalForm = v}};
    enforce domain uml pack : Package{ownedComment = :Comment{body = ("Prior Version " + v)}};
    when {
            OntoToPackage(ont, pack);
    }
} // PriorVersionInfoToComment

top relation IncompatibleWithToComment
{
    n : string;
    checkonly domain owl ont:OWLOntology {OWLincompatibleWith = :OWLOntology{
            uriRef = :UniformResourceIdentifier{name = n}}};
    enforce domain uml pack : Package{ownedComment = :Comment{body = ("Incompatible With " +
    n)}};
    when {
            OntoToPackage(ont, pack);
    }
} // IncompatibleWithToComment

top relation BackwardsCompatibleWithToComment
{
    n : string;
    checkonly domain owl ont:OWLOntology {OWLbackwardsCompatibleWith = :OWLOntology{
            uriRef = :UniformResourceIdentifier{name = n}}};
    enforce domain uml pack : Package{ownedComment = :Comment{body = ("Backwards Compatible
    With " + n)}};
    when {
            OntoToPackage(ont, pack);
    }
} // BackwardsCompatibleWithToComment
```

D.4.4 Classes

D.4.4.1 OWL Class to UML Class

Classes in OWL are identified by uri, except for restriction classes that are blank nodes. A class in UML is identified by name within package.

```
    top relation OClassToUClass
{
    checkonly domain owl cl:OWLClass {ontology = ont : OWLOntology};
    enforce domain uml ucl:Class {owningPackage = pack : Package};
    when {
            OntoToPackage(ont, pack)
    };
    where {
            UClassToClass(cl, ucl);
            AnonClassToClass(cl, ucl);
    }
} // OClassToUClass
```

D.4.4.2 Class Identified by URI

```
relation UClassToClass
// map an OWL class identified by uri to a topic.
{
    identifier, un : string;
    checkonly domain owl cl:OWLclass {uriRef = :UniformResourceIdentifier{name = identifier}};
// Note that an instance of an anonymous class fails to have a uri, so is excluded
    enforce domain uml ucl:Class {name = un};
    where {
            URIRefToName(identifier, un);
    }
} // UClassToClass
```

D.4.4.3 Anonymous Class to Class

An anonymous class in OWL is a blank node and is identified by a nodeID. This is unique within an ontology, if not persistent. This identifier will serve as a name for the corresponding UML class, where the name is persistent.

```
relation AnonClassToClass
// map an anonymous OWL class to a UML class.
{
    ID : string;
    checkonly domain owl aclass:OWLClass {nodeID = ID};
    enforce domain uml ucl : Class {name = ID};
    when{
            aclass.uriRef->isEmpty; // Some classes have also uri refs. These classes are not
    anonymous.
    }
} // AnonClassToClass
```

D.4.5 Hierarchy

D.4.5.1 Subclass, Equivalent Class

```
    top relation SubclassToGeneralization
// map the RDFSsubclassOf meta-association to a UML subclass/superclass relationship (Figure
    D.6).
{
    checkonly domain owl subcl:OWLClass {RDFSsubClassOf = supercl : OWLClass};
    enforce domain uml usubcl:Class {superClass = usuper : Class};
    when {
            OClassToUClass(subcl, usubcl);
            OClassToUClass(supercl, usuper);
} // SubclassToGeneralization

top relation EquivalentClassToMutualGeneralization
// map equivalent classes to a pair of UML subclass/superclass relationships (Figure D.6).
{
    checkonly domain owl class1:OWLClass {OWLEquivalentClass = class2 : OWLClass};
    enforce domain uml uclass1:Class {superClass = class2 : Class{superClass = class1}};
    when {
            OClassToUClass(class1, uclass1);
            OClassToUClass(class2, uclass2);
} // EquivalentClassToMutualGeneralization
```

D.4.5.2 Universal Superclass

OWL has a universal superclass called owl:Thing acting as a default domain and range for object properties. A comparable model library element for use in UML is available in Annex A, sub clause A.6, Table A.4 and is created for use in the QVT mapping below.

```
top relation UniversalSuperclass
{
    owlcl : OWLClass;
    checkonly domain owl ont : OWLOntology{};
    enforce domain uml thing : Class(name = "Thing", owningPackage = pack : Package,
            _class = ucl : Class}; // _class is opposite metaproperty to superClass
    when {
            OntoToPackage(ont, pack);
            owlcl = ont.owlUniverse; // will instantiate for each OWL object which is an OWL-
    Class
            ucl = pack.ownedMember; // will instantiate for each ownedMember which is a Class
            SubclassToGeneralization(owlcl, ucl); // forces wait until subclasses structure has
    been generated
            ucl.superClass->isEmpty; // selects only those classes without superclasses
    }
} //UniversalSuperclass
```

D.4.6 Constructed Classes

OWL allows classes to be constructed by union, intersection, and difference. OWL also allows classes to be declared disjoint. Union and intersection can be mapped to subclass relationships, while disjoint and difference are not mapped.

```
top relation IntersectionToUML
```

```
// OWL intersection to subclass relationships
// Wil generate an instance of superClass for each instance of OWLintersectionOf
{
   checkonly domain owl interclass : IntersectionClass{OWLintersectionOf = oclass : OWLClass,
          ontology = ont : OWLOntology};
   enforce domain uml usub:Class{superClass = uclass : Class, owningPackage = pack : Package};
   when {
          OntoToPackage(ont, pack);
          OClassToUClass(interclass, usub);
          OClassToUClass(oclass, uclass);
   }
} // IntersectionToUML

top relation UnionToUML
// OWL union to subclass relationships
// Wil generate an instance of _Class for each instance of OWLunionOf
{
   checkonly domain owl unclass : UnionClass{OWLunionOf = oclass : OWLClass,
          ontology = ont : OWLOntology};
   enforce domain uml usuper:Class{_Class = uclass : Class, owningPackage = pack : Package};
   when {
          OntoToPackage(ont, pack);
          OClassToUClass(unclass, usuper);
          OClassToUClass(oclass, uclass);
   }
} // UnionToUML
```

D.4.7 Data Range

A data range in OWL is either a literal type or an enumeration of literals. The mapping of literal types is application specific, but the enumeration corresponds to the enumeration in UML.

```
top relation EnumerationToEnumeration
{
   checkonly domain owl edt:OWLDataRange{OWLoneOf = ol:RDFSLiteral{lexicalForm = v}};
   enforce domain uml enum:Enumeration{ownedLiteral = ul : EnumerationLiteral};
   where {
          OWLLiteralToUMLLiteral(ol, ul); // Not supplied
          OClassToUClass(edt, enum);
   }
} // EnumerationToEnumeration
```

D.4.8 Range Restriction Restriction Classes

The restriction classes allValuesFrom, someValuesFrom, and HasValue define subclasses of the domain on which the specified property has constrained values. The mapping provided here uses an anonymous class, declared as a subclass of the domain of the property (if any), with the restriction indicated in an attached comment. The mapping for hasValue only includes the case where the value is a literal.

```
top relation AllValuesFromToClass
{
   cn, pn : string;
   up : UML:...:Kernel:Property;
   checkonly domain owl avr : AllValuesFromRestriction{OWLallValuesFrom = oc : OWLClass,
```

```
        OWLonProperty = op : RDFProperty};
    enforce domain uml rcl : Class{comment = ("AllValuesFrom " + cn + " on " pn)};
    when {
        OClassToUClass(avr, rcl);
        OWLPropToUMLProp(op, up);
    }
    where {
        cn = rcl.name;
        pn = up.name;
        SubclassOfPropDomain(op, rcl);
    }
} // AllValuesFromToClass

relation SubclassOfPropDomain
// Makes UML mapping of restriction class subclas of mapping of property domain, if any
{
    checkonly domain owl op:Property{RDFSdomain = oc : OWLClass};
    enforce domain uml uc:Class{superClass = uc};
    when {
        OClassToUclass(oc, uc);
    }
} // SubclassOfPropDomain

top relation SomeValuesFromToClass
{
    cn, pn : string;
    up : UML::...::Kernel::Property;
    checkonly domain owl svr : SomeValuesFromRestriction{OWLsomeValuesFrom = oc : OWLClass,
        OWLonProperty = op : RDFProperty};
    enforce domain uml rcl : Class{comment = ("SomeValuesFrom " + cn + " on " pn)};
    when {
        OClassToUClass(svr, rcl);
        OWLPropToUMLProp(op, up);
    }
    where {
        cn = rcl.name;
        pn = up.name;
        SubclassOfPropDomain(op, rcl);
    }
} // SomeValuesFromToClass

top relation HasValueToClass
// Only applies to case where value is a literal
{
    v, pn : string;
    up : UML:...:Kernel:Property;
    checkonly domain owl hvr : HasValueRestriction{OWLhasValue = olit : RDFSLiteral{lexicalForm
    = v},
        OWLonProperty = op : RDFProperty};
    enforce domain uml rcl : Class{comment = ("HasValue " + v + " on " pn)};
    when {
        OClassToUClass(hvr, rcl);
        OWLPropToUMLProp(op, up);
    }
    where {
```

```
        pn = up.name;
        SubclassOfPropDomain(op, rcl);
    }
} // HasValueToClass
```

D.4.9 Properties in OWL

Properties in OWL are similar in concept to properties and associations in UML, but quite different in detail The mappings follow as closely as possible the profiles given in 14.3.6.

An OWL property will be mapped to a UML property, which is an ownedAttribute of a Class in case:

- It is a datatype property.
- It is an object property with no inverse and is not inverse functional.

An OWL property will be mapped to a UML binary association in case:

- It is an object property with an inverse (including symmetric property). In this case the property and its inverse will be mapped to the two ends of the association.
- It is inverse functional. An inverse functional property generates a partition of its range. Even if it has no inverse, it must be mapped to an association because the corresponding multiplicity must apply to the end opposite the end corresponding to the inverse functional property.

Cardinality constraints will be mapped to multiplicities, this includes functional and inverse functional properties.

A property can have possibly several classes declared as its domain, and possibly several declared as its range. In either case the result is

- no classes declared - owl:Thing
- one class declared - the class
- more than one class declared - the intersection of the declared classes.

D.4.9.1 Property to Owned Attribute

```
top relation DTPropToAttribute
{
    identifier, un : string;
    checkonly domain owl dtp:OWLDataTypeProperty {uriRef = :UniformResourceIdentifier{name =
    identifier},
                        RDFSrange = :TypedLiteral {datatypeURI = dt},
                        ontology = ont:OWLOntology};
    enforce domain uml prop : Property{name =un, owningPackage = pack : Package, type = tp :
    PrimitiveType};
    when {
        OntoToPackage(ont, pack);
        LiteralToPrimitiveType(dt ,tp); // relates RDF literal types to UML primitive types
    }
    where {
        URIRefToName(identifier, un);
    }
} //DTPropToAttribute
```

```
top relation ObjPropToAttribute
{
   identifier, un : string;
   checkonly domain owl op:OWLObjectProperty {uriRef = :UniformResourceIdentifier{name =
   identifier},
                           ontology = ont:OWLOntology};
   enforce domain uml prop : Property{name =un, owningPackage = pack : Package};
   when {
         OntoToPackage(ont, pack);
         op.inverseProperty->isEmpty;  // no inverse
         op.OWLinverseOf->isEmpty;
         not op.oclIsTypeOf(SymmetricProperty); // not its own inverse
         not op.oclIsTypeOf(InverseFunctional); // not inverse functional
   }
   where {
         URIRefToName(identifier, un);
   }
} //ObjPropToAttribute

top relation AddAttributeClass
// Property and domain already mapped. Add Class.
{
   checkonly domain owl prop:Property{};
   enforce domain uml upr:Property{class = cl : Class};
   when {
         ObjPropToAttribute(prop, upr) OR DTPropToAttribute(prop, upr);
         PropDomain(prop, cl);
   }
} //AddAttributeClass

top relation AddAttributeType;
// property and range already mapped. Add type.
{
   checkonly domain owl prop:Property{};
   enforce domain uml upr:Property{type = cl};
   when {
         ObjPropToAttribute(prop, upr);
         PropRange(prop, cl);
   }
} // AddAttributeType
```

D.4.9.2 Property to Association

```
top relation PropertyPairToAssociation
// Property and its inverse go to an association whose name is the concatenation of the prop-
   erty names.
{
   assocID : string;
   pack1 : Package;
   checkonly domain owl prop:Property{ontology = ont:OWLOntology,
         OWLinverseOf = invp : Property{ontology = invont : OWLOntology}};
   enforce domain uml assn : Association(memberEnd = ps : Sequence(Property) {p1,  p2},
               name = assocID, owningPackage = pack : Package};
   when {
         OntoToPackage(ont, pack); // association will be in this package
```

```
        OntoToPackage(invont, pack1); // even though the inverse might be in another ontol-
    ogy
        invp.equivalentProperty->isEmpty;  // no equivalent properties
        invp.OWLEquivalentProperty->isEmpty;
    }
    where {
        PropertyToAProperty(prop, p1);
        PropertyToAProperty(invp, p2);
        assocID = p1.name + p2.name; // Association name is concatenation of property names
    }
} // PropertyPairToAssociation

top relation SymmetricPropToAssociation
// Symmetric property goes to an association both of whose member ends are the same property
{
    assocID : string;
    checkonly domain owl prop:SymmetricProperty{ontology = ont:OWLOntology};
    enforce domain uml assn : Association{memberEnd = ps : Sequence(Property) {p1,  p1},
        name = assocID, owningPackage = pack : Package};
    when {
        OntoToPackage(ont, pack); // association will be in this package
    }
    where {
        PropertyToAProperty(prop, p1);
        assocID = p1.name;
    }
} // SymmetricPropToAssociation

top relation InverseFunctToAssociation
// Inverse functional property with no inverse go to an association
{
    checkonly domain owl prop:InverseFunctionalProperty{ontology = ont:OWLOntology};
    enforce domain uml assoc:Association{memberEnd = ps : Sequence(Property){p1,  p2},
        name = assocID, owningPackage = pack : Package};
    when {
        prop.OWLinverseOf->isEmpty; // The inverse functional property has no inverse
    declared
        prop.inverseProprety->IsEmpty;
        OntoToPackage(ont, pack); // association will be in this package
    }
    where {
        PropertyToAProperty(prop, p1);
        PropertyToOppProperty(prop, p2);
        assocID = p1.name;
    }
} //InverseFunctToAssociation

relation PropertyToAProperty
// OWL property to UML property in context of an association end
{
    identifier, un : string;
    checkonly domain owl prop:Property{uriRef = :UniformResourceIdentifier{name = identi-
    fier}};
    enforce domain uml uprop : Property{name = un};
    where {
```

```
            URIRefToName(identifier, un);
    }
} // PropertyToAProperty

relation PropertyToOppProperty
// OWL property to opposite UML property in context of an association end (for inverse func-
    tional properties)
// This property has no corresponding OWL property, so can be fully specified.
{
    identifier, un, onopp : string;
    checkonly domain owl prop:Property{uriRef = :UniformResourceIdentifier{name = identi-
    fier}};
    enforce domain uml uprop : Property{name = unopp, lowerValue = '0', upperValue = '1'};
    where {
            URIRefToName(identifier, un);
            unopp = "opposite_" + un;
    }
} // PropertyToAProperty

top relation AddTypeAssocEnd
// association already mapped. Add types to properties at association ends.
// The type of the association end property is the class corresponding to the range of the cor-
    responding OWL property
{
    checkonly domain owl prop:Property{};
    enforce domain uml Assn:Association(memberEnd = upr : Property{type = ran : Class}};
    when {
            PropertyToAProperty(prop, upr);
            PropRange(prop, ran);
    }
} // AddTypeAssocEnd

relation OWLPropToUMLProp
// Returns the UML property corresponding to an OWL Property. The UML property has already
    been created.
{
    checkonly domain owl prop:Property{};
    enforce domain uml uprop : Property{};
    when {
            DTPropToAttribute(prop, uprop) OR
            ObjPropToAttribute(prop, uprop) OR
            PropertyToAProperty(prop, uprop);
    }
} // OWLPropToUMLProp
```

D.4.10 Domains, Ranges and Property Types

D.4.10.1 Domains

```
top relation PropDomain
{
    checkonly domain owl prop:Property {};
    enforce domain uml cl : Class{};
    where {
            DefaultDomain(prop, cl);
```

```
        SingleDomain(prop, cl);
        MultDomain(prop, cl);
    }
} // PropDomain

relation DefaultDomain
// Create default domain for property
{
    checkonly domain owl prop:Property{ontology = ont : OWLOntology};
    enforce domain uml usuper : Class{};
    when {
        prop.RDFSdomain->isEmpty; // no domains declared, so default
        UniversalSuperclass(ont, usuper); // usuper has already been created for this ontol-
    ogy
    }
} // DefaultDomain

relation SingleDomain
// Find single domain for property
{
    checkonly domain owl prop:Property{RDFSdomain = dom: OWLClass, ontology = ont : OWLOntol-
    ogy};
    enforce domain uml domcl : Class{};
    when {
        prop.RDFSdomain->size() = 1; // only one domain declared
        OClassToUClass(dom, domcl);
    }
} // SingleDomain

relation MultDomain
// Find intersection of multiple domains for property as subclass called "DomainIntersection_"
    + name of property
// Assumed that the collection of domains of a property is the deductive closure of all sources
    of domains, in particular a range of an inverseOf property
{
    din : string;
    obj : NamedElement;
    checkonly domain owl prop:Property{RDFSdomain = dom: OWLClass, ontology = ont : OWLOntol-
    ogy};
    enforce domain uml domintcl : Class{name = rin, superClass = domcl: Class,
        owningPackage = pack :Package};
    when {
        prop.RDFSdomain->size() > 1; // if more than one domain declared
        OClassToUClass(dom, domcl); // will be instantiated once for each class
        OWLPropToUMLObj(prop, obj); // Need the property to be created so can get its name
        OntoToPackage(ont, pack);
    }
    where {
        din = "DomainIntersection_" + obj.name;
    }
} // MultDomain
```

D.4.10.2 Ranges

```
top relation PropRange
```

```
{
    checkonly domain owl prop:Property {};
    enforce domain uml cl : Class{};
    where {
            DefaultRange(prop, cl);
            SingleRange(prop, cl);
            MultRange(prop, cl);
    }
} // PropRange

relation DefaultRange
// Create default range for property
{
    checkonly domain owl prop:Property{ontology = ont : OWLOntology};
    enforce domain uml usuper : Class{};
    when {
            prop.RDFSrange->isEmpty; // no ranges declared, so default
            UniversalSuperclass(ont, usuper); // usuper has already been created for this ontol-
    ogy
    }
} // DefaultRange

relation SingleRange
// Find single range for property
{
    checkonly domain owl prop:Property{RDFSrange = ran: OWLClass, ontology = ont : OWLOntol-
    ogy};
    enforce domain uml rancl : Class{};
    when {
            prop.RDFSrange->size() = 1; // only one range declared
            OClassToUClass(ran, rancl);
    }
} // SingleRange

relation MultRange
// Find intersection of multiple ranges for property as subclass called “RangeIntersection_” +
    name of property
// Assumed that the collection of ranges of a property is the deductive closure of all sources
    of ranges, in particular a domain of an inverseOf property
{
    rin : string;
    obj : NamedElement;
    checkonly domain owl prop:Property{RDFSrange = ran: OWLClass, ontology = ont : OWLOntol-
    ogy};
    enforce domain uml ranintcl : Class{name = rin, superClass = rancl: Class,
            owningPackage = pack : Package};
    when {
            prop.RDFSrange->size() > 1; // if more than one range declared
            OClassToUClass(ran, rancl); // will be instantiated once for each class
            OWLPropToUMLObj(prop, obj); // Need the property to be created so can get its name
            OntoToPackage(ont, pack);
    }
    where {
            rin = “RangeIntersection_” + obj.name;
    }
```

```
} // MultRange
```

D.4.11 Cardinalities and Multplicities

```
top relation CardinalityToMultiplicitiy
{
   checkonly domain owl oprop : Property{};
   enforce domain uml uprop : Property{};
   when {
          OWLPropToUMLProp(oprop, uprop); // Delays until properties have been created
   }
   where {
          if oprop.oclIsTypeOf(FunctionalProperty)  {
                 FunctionalToUMult(oprop, uprop);
                 FunctionalInverseFunToULMult(oprop, uprop);
   }      } else if oprop.oclIsTypeOf(InverseFunctionalProperty){
                 InvFunctionalToUMult(oprop, uprop);
          } else {
                 CardToULMult(oprop, uprop);
                 MaxCardToUMult(oprop, uprop);
                 MinCardToLMult(oprop, uprop);
          }
          AddUnlimitedMax(uprop);
          AddZeroMin(uprop);
   }
} // CardinalityToMultiplicitiy

relation CardToULMult
// max and min cardinality are the same
{
   c : string;
   checkonly domain owl oprop : Property{
          propertyRestriction = :CardinalityRestriction{OWLCardinality = :TypedLiteral{lexi-
   calForm = c}};
   enforce domain uml uprop : Property{upperValue = c, lowerValue = c};
} //MaxMinCardToULMult

relation MaxCardToUMult
// max cardinality
{
   u : string;
   checkonly domain owl oprop : Property{
          propertyRestriction = :MaxCardinalityRestriction{OWLmaxCardinality = :TypedLit-
   eral{lexicalForm = u}}};
   enforce domain uml uprop : Property{upperValue = u};
} //MaxCardToUMult

relation MinCardToLMult
// min cardinality
{
   l : string;
   checkonly domain owl oprop : Property{
          propertyRestriction = :MinCardinalityRestriction{OWLminCardinality = :TypedLit-
   eral{lexicalForm = l}}};
   enforce domain uml uprop : Property{lowerValue = l};
```

```
} //MinCardToLMult

relation FunctionalToUMult
// Functional property
{
   checkonly domain owl oprop : FunctionalProperty{};
   enforce domain uml uprop : Property{upperValue = "1"};
} //FunctionalToUMult

relation InvFunctionalToUMult
// If a inverse functional property has an inverse, the multiplicity goes on the inverse
{
   checkonly domain owl ifprop : InverseFunctionalProperty{};
   enforce domain uml opprop : Property{upperValue = '1'};
   when {
          ifprop.OWLinverseOf->exists(invprop : Property | PropertyToAProperty(invprop,
   opprop))
          or
          ifprop.inverseProperty->exists(invprop : Property | PropertyToAProperty(invprop,
   opprop))
   }
} // InvFunctionalToUMult

relation AddUnlimitedMax
// Add unlimited max cardinalities to UML properties with no max cardinality
{
   enforce domain uml prop:Property{upperValue = "*"};
   when {
          prop.upperValue->isEmpty;
   }
} // AddUnlimitedMax

relation AddZeroMin
// Add zero min cardinalities to UML properties with no min cardinality
{
   enforce domain uml prop:Property{lowerValue = "0"};
   when {
          prop.lowerValue->isEmpty;
   }
} // AddZeroMin
```

D.4.12 Subproperty, Equivalent Property

```
top relation SubpropertyToGeneralization
// a property generalizes by subsetting
{
   checkonly domain owl prop: Property{RDFSsubPropertyOf = superprop : Property};
   enforce domain uml uprop{subsettedProperty = superuprop : Property};
   when {
          OWLPropToUMLObj(prop, uprop);
          OWLPropToUMLObj(superprop, superuprop);
   }
} //SubpropertyToGeneralization

top relation EquivPropertyToGeneralizations
```

```
// a Equivalent properties by mutual subsetting
{
   checkonly domain owl prop: Property{OWLequivalentProperty = equivprop : Property};
   enforce domain uml uprop : Property{subsettedProperty = equivuprop : Property{
          subsettedProperty = uprop}};
   when {
          OWLPropToUMLObj(prop, uprop);
          OWLPropToUMLObj(equivprop, equivuprop);
   }
} //EquivPropertyToGeneralizations
```

D.4.13 Annotation Properties to Comments

OWL has an annotation property facility, including several built-in annotation properties, whose domain is OWLUniverse, while UML has only comments on NamedElement. Note that OWLversionInfo is an annotation property, not an ontology property. Note also that since the mapping doesn't map Individuals to UML, annotation properties on individuals are not mapped.

The built-in annotation properties are modeled as meta-associations, while the user-defined annotation properties are instances of OWLAnnotationProperty.

```
top relation VersionInfoToComment
{
   v : string;
   checkonly domain owl res :OWLUniverse{OWLversionInfo = :RDFSLiteral{lexicalForm = v}};
   enforce domain uml ne :NamedElement{ownedComment = :Comment{body = ("Version " + v)}};
   when {
          UniverseToNamedElement(res, ne);
   }
} // VersionInfoToComment

relation UniverseToNamedElement
{
   checkonly domain owl res :OWLUniverse{};
   enforce domain uml ne :NamedElement{};
   when {
          OntoToPackate(res, ne) OR
          OClassToUClass(res, ne) OR
   OWLPropToUMLProp(res, ne);
   }
} // UniverseToNamedElement

function StringFromURIRef(uriref : URIReference) : string
{
   uriref.fragmentIdentifier.name->isEmpty then uriref.name
          else  uriref.name + "#" + uriref.fragmentIdentifier.name;
} // StringFromURIRef

top relation RDFSCommentToComment
{
   com : string;
   checkonly domain owl res:OWLUniverse{RDFScomment = :RDFSLiteral{lexicalForm = com}};
   enforce domain uml ne:NamedElement(comment = com);
   when {
```

```
            UniverseToNamedElement(res, ne);
    }
} // RDFSCommentToComment

top relation RDFSLabelToComment
{
    com : string;
    checkonly domain owl res:OWLUniverse{RDFSlabel = :RDFSLiteral{lexicalForm = com}};
    enforce domain uml ne:NamedElement(comment = com);
    when {
            UniverseToNamedElement(res, ne);
    }
} // RDFSLabelToComment

top relation SeeAlsoToComment
{
    com : string;
    checkonly domain owl res:OWLUniverse{RDFSseeAlso = sr : RDFSResource};
    enforce domain uml ne:NamedElement(comment = com};
    when {
            UniverseToNamedElement(res, ne);
    }
    where {
            if sr->oclIsTypeOf(RDFSLiteral) then com = sr.lexicalForm
                    else   com = StringFromURIRef(sr.uriRef);
    }
} // SeeAlsoToComment

top relation IsDefinedByToComment
{
    com : string;
    checkonly domain owl res:OWLUniverse{RDFSisDefinedBy = sr : RDFSResource};
    enforce domain uml ne:NamedElement(comment = com};
    when {
            UniverseToNamedElement(res, ne);
    }
    where {
            if sr->oclIsTypeOf(RDFSLiteral) then com = sr.lexicalForm
                    else   com = StringFromURIRef(sr.uriRef);
    }
} // IsDefinedByToComment

top relation AnnotationPropertyToComment
// Map instances of annotation properties to comments. Excludes built-in annotation proper-
    ties.
{
    propuribase, propfrag, note : string;
    checkonly domain owl ap :OWLAnnotationProperty{uriRef = propuri:URIReference,
                    predicateStatement =  :RDFStatement{
                    RDFsubject = res : OWLUniverse, RDFobject = obj : RDFSResource};
    enforce domain uml ne:NamedElement{comment = (propname + " " + note)};
    when {
            UniverseToNamedElement(res, ne);
    }
    where {
```

```
            propname = StringFromURIRef(propuri);
            if obj->oclIsTypeOf(RDFSLiteral) then note = obj.lexicalForm
                   else   note = StringFromURIRef(obj.uriRef);
    }
} // AnnotationPropertyToComment

} // transformation OWLUMLSource
```

D.5 OWL but not UML

D.5.1 Predicate Definition Language

OWL permits a subclass to be declared using subClassOf or to be inferred from the definition of a class in terms of other classes. It also permits a class to be defined as the set of individuals that satisfy a restriction expression. These expressions can be a boolean combination of other classes (**intersectionOf**, **unionOf**, **complementOf**), or property value restriction on properties (requirement that a given property have a certain value – **hasValue**). The property **equivalentClass** applied to restriction expressions can be used to define classes based on property restrictions.

For example, the class definition[6]

```
    <owl:Class rdf:ID="TexasThings">
    <owl:equivalentClass>
           <owl:Restriction>
                  <owl:onProperty rdf:resource="#locatedIn" />
                  <owl:allValuesFrom rdf:resource="#TexasRegion" />
           </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

defines the class *TexasThings* as a subclass of the domain of the property *locatedIn*. These individuals are precisely those for which the range of *locatedIn* is in the class *TexasRegion*. Given that we know an individual to be an instance of *TexasThings*, we can infer that it has the property *locatedIn*, and all of the values of *locatedIn* associated with it are instances of *TexasRegion*. Conversely, if we have an individual that has the property *locatedIn* and all of the values of *locatedIn* associated with that individual are in *TexasRegion*, we can infer that the individual is an instance of *TexasThings*.

Because it is possible to infer from the properties of an individual that it is a member of a given class, we can think of the complex classes and property restrictions as a sort of predicate definition language.

UML provides but does not mandate the predicate definition language OCL. Note that a subsumption reasoner could be built for UML. But because UML is strongly typed, it could work in the way mandated for OWL only if there were a universal superclass provided in the model library, which is rarely provided in practice.

OCL and CL (Common Logic) are two predicate definition languages that are relevant to the ODM. Both are more expressive than the complex class and property restriction expressions of OWL Full. There are also other predicate definition languages of varying expressive powers that particular applications might wish to use.

The ODM does not mandate any particular predicate definition language, but will provide a place for a package enabling the predicate definition language of choice for an application. In particular, the ODM includes a metamodel for CL.

6. OWL Web Ontology Language Guide http://www/w3/org/TR/2003/PR-owl-guide-20031215/ section 3.4.1

D.5.2 Names

A common assumption in computing applications is that within a namespace the same name always refers to the same object, and that different names always refer to different objects (the **unique name assumption**). As a consequence, given a set of names, one can count the names and infer that the names refer to that number of objects.

Names in OWL do not by default satisfy the unique name assumption. The same name always refers to the same object, but a given object may be referred to by several different names. Therefore counting a set of names does not warrant the inference that the set refers to that number of objects. Names, however, are conceptually constants, not variables.

OWL provides features to discipline names. The unique name assumption can be declared to apply to a set of names (**allDifferent**). One name can be declared to refer to the same object as another (**sameAs**). One name can be declared to refer to something different from that referred to by any of a set of names (**differentFrom**).

Two classes can be stated to be equivalent (**equivalentClass**) and two properties can be stated to be equivalent (**equivalentProperty**). Equivalent classes have the same extents, equivalent properties link the same pairs.

Although a UML class may be defined to contain a definite collection of names, names at the M0 level are not prescribed. Applications modeled in UML are frequently implemented using systems like SQL that default the unique name assumption, but this is not mandated. UML places no constraints on names at the M0 level.

In particular, it is permitted for applications modeled in UML to be implemented at the M0 level using names that are existentially quantified variables. Note that the UML constraint language OCL uses variables. OWL does not support variables at all.

D.6 In UML But Not OWL

D.6.1 Behavioral and Related Features

UML allows the specification of behavioral features, which declare capabilities and dynamic aspects of the system. OCL can be used to restrict derived properties. Facilities of UML that can be used to describe application programs include:

- operations, which describe the parameters of methods;
- static operations, which have a shared implementation for all subclasses;
- interface classes, which specify an interface to a set of attributes and operations that could be implemented by one or more classes; and
- active classes, which are classes that have a separate thread of execution control for each instance.

ODM omits these features of UML.

D.6.2 Complex Objects

UML supports various flavors of the part-of relationship between classes. In general, a class (of parts) can have a part-of relationship with more than one class (of wholes). One flavor (**composition**) specifies that every instance of a given class (of parts) can be a part of at most one whole. Another (**aggregation**) specifies that instances of parts can be shared among instances of wholes.

Composite structures defined in classes specify runtime instances of classes collaborating according to **connectors**. They are used to show the valid combination of instances of internal classes and how these instances are combined to form the structure of the containing class. These diagrams extend the capabilities of class diagrams, which do not specify how internal parts are organized within a containing class and have no direct means of specifying how interfaces of internal parts interact with its environment.

Ports model how internal instances are to be organized. Ports define an interaction point between a class and its environment, or a class and its contents. They allow you to group the required and provided interfaces into logical interactions that a component has with the outside world. **Collaboration** provides constructs for modeling roles played by connectors.

Although not strictly part of the complex object feature set, the feature **template** (parameterized class) is most useful where the parameterized class is complex. One could for example define a multimedia object class for movies, and use it as a template for a collection of classes of genres of movie, or a complex object giving the results of the instrumentation on a fusion reactor that would be a template for classes containing the results of experiments with different objectives.

Although it is recognized that there is a need for facilities to model mereotopological relationships in ontologies, and UML provides a capability in this space, there does not seem to be sufficient agreement on the scope and semantics of existing models for inclusion of specific mereotopological modeling features into the ODM at this stage.

D.6.3 Access Control

UML permits a property to be designated **read-only**. It also allows classes to have **public** and **private** elements. ODM omits access control features.

D.6.4 Profiles

UML has a facility called **Profile**, whereby a specialist developer can make lightweight extensions to the modeling language by defining **stereotypes**, which define subclasses of metaclasses. This enables the developer to either articulate the metaclass into a number of kinds or to rename the metaclass.

OWL DL does not have a facility like this. One can achieve the same effect in OWL Full by defining subclasses of owl:Class or rdf:Property, since OWL is its own metalanguage.

Profiling makes it possible to reuse UML graphical rendering conventions, and take advantage of UML graphical editors and other tools. Stereotypes are used to extend the functionality of basic symbols, and thus reduce the number of symbols users are required to remember. While OWL does not have a standard graphical representation, the ODM profiles for RDF and OWL provide one. Because OWL DL does not support an equivalent of stereotypes, and because the functional equivalent of stereotypes in OWL Full is a user capability rather than a metamodeler capability as in UML, the mappings from UML to OWL and from OWL to UML disregard this feature.

Annex E
Mappings - Informative, Not Normative

In developing the mappings for the various ODM languages, the team concluded that the mappings we specify cannot in practice be normative.

In our discussion in 10.2.3, for example we see that there are two different ways to map n-ary associations from UML to OWL, depending on whether we take OWL Full or OWL DL as target. In 10.2.2, we note that OWL has a mandatory universal superclass (`owl:Thing`) which can map to a universal superclass in UML, but this is contrary to normal practice in UML modeling. A particular project might analyze the uses of universal properties in the OWL source model and choose to declare a number of more general but not universal superclasses in the UML target.

In the W3C Semantic Web Best Practices and Deployment task force's report on Topic Map mappings [RDF/TM], the point is made several times that there are different ways to map particular structures, and that each way has its advantages and disadvantages. In any particular project, design decisions will be taken in favor of advantages and against disadvantages so different projects will map in different ways.

There are several kinds of problems. One we can call structure conflation, where two constructs in one system map to a single construct in the other. In this case, a general-purpose mapping doesn't round trip. UML binary associations and class-valued attributes map to OWL properties, for example. In topic maps, three different kinds of identifiers map to one kind in OWL.

But there is nothing to stop a particular project from specifying naming conventions so there is a record in the target of what construct the source was, and from maintaining that convention in subsequent development.

A second kind of problem we will call structure loss. Here a complex construct is mapped to a collection of simpler constructs. There is insufficient information in the target metamodel for a general mapping to map collections of simple constructs to complex constructs in the source metamodel. Examples here are UML N-ary associations and association classes, which get mapped to a class and a collection of properties. In Topic Maps, the Association construct is typed itself and has N typed roles. The association maps to a class and the typed roles to properties. It is in general impossible to reliably map the reverse.

But again, there is nothing to stop a particular project from using naming conventions or annotations to retain a memory of the structure, and maintaining those conventions in subsequent maintenance so as to be able to reverse map.

Alternatively, a TM project could decide to limit itself to binary associations, making possible mapping associations directly to properties in that particular case.

The third kind of problem we will call trapdoor mappings, where a kind of construct in the source is mapped to a very specific arrangement of a general structure in the target. The analogy is with cryptography, where the encryption function takes any plaintext into an encrypted text, but almost no encrypted texts map back to plain text.

In topic maps, this occurs with the mapping of scope and variant names to specific properties in OWL identified with TM URIs. OWL properties map to TM associations with specific roles named with OWL URIs. Unless the source for a reverse mapping happened to maintain these conventions, it would be impossible to reverse in a sensible way.

A fourth kind of problem stems from what we will call feature lack, that the target metamodel lacks a feature present in the source. In this case there is no apparent general way to map the feature from the source. But in a particular project the feature may for example be used in a particular way leading to a mapping to target features particularized by naming conventions. OWL restriction classes relative to UML or Topic Map are of this kind.

The fifth kind of problem is what we will call incompatible structural principles. The different metamodels are organized very differently. UML is organized around classes, with instances as subordinate objects. OWL has both classes and individuals typed only by a universal superclass. In Topic Maps, a Topic instance can be either typed or not. But a particular project might use a particular discipline in its use of these structures leading to mappings not otherwise feasible.

In practice, the mappings provided in the ODM can be useful, though. First, they show feasibility of one set of design choices for the mappings, providing a baseline from which a particular project can vary. Second, they bring clearly to the fore the detailed relationships among the metamodels. These relationships can help those who understand one of the target languages to come to an understanding of the others. UML is similar to ER, but both are very different from RDFS/OWL, and all are quite different from TM. CL has far greater functionality than any of the others.

So although normative mappings are not feasible, we argue that the mappings presented have strong informative value.

Annex F
The Relationship of the Business Nomenclature Metamodel to the ODM

The Business Nomenclature (BN) metamodel from the CWM standard is shown in Figure F.1.

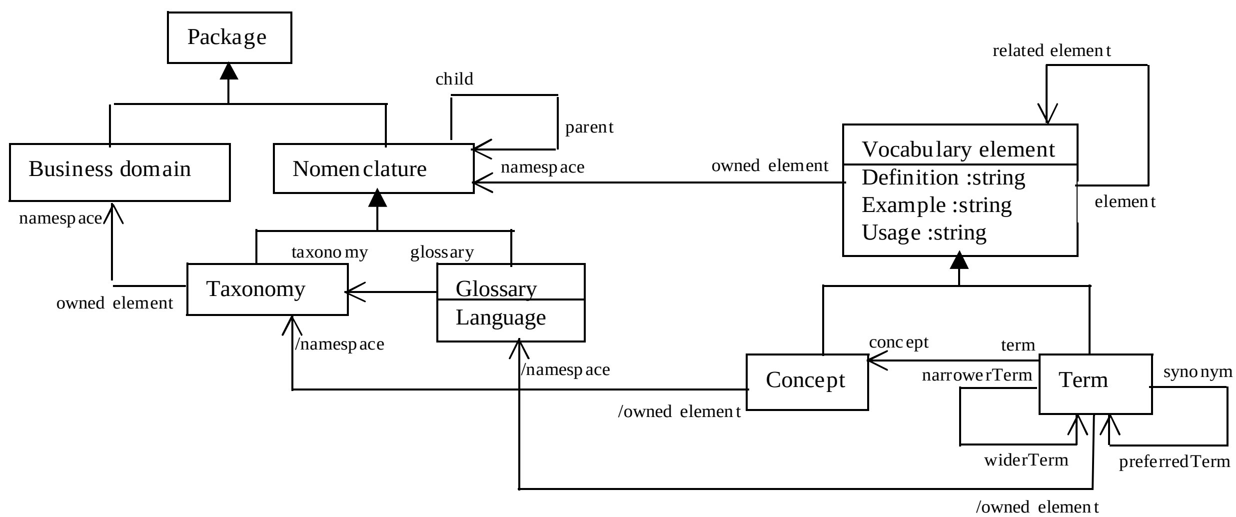

Figure F.1 - The Business Nomenclature Metamodel

The metamodel has been redrawn and slightly modified to simplify the picture without losing anything essential:

- Derived attributes have been omitted
- The aggregation notation on the associations whose source is *Nomenclature* have been removed
- The cardinality constraints have been omitted
- The *Model Element* metaclass has been omitted
- Two associations at *Concept* and *Term* derived from *element*/*relatedelement* have been omitted.

In this note we show that BN can be comfortably represented as an M1 model in place of its native M2 model, and in the three key ODM metamodels: UML, OWL and Topic Maps.

We can exhibit BN as an UML model simply by reinterpreting the diagram.

- The terms represented in rectangles (the upper rectangle where two rectangles are conjoined) are natively MOF classes. To see BN as a UML model, we interpret the rectangles as UML Classes.
- The terms represented as attached to ends of unadorned lines are names of pairs of MOF associations and their opposites. In the UML interpretation, these are interpreted as UML associations and their opposites.
- Terms represented in the lower of two attached rectangles are names of MOF attributes. In the UML interpretation, these are interpreted as UML attributes.

The systems of lines terminating in an arrow represent the MOF subclass relationship, with the MOF classes at the unadorned ends being MOF subclasses of the MOF class at the end with the arrow. In the UML interpretation, the system of lines represents the UML subclass relationship.

Similarly, MOF cardinality constraints (not shown) would be interpreted as UML cardinality constraints.

We now want to see BN as an OWL model. Note that OWL does not have a native graphic notation. However, the mapping from UML to OWL in Chapter 16 allows us to interpret the MOF diagram as a visualization of an OWL ontology:

- UML classes are OWL *classes*.
- UML associations are pairs of OWL *object properties* with their *inverses*.
- UML attributes are OWL *datatype properties*.
- UML subclasses are M1 instances of the *RDFsubclassOf* MOF association.
- UML cardinality constraints are OWL cardinality restrictions.

So with these interpretations, Figure F.1 is a visualization of an OWL model.

In a similar way, by following the mapping in Clause 17, Mapping Topic Maps to OWL, the BN metamodel can be interpreted as a Topic Map:

- OWL classes are *Topics*.
- OWL object properties are TM *Topics which are types of associations*. The names of the ends are interpreted as TM *Topics which are types of association roles*.
- OWL datatype properties are TM *Topics which are types of occurrences*.
- OWL subclasses are M1 instances of TM *Association* participating in the TM *type* meta-association with the M1 instance 'supertype-subtype' of *Topic*, with M1 instances of *Topic* playing roles of supertype and subtype.
- OWL cardinality constraints are not mapped, since TM does not support a comparable feature.

With these interpretations, Figure F.1 is a visualization of a Topic Map.

An instance of BN in the various systems is:

- An M1 instance of the MOF metamodel in Figure F.1 would be something like the INSPEC thesaurus. All the metaclasses would be populated with concrete strings representing the vocabulary elements and higher-level constructs organizing this particular thesaurus.
- An M0 instance of the UML model would be exactly the same thing using the UML instance model rather than the MOF instance model.
- The INSPEC thesaurus is an OWL ontology M1 model instance of BN with an M0 population of Individual and Statement.
- The INSPEC thesaurus is a Topic Map is a collection of instances of *Topic* linked by instances of *Association*. (The Topic Maps metamodel is a mixture of M2 and M1 level constructs, so some of the instances of Topic are M1 and others are M0.)

Finally, the fact that an M1 instance of BN is identical to an M0 instance of the equivalent UML model suggests that BN is more appropriately modeled at the UML M1 level than the MOF M2 level. M1 instances of the ODM metamodels can be left unpopulated if there are no M1 instances of their instances models. BN does not have its own instance model, instead relying on the MOF instance model, hence the instances models of the other metamodels.

That the ODM casts light on the strategy for modeling other systems suggests a further use for ODM components. In particular, the component Common Logic has a sound model theory, so that CL can be used to ground other models like SBVR (Semantics of Business Vocabulary and Business Rules) or PRR (Production Rule Representation).

Annex G
MOF QVT: A Brief Tutorial

G.1 General

Mappings are expressed in the Final Adopted Specification version of the MOF Query / Views / Transformations [MOF QVT]. Since this is very new, we present here some key points on QVT Relations which may assist the reader. More detailed points are included as comments where they appear in the individual mapping clauses.

G.2 Sketch of QVT

A transformation is represented as a Relation statement, which has four parts. Two of the parts, the checkonly and when clauses, describe patterns of objects in the models being transformed. Elements of these patterns are named by variables. If a collection of objects exists in the models which satisfy the pattern, the elements in positions named by variables instantiate the variables. If more than one such collection exists, the variables are instantiated multiple times, once for each collection.

The other two parts, the enforce and where clauses, carry out the transformation. The enforce clause describes a pattern in the same way as the checkonly clause, but where the pattern described does not exist in the model, sufficient objects are added to the model to satisfy the pattern. The where clause indicates further transformations which must be executed by calling other relations. A transformation is carried out for each set of instantiations of the variables.

The when clause includes calls to relations which are treated as predicates, returning true if the predicates in the checkonly and when clauses are satisfied, and if the enforce clauses in the relations called in the where clause find that the patterns specified to be constructed already exist. No work is done via a when clause. The when clause can also contain OCL expressions, which can also instantiate variables. In the when clause, the various predicates are connected by either semicolon, interpreted as conjunction, or “or,” interpreted as disjunction. In evaluating a disjunction, each variable instantiation making the disjunction true is successively generated.

OCL functions can be used.

There are two kinds of variables in QVT: a native QVT variable and the variables appearing in iterators in OCL statements. Native QVT variables have scope the relation they occur in. All variables must be declared as to type, but the declaration can occur in the checkonly or enforce clauses as well as by specific declaration. A variable cannot be declared in a relation call.

OCL iterator variables have scope as in OCL, namely limited to the subexpression they appear in. But a QVT relation call containing OCL variables can appear in the body of an iterator.

Relations are executed in two different ways. A top relation is executed by the system. Relations not top are executed by being called in a where clause. Both kinds of relations can be called in a when clause. The guard patterns in the checkonly and when clauses control the sequence of evaluation of top relations.

When a relation is called, its parameters are passed as for a normal procedure call. The variables declared in domains in checkonly and where clauses are treated as formal parameters, associated sequentially with variables in the calling sequence.

Once an object is created, its type is fixed. An object will match a pattern specifying a supertype, but not one specifying a subtype. Therefore an object must be created with the most specific type applying to it. Properties of the object specified in supertypes can be added by relations in the where clause, so the common MOF model architecture of abstract classes is supported. Note that subclass relationships must be directly declared in the MOF model. QVT does not recognize overlapping subclasses.

In case a model element (A) depends on another model element (B) (has an association with lower multiplicity greater than 0 at the member end), the mapping of A can be delayed until after the mapping of B by placing the relation mapping B in the when clause of the relation mapping A.

In case a model element (B) is subordinate to another model element A, the relation mapping B is placed in the where clause of the relation mapping A, so B is mapped after A. A subordinate model element B is a part of A which makes little sense apart from A. B often would have an identifier which includes the identifier of A. But the decision to consider one element as subordinate another is a design choice. There is no absolute criterion.

G.3 Repository Issues

Mappings in QVT involve finding instances of patterns. It therefore becomes important to know what populations of the various ODM model instances must be accessible in the repository serving the QVT pattern matching engine. These same considerations apply to any reasoning procedures that a repository may be called upon to perform.

Four of the five metamodels have a packaging construct. In UML, the packaging construct is called Package, in OWL OWLOntology, in Topic Maps the Topic Map and in Common Logic the Module. In these metamodels, structural declarations and constraints are bounded in scope by the packaging constructs containing them, and by the packaging construct contents that may be imported into them.

RDFS on the other hand does not have a bounded packaging construct. RDFS elements are instances of Statement, which are grouped into Graphs, which may optionally be named. But an instance of Graph does not necessarily have any semantic relevance. In particular, it does not necessarily bound the scope of a structural declaration.

There is a relationship between the Graph metaclass of RDFS and the OWLOntology metaclass of OWL. All OWL declarations are equivalent to RDF statements, so OWL declarations involve RDF Graphs. However, a given instance of RDF Graph can contain (parts of) possibly several Ontologies, and a given Ontology can have parts in possibly several RDF Graphs.

So when performing a mapping using QVT or performing any reasoning task, the repository must make accessible the relevant packaging structure contents and the contents of imported structures. The computation of patterns and other reasoning computations are limited in scope to the relevant packaging structure contents. However, in the case of RDFS the application is responsible for making the relevant RDF Graphs accessible to whatever reasoning mechanisms the repository may be called upon to perform. The mechanisms for specifying the relevant graphs is outside the scope of the ODM.

There is also a relationship between the population of a model instance accessible to a repository procedure and the MOF concept of navigability. A process in a repository can link from one object to another via any property, whether designated navigable or not. Designating a property navigable is an instruction to repository designers to make it easy to find links using that property, by for example providing indexes. Although using a non-navigable property is legal, is discouraged on the grounds that it not necessarily efficiently supported.

For example, in UML there is no specified property linking an instance of Package to a package that may import it. This reflects the fact that a package does not necessarily know which other package might import it. However, if the repository is known to contain the contents of all relevant packages, it is possible to find all packages importing a given package, relative to the content of the repository.