Ontology Definition MetaModel

Initial Submission

Submitted By:

DSTC

18 August 2003

Copyright © 2003 DSTC Pty Ltd.

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

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version.

The copyright holders listed above have agreed that no person shall be deemed to have infringed the copyright, in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

NOTICE: The information contained in this document is subject to change with notice.

The material in this document details a submission to the Object Management Group for evaluation in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification by the submitter.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The Object Management Group and the companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

All Rights Reserved. No part of the work covered by copyright hereon may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner. All copies of this document must include the copyright and other information contained on this page.

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

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

OMG® is a registered trademark of the Object Management Group, Inc.

Table of Contents

Chapter 0 Overview 7

- 0.1 Primary Contact for the Ontology Definition submission 7
- 0.2 Acknowledgments 7
- 0.3 Structure of This Submission 7
- 0.4 Resolution of RFP Requirements 8
  - 0.4.1 Mandatory Requirements 8
  - 0.4.2 Optional Requirements 9
  - 0.4.3 Issues to be discussed 9
  - 0.4.4 Evaluation Criteria 10
- 0.5 Proof of Concept 10
- 0.6 Changes to other OMG Specifications 11

Chapter 1 Design Rationale 13

- 1.1 Motivation for an Ontology Definition Meta-Model 13
  - 1.1.1 The Difference between Ontologies and Models 13
  - 1.1.2 The Relationship between Ontologies and Models 14
  - 1.1.3 Knowledge Representation and the Model-Driven Architecture 15
- 1.2 General approach 15
- 1.3 Names 18
  - 1.3.1 Properties of names 19
- 1.4 Structural elements 20
  - 1.4.1 Classes 20
    - 1.4.1.1 Dimension 21
    - 1.4.1.2 Concept versus representation classes 24
  - 1.4.2 Instances 25
- 1.5 Meta-properties 25
  - 1.5.1 Identity of class 26
  - 1.5.2 Unity 27
- 1.6 Associations 27
  - 1.6.1 Structured associations 27
    - 1.6.1.1 Sub-associations 27
    - 1.6.1.2 Part/Whole 27
    - 1.6.1.3 Set/Instance 28
- 1.7 Structured collections of terms 28
  - 1.7.1 Indexed systems of names 28

Table of Contents

1.8 Modules. 30
1.8.1 Revision 30
1.8.2 Extent-descriptive predicates 30
1.9 Players, roles and acts 31
1.10 State. 32

Chapter 2 Using UML for Ontologies..33

2.1 Extensions to UML 2.0. 33
2.1.1 Ontology::Root package 34
2.1.2 Ontology::Representations package 35
2.1.3 Ontology::Multiplicities package 36
2.1.4 Ontology::Expressions package 37
2.1.5 Ontology::Statements package 38
2.1.6 Extending Kernel::Classifiers 41
2.1.7 Ontology::Classes package 43
2.2 Guidelines for use of UML 44
2.3 Supporting multiple ontologies. 45

Chapter 3 Relationship of OWL and UML 47

3.1 OWL Lite RDF Schema Features 47
3.2 OWL Lite Equality and Inequality 48
3.3 OWL Lite Property Characteristics 49
3.4 OWL Lite Property Type Restriction 51
3.5 OWL Lite Restricted Cardinality 52
3.6 OWL Lite Class Intersection 53
3.7 Incremental Language Description of OWL DL and OWL FULL 53

Chapter 4 Conformance..57

4.1 Single Ontology Conformance 57
4.2 Single Knowledge Base Conformance 57
4.3 Multiple Ontology Conformance 57
4.4 Multiple Knowledge Base Conformance 58
4.5 OWL DL Source-Mapping Conformance 58
4.6 OWL DL Target-Mapping Conformance 58

Table of Contents

Chapter 5 References..59

Appendix A Starter Ontology..61

A.1 Introduction. 61
A.2 WebKB2 notation 61
A.3 Starter Ontology 62

Table of Contents

Overview 0

DSTC is delighted to submit this response to the ADTF's RFP for an Ontology Definition Metamodel (ad/03-03-40).

We believe that this Ontology Definition specification offers three main benefits:

- *Expressive power for knowledge representation*
- *Based on UML's semantic foundations*
- *Support for multiple ontologies.*

0.1 Primary Contact for the Ontology Definition submission

The primary contact for this Ontology Definition submission is:

Dr Kerry Raymond
Distinguished Research Leader
DSTC Pty Ltd
University of Queensland
Brisbane 4072, Australia
Phone: +61 7 3365 4310
Fax: +61 7 3365 4311
Email: pegamento@dstc.edu.au

0.2 Acknowledgments

The submitters wish to acknowledge the contributions of A/Prof Bob Colomb, Dr Philippe Martin and Dr Kerry Raymond in the preparation of this specification.

0.3 Structure of This Submission

This Chapter contains contact information and explains how the proposal addresses the RFP requirements.

Chapter 1 explains the design rationale for the Ontology Definition Metamodel.

Chapter 2 demonstrates how the requirements of ontologies can be achieved through the re-use and extension of the UML 2.0 Kernel.

Chapter 3 describes the relationship between OWL and UML.

Chapter 4 describes the various levels of conformance appropriate to ODM.

Chapter 5 contains a bibliography.

Appendix A contains a “starter” ontology, proposed as a standard core for extension by other more domain-focussed ontologies.

0.4 Resolution of RFP Requirements

This section describes how this submission meets the mandatory and optional requirements identified in the RFP.

0.4.1 Mandatory Requirements

The following mandatory requirements are taken from Section 6.5 in the RFP.

6.5.1. Ontology Definition Metamodel representing the semantics of ontologies including but not necessarily limited to OWL ontologies.	The model for ontology definition given in Chapter 2 can express ontologies richer than that of OWL.
6.5.1.1. Depict using UML.	The model for ontology definition in depicted in UML in Chapter 2.
6.5.1.2. Use v2.x of MOF, UML and OCL.	This submission is based on v2.0 of MOF, UML and OCL. However, prototyping has been limited by the lack of availability of tools conformant to these v2.0 specifications.
6.5.2. UML2 profile	This will be provided in the final submission.
6.5.3. Mapping between meta-model and profile	This will be provided in the final submission. We foresee no problems in ensuring that the mapping is two-way and bounded.
6.5.3.1. Support for forward and reverse engineering between ontologies	Section 2.1.6 describes associations for tracking relationships between separately specified Classifiers, while Section 2.3 discusses some of the issues associated with the use of multiple ontologies. Since ontologies can be expressed using an extension of UML 2.0, transformations between ontologies can be accomplished using the outcomes of the ongoing RFP into Query / View / Transformation [ref???].
6.5.4. Mapping between ODM and OWL	The relationship between OWL and UML is discussed in Chapter 3. The final submission will document the mapping (using the techniques from the Query / View / Transformation RFP, if available).

6.5.5 XMI XSD for ODL	This will be provided in the final submission. It will be automatically derived from the ODM using the XMI Schema generation rules [ref???].

0.4.2 Optional Requirements

The following optional requirements are taken from Section 6.6 in the RFP.

6.6.1. Mapping multiple ontologies into a single UML model	It should be possible to map multiple ontologies into a single UML model, by coalescing those elements identified as representing the same concepts. This could be accomplished by a transformation (as defined by the Query / View / Transformation RFP).
6.6.2 Support for round-trip engineering	The class TaggedComment in Section 2.1.1 provides support for the definition of a name-value pairs which can be used to encode information not representable directly in ODM. This provides considerable flexibility for round-trip engineering without loss of information.
6.6.3 Mapping to DAML+OIL or other ontology languages	Not provided in this initial submission, but may be provided in a revised submission.
6.6.4 Support for OWL Full	This ODM specification is capable of expressing all concepts contained in OWL Full. A mapping may be provided in a revised submission.
6.6.5 Metadata to define context and scope of an ontology.	Not provided in this initial submission, but may be provided in a revised submission.

0.4.3 Issues to be discussed

The following issues have been taken from Section 6.7 in the RFP.

Strategy for mapping names between ODM and OWL	A discussion of the mapping from OWL names is given in Section 3.2.
Properties in OWL not supported in UML	A discussion of the mapping from OWL names is given in Section 3.3 and Section 3.4.
Iterative development across mapped environments	Issues relating to iterative development across mapped environments relate to round-trip transformation issues to be addressed in a revised submission based on the concepts in the Query/View/Transformation RFP.

Exclusive use of UML profiling	DSTC does not believe that the exclusive use of UML profiling is sufficient for the expressive power needed for ODM and a full MOF meta-model is presented. Use of a full MOF meta-model enables the use of many MDA tools such as repository generators, XMI generators, HUTN generators, and Query/View/Transformation tools.
Relationship to Business Nomenclature in CWM	The Business Nomenclature in the Common Warehouse Metamodel could be used as the basis for a domain ontology.

0.4.4 Evaluation Criteria

The following Evaluation Criteria are taken from Section 6.8 of the RFP.

Compactness and clarity	The model presented here has been selected to minimise the number of concepts and relationships within the bounds of achieving a desired level of expressive power. In modelling, there is always a trade-off between elaborating more concepts and relationships in a very precise way versus overloading concepts and using a smaller set in a less precise way (compensating with constraints or additional natural language semantics).
Extent of mapped regions	The mappings between ODM and OWL are not completed in this initial submission, but we predict that the expressive power of ODM is a superset of OWL. Hence all of OWL should be capable of being mapped to ODM but not vice versa. A revised submission will contain the mapping and will indicate the aspects of ODM that cannot be mapped into OWL.
Expressiveness of ODM	The model presented in this intial submission has an expressive power greater than that of OWL. Support for expression of types and statements across multiple ontologies enables the co-existence of divergent beliefs, but with support to detect and resolve such divergencies.

0.5 Proof of Concept

DSTC Pty Ltd is currently engaged in a 7 year research programme into Enterprise Distributed Systems Technology with major projects devoted to knowledge representation. DSTC Pty Ltd has extensive experience in the standardisation, implementation and use of MOF, XMI and UML. The DSTC has been developing MOF-based tools since 1996.

DSTC has 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 has been 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.

0.6 Changes to other OMG Specifications

This specification proposes an extension to UML 2.0. As such, no changes to UML 2.0 or other OMG specifications are required. However, we believe some of the extensions proposed in this specification address a wider need than just that of ontologies, and should be considered for incorporation into UML during some subsequent revision. A revised submission will enumerate the extensions recommended for inclusion in UML.

Design Rationale 1

This chapter explains why ontologies are needed in Section 1.1, and then explores the requirement of ontologies in more detail in the following sections.

1.1 Motivation for an Ontology Definition Meta-Model

The Ontology Definition Meta-Model RFP defines an ontology as follows:

> *An "ontology", as used in this context, defines the common terms and concepts (meaning) used to describe and represent an area of knowledge.*

To those familiar with object-oriented modelling techniques, it may seem that an ontology is just a model and that, given the extensive effort already made into the standardisation of UML, an RFP to further extend UML to support ontologies is unnecessary. Many of the requirements of ontologies can already be accommodated by UML. Both ontologies and object-modelling aim to represent "real world" concepts in some systematic form that supports their manipulation by a computer. Although different terminology may be employed, both ontologies and object-oriented modelling are built on classes/concepts and relationships, including the specialised relationships of subtyping and aggregation.

Although the RFP does not explicitly request models to represent knowledge (instances of the ontology), nonetheless it is clear that to define a set of "types" one must have a good understanding of the "instances". Hence this submission addresses the extension of UML instances to support knowledge representation, as it motivates the need to extend UML types.

1.1.1 The Difference between Ontologies and Models

So then why are ontologies different to O-O models? The simplest answer is the difference in their motivation. O-O models are abstractions; they eliminate or simplify concepts and relationship down to those specifically relevant to the task at hand (usually the design of some IT system), whereas ontologies are intended for knowledge representation in which the goal is "if something is known, it should be able to be record in a machine-interpretable manner" in which everything is worth capturing. When specifying an ontology, we do not wish to eliminate or simplify concepts and relationships.

It follows that no ontology should ever be regarded as complete (since scope of knowledge grows continually) and that toolsets which work with ontologies must support their continual update and evolution, as well as their interaction with ontologies.

Although O-O modelling techniques like UML support the specification of instances as well as types, in practice O-O models include few if any instances. Typically the types in the model are used to develop an IT system and then the specific instances are created, queried and manipulated within the implemented system instead. Given that the intended purpose of most IT systems is the management of large collections of information (e.g. thousands of customers, millions of orders), it is not generally regarded as scalable to represent these within the O-O model, apart perhaps for some particularly distinguished instances.

However, the goal of knowledge management is somewhat different; the knowledge is captured for the purpose of inferencing, the logical derivation of conclusions. For the purpose of inferencing, it is often not necessary to record the particulars of all instances; it is often sufficient to express quantified statements about the population of instances. For example, “most DSTC researchers prefer Java to COBOL” is knowledge about the collective instances of DSTC researchers, but does not elaborate the details of any individuals. While it is not possible to query this imprecise knowledge to obtain the list of researchers who prefer Java individually, nonetheless it can be used for inferencing, e.g. that the DSTC organisation might be interested in purchasing site licences for Java tools.

In O-O modelling, the assumption is that instances represent the “truth” and the implemented systems do not usually support the ability to hold alternative values. However, in knowledge management, the assumption is that the statements (instances) represent with beliefs/opinions and that there may be multiple different pieces of knowledge about the same subject, some of which may be inconsistent. For example,

“Most DSTC researchers prefer Java to COBOL”

“Some DSTC researchers prefer Java to COBOL”

“Some DSTC researchers prefer COBOL to Java”

“Most DSTC researchers prefer COBOL to Java”

Only the first and fourth statements are inconsistent but the first three (or the last three) are all different but not inconsistent. Generally a knowledge base will permit inconsistent knowledge to be created by different sources (known as divergent beliefs) but will not permit a single source to create inconsistent knowledge. Therefore, when working with knowledge, it is important to capture and preserve the source of that knowledge, as inferencing may wish to favour certain alternatives based on their source (e.g. what is the quality of information from that source).

1.1.2 The Relationship between Ontologies and Models

An O-O model should always be an abstraction of some more complete ontology, in which we have selected particular concepts and relationships as being “important” to the task at hand while other concepts and relations less important to the task at hand may be combined/summarised/simplified or even omitted entirely. As a consequence, there should be a transformation from some ontology to any O-O model. The value of explicitly defining this transformation (especially in a declarative way) is that it provides a semantic basis for interoperability between IT systems built from different

O-O models but fundamentally "about the same domain" via the ontology. The implication is that O-O modelling should not be done as a "green field", but through the highlighting of concepts and relationships of interest in an ontology.

1.1.3 Knowledge Representation and the Model-Driven Architecture

Knowledge representation with all its imprecision and divergency is nonetheless very important for the Model-Driven Architecture. In the MDA, transformations are used to derive systems from models. However, it is widely agreed that "one size fits all" transformations are unlikely to produce satisfactory systems, and that transformations must be parameterised, customised, or guided with additional information to ensure that the resultant system has "the right non-functional or stylistic characteristics, which cannot be determined from information in the PIM" [1]. The MDA Guide [1] describes this additional information as:

> *Often the additional information will draw on the practical knowledge of the designer. This will be both knowledge of the application domain and knowledge of the platform.*

The specification of non-functional and stylistic requirements and the designer's knowledge of the domain and the platform is usually expressed in the language of knowledge representation, potentially imprecise and potentially divergent, from which conclusions about design choices must be inferred. It is knowledge like "the bandwidth between the bank branches is pretty limited but it's good internally" that often shape the design of our systems. Frequently it is knowledge of the "general case" that is important for optimising system performance, but in O-O modelling techniques like UML, it is difficult to distinguish the general case from the full range of legitimate (but perhaps rare) possibilities. How often will this optional attribute "name" of class Person actually have a value? Will there to be more Customers or more Branches in the Bank?

1.2 General approach

"An ontology is an explicit specification of a conceptualization" [2]. "[An ontology characterizes] the structure and behaviour of things in the real world" [3]. An ontology is therefore the same sort of thing as a conceptual model such as is customarily represented in a UML Class diagram. We want to distinguish ontologies from models, so we restrict the use of the term "ontology" to models which are apart from any particular application. These ontologies are typically used to facilitate interoperation among more-or-less autonomous systems. In particular, ontologies are prerequisites for natural language interfaces with information systems.

Ontologies differ to a degree from models since they often mix levels. The Koechel catalogue of Mozart's music consists largely of instances, while the Bib-1 set of use attributes from the Z39.50 standard consists exclusively of types. Major systems like SNOMED [4] often consist of a mixture of types and instances. Product catalogues consist of items which in the model are often instances (names of products) but in an ontology are taken as types (there are typically many instances of a named product).

An object in an ontology will often be represented many times in a model, often in different ways. Further, there will often be many objects in a model which are not in an ontology since they refer to implementation of objects or actions participating in interoperations but are not themselves visible (like the warehouse bins supporting an order entry application).

An ontology can also be seen as a machine-interpretable repository of a body of knowledge (the two views are not exclusive). Although it is now very common for bodies of knowledge to be stored in machine-readable form as text, knowledge in an ontology must also be accessible to inferencing systems implemented as computer programs. It must therefore be represented in formal languages as distinguished from the natural language of text representations.

A large number of formal languages have been proposed, none of which has the full power of expression of natural language. Moreover, many of these formal languages are either experimental, controversial, or both. The best-established formal language is the first order predicate calculus (FOPC), with some small extensions such as modal logic. While automated inferencing systems exist for FOPC, practical considerations lead most people to restrict their knowledge representation to subsets of FOPC which are more computationally tractable. The most relevant for the present proposal are the relational calculus, as exemplified by SQL; datalog, as exemplified by SQL with recursion as in SQL:1999; and description logics as exemplified by OWL. So the present proposal recognizes that the representation systems used will limit the knowledge representable [5].

We also recognize that it is not possible to fully define any knowledge, in that any formal system rests on undefined primitives [6]. We further recognize that it is not possible to represent all of the knowledge in any world. This is because declarative knowledge must always be interpreted in the context of a body of practices, norms and expectations in the society to which the knowledge applies [7]. But it is certainly possible to represent sufficient knowledge for practical purposes, as attested by the large number of ontologies which have been created and are widely used [4][8].

Ontologies are generally representations of knowledge about aspects of the world limited to the concerns of particular groups of people and organizations, for particular purposes. Hardly any of the ontologies presently in use purport to be universal. All ontologies are representations of reality. However, it is worth noting a distinction between physical and social reality, in particular between physical facts and institutional facts [7]. A physical fact exists independently of any human society, while an institutional fact exists only in the context of a society. A man and a woman are physical facts, but a marriage is institutional.

Institutional facts are created by social actions called speech acts. An institutional fact is a record of its creating speech act having been made. At a physical level, an institutional fact is the sum of its representations. One representation can be copied, but if all representations are lost, the institutional fact ceases to exist – it is forgotten. Most information systems are mostly about storing institutional facts. In a community of interoperating systems, different aspects of institutional reality are created by different players. Much of the activity of interoperating systems can be thought of as epistemological rather than ontological, since it consists of some players in the interoperating community finding out about changes in the institutional reality made by

other players. (This gives another distinction between ontologies and models. Models tend not to separate the ontological from the epistemological, whereas ontologies must.)

Finally, modelling languages in general and UML in particular generally separate structural descriptions from integrity constraints and predicate statements. Some integrity constraints (e.g. cardinality) are often expressed as annotations on the structural descriptions, but most systems permit general integrity constraints to be expressed in a textual language. There are many well-established such languages, including UML OCL, SQL;1999 assertions [9], and OWL. This proposal does not attempt to develop a new constraint language, but provides a set of structural description capabilities intended to be suitable to ontology development, which are capable of being translated into descriptions suitable for inclusion in one of the established constraint languages. Similarly, simple predicates can be expressed in the modelling language, but more general predicates can be expressed in one of the established textual languages.

In a system of interoperating information systems there is a difference between ontological and epistemological considerations. The ontology describes the way the world is in which the systems interoperate, while the models of individual systems reflect what they know about the world. In fact, many of the actions of individual systems have the function of obtaining information which already exists.

The community in a university involved in courses, enrolment and grades is loosely coupled and held together by standards and ontologies. The community includes the Faculties who are responsible for course offerings and student admissions, the students who are responsible for enrolling in courses, and the lecturers who are responsible for assigning grades. Some of the information is held in the central student records system, but much is held in idiosyncratic spreadsheets in the workstations of lecturers and students.

The ontology is that there are students, courses and enrolments (association between student and course). An attribute of enrolment is grade. In the lecturer's spreadsheet is recorded the various components of the grade in terms of the results of various assessments.

Let us assume that students maintain spreadsheets containing their record of study (enrolments and grades). Assume that the Faculties, lecturers and students share the schema involving students, courses, enrolments and grades; and that the students and lecturers share the schema involving grade components. Instances of students and courses are created by the Faculties.

Instances of enrolments are created by the individual students. Values of the grade attribute are created by the Faculties on advice of the lecturer. The lecturer creates the values of the individual assessments making up the grade.

Students must find out their student ID and verify their enrolment, updating their individual records. Lecturers must find out the students enrolled in their courses (class lists). The Faculties must find out the grades advised by the lecturers. The students must find the grades awarded by the Faculties and the individual assessment reports awarded by the Lecturers.

So we have an ontology at the type level which pre-exists, and at the instance level which is created by various parties, with the information being propagated to the other parties in various ways. Aspects of the world are seen as fixed by those parties who cannot create them.

1.3 Names

Whatever structure an ontology has, there will be in any ontology a collection of concepts. These concepts will be modelled as classes, associations, attributes and possibly other structures. But at the most basic level we need for each concept a name, representation, description, see also, and version. Some reference is made to the UML Business Nomenclature model (BN).

Name – a concept has a name, e.g. ‘man’, but this name will differ among the possible users of the ontology, partly due to naming conventions in different systems but more due to multiple languages in a world-scale information system. Our ‘man’ could be someone else’s ‘male’ or ‘uomo’. So in the environment of the ontology, there will be a collection of synonymous names. Analog of ‘vocabularyElement’ in BN. Names are not necessarily atomic, but can be composed from atoms according to formation rules expressed in an appropriate system such as Backus-Naur Form or an XML DTD.

Representation – the concept needs a language-independent representation. This will generally be an alphanumeric identifier of some kind. These exist in for example the Z39.50 use attributes and in the various EDI standards. The representation is the primary key for the set of synonyms. BN has sets of synonyms and preferred terms, but not quite the representation called for.

Description – a natural language statement which provides a definition and links into the semantics of the concept. This is necessarily informal, since the ultimate source of the semantics is an only partially articulated world of practices and attitudes. BN does not seem to have this facility, but it is the core of the GILS semantics project [10].

See also – when a user is selecting a concept to use for some purpose they use the name and definition to help decide which concept to use. Say they are assigning an instance to a class. Particularly in larger systems there may be other classes which are actually more appropriate but the user didn’t think of. Many systems of nomenclature have cross-references to other concepts to give the user a sense of what possibilities there are and make the choice more reliable. An extreme example of this comes from the Library of Congress classification schedule [11] where the following note is attached to the schedule entry TD *Environmental technology. Sanitary engineering*:

> *The promotion and conservation of the public health, comfort and convenience by the control of the environment. Cf GF Human ecology HC68, HC95-710 Environmental policy (General) HD3840-4730, Economic aspects (Government ownership, municipal industries, finance, etc.) QH75-77 Landscape protection RA565-604 Public health S622-627 Soil conservation S900-972 Conservation of natural resources TC801-978 Reclamation of land TH6014-7975 Building and environmental engineering*

Notice how diverse the classifications which must be considered before a work is finally assigned to a subclass within TD. A small difference in judgment can result in a work being assigned to widely differing classes.

BN does not seem to have this concept. It is distinct from relatedTerm, since relatedTerm designates an instance at the other end of an association, and from ‘synonym’ since the see also terms are not synonyms but terms with meanings close enough that a user might confuse them. (This comment is itself an example of a see also note.)

Version – we leave the details of versioning to another group, but there is a requirement to know which version of the ontology is the last version in which a concept was changed.

1.3.1 Properties of names

There are clearly class, association, attribute and function names, which are implemented in the schemas and modelled in the Class diagram or as signatures of methods. This gives rise to the possibility of formulas composed from these schema-level names, instance names, and what in programming or logic are generally known as variable names. Formulas must be well-formed according to some formation rule.

There are names of instances which refer to entities in the world: student names and identifiers, course names and identifiers, and values of grades, assessments, and various aggregations (e.g. average mark on an assessment, average percentage mark for the course, student’s grade point average).

It seems that a key classification of names is whether the name of the referent is known or unknown.

Names with known referent are such as ‘Bob Colomb’, ‘80823341’, HIST3404, ‘Medieval History’, ‘6 (grade of)’, “78% (percent mark)”. Notice that all these names are composed of strings of symbols drawn from the English typewriter keyboard. Some of these symbols are numerals, some are not.

Names whose referent is unknown can be either the result of calculation (bottom student in the class), or not yet created (grade before the end of the semester).

But names are subtle. The same thing can be designated by multiple names (synonyms). A thing can be designated declaratively (16 Main St.) or procedurally (second house on the right in the first street to the south of the Junction Hotel). A name may designate a unique individual only within a context, and the context can be pretty narrow. A thing may be named even if it is not known (the result of a calculation not yet carried out, or the result of a query not completed), or even if it does not yet exist or might never exist (The name ‘Robert’ was selected as the name for the first son of one of the author’s mother’s parents. Since they had only girls, the name never designated until he came along.) The result of a calculation may be designated simply by a formula. Further, names composed of digits are often numbers, but sometimes not. ‘3’ names a telco, and the student numbers are not generally usefully thought of as integers.

Because of this subtlety we propose that the ODM not make any commitment regarding names. OWL does not. Names may or may not designate unique objects, and are subject to a variety of constraints and axioms. I propose that a name be allowed to be a well-formed formula of a designated set of formation rules, and be subject to a designated set of axioms. A simple way to represent some of the axioms are for a name to be constrained according to the axioms associated with any functions of which it is in the domain. So if a type is necessarily the domain of an arithmetic operator, it must be an appropriate digit string and support the operator specified. This gives a simple way to introduce dimensions and measures.

Representing names by formulas may seem strange to a programmer, but it is extremely common among users of spreadsheets. A lecturer might have 125 students in a spreadsheet whose final percent mark is designated by a formula, and whose grade is designated by another. The formulas may be partially instantiated in that three of the four assessment items may have been marked.

Under this proposal, an ontology builder is free to make any formation rules and constraints that are suitable for the situation.

1.4 Structural elements

1.4.1 Classes

As already in UML, the main structural elements are classes and associations. A class can have attributes, which are a light-weight form of association. The value set of an attribute has all the characteristics of a class. Both attributes and associations are *properties* pertaining to a class. Properties can be mandatory or optional, and can be subject to other cardinality constraints.

A class can be either a class of identifiable individuals (a *countable* class), as in UML, or can be a *bulk* class. Examples of bulk classes include water, which comes in volumes; money which comes in amounts; wheat, which comes in either masses or volumes; or anonymous individuals like apples, pens or sheets of paper, which come in quantities (cases, dozens, reams). One can identify something as being some of a bulk class, but can identify below that only if there is an association with a countable class of containers, either physical or metaphorical. We can identify the water delivered to account 54129 for the period 1 January to 31 March, 2003; or the amount of money owed in respect of that delivery; or the wheat in truck VH9302 at 10:00 AM on 23 June, 2003; or the pens in stock in the Southbank warehouse at the close of business on 30 June, 2002.

Classes can be related to each other as subclasses. A subclass can be either *asserted* or *defined*. A defined subclass satisfies a predicate on the properties pertaining to a class. An asserted subclass is simply declared to be so. Classes can be in an asserted subclass relationship in the absence of any properties. We may have two defined subclasses of *Product*: *Inventory* and *Service*, distinguished by whether or not there is an inventory record for the product. We may have two defined subclasses of *Student*: *Passed* and *Failed*, distinguished by whether the grade is respectively greater than or equal to 4 or less than 4. We may have separate tables for invoices and payments but declare them

both as asserted subclasses of *Financial transaction*. Hierarchical term lists found in many primitive classification systems, such as the ACM subject classification are defined as asserted subclass systems.

Classes can be complexes of parts. As with the subtype complex, there are restrictions on part structures. In particular, a whole can not be a part of itself. There are a number of other axioms or restrictions which have been discussed in the mereology literature. One of these is whether the part relationship is transitive or not. We recommend this distinction: The whole/part relationship may be transitive. The set/member relationship is not.

1.4.1.1 Dimension

Class names can be used in statements like “this bottle has a capacity of 0.5 litre”. In UML this sort of statement is often modelled by treating ‘this bottle’ as an instance of a class “bottle”. The class ‘bottle’ is the domain of an attribute function called ‘capacity’ whose value set is of type ‘real’. Another way to model it is to have an association ‘has-capacity’ one end of which is class ‘bottle’ and the other a class ‘volume’ which is populated by several real numbers (say there are 5 standard bottle capacities: 0.3 litre, 0.6 litre, 1 litre, 2 litre, 4 litre). The association would be functional.

What to do about the term ‘litre’? In the value set option, it has no place. In the association option it is part of the name, either of the class ‘capacity-litres’ or of each capacity instance as shown in the example.

This is an instance of the general problem of dimension. There are a large number of dimensions and units of measure. One reason one might want an explicit representation of the dimension of an attribute or class is that different dimensions permit or forbid certain arithmetical operations. For example, it is semantically valid to add two volumes of the same unit, or two distances of the same unit. But it is not semantically valid to multiply two volumes, while it is semantically valid to multiply two distances (giving a different dimension, ‘area’). It is questionable to add two temperatures. One can compute the average of a set of temperatures, but it does not make sense to add the temperature of the cup to the temperature of the coffee poured into it.

Another reason is that a system of dimensions is an aspect of reality that can be factored out of specific applications. A system of dimensions includes

- the names of the dimensions: distance, area, volume, time, charge, voltage, acceleration, capacitance, etc.
- rules for determining the dimension of the result of a calculation on dimensioned objects: area is length x length, acceleration is length / (time x time), etc.
- a system of interdefined units (metres, kilometres, millimetres; grams, kilograms, milligrams; square yard, perch, acre).
- The rules can incorporate some physics or other science. For example, the temperature of the cup can’t be added to the temperature of the coffee, but the temperature of the filled cup can be calculated using a formula involving mass and specific heat of the constituents as well as their temperature.

Given that there are lots of reasons to want to include dimension in our ontology model, the question becomes how to do it.

One possibility is to introduce a system of annotations to classes so that the class ‘volume’ becomes ‘volume’ – dimension note volume in litres’. However, this approach introduces a new and necessarily complicated (as we will see below) modelling construct, so should be avoided unless necessary.

Since UML already has classes and multiple inheritance, one might try representing dimension as class. All dimensions can be represented in units in generally more than one way. Dimensions and their units often come in systems (e.g. the American foot-pound-second, the SI meter, kilogram, second, the SI centimetre, gram, second). Money is a dimension with units the various currencies, which is a sort of system. But there are many dimensions that are idiosyncratic, such as “burn-rate” with units million US dollars per month used in the venture capital industry, widely reported during the dot-com boom of the late 1990s. The same unit name often occurs in more than one system. Sometimes the units are the same, as between the British and American ‘mile’, and sometimes different as between the British and American ‘gallon’ or the Troy and Avoirdupois ‘ounce’. Sometimes the same unit name refers to two different dimensions, such as the (fluid) ounce versus the (unit of force) ounce.

The most general structure is the dimension system. Individual dimensions are not themselves dimension systems, so it is not appropriate to consider them subtypes. We propose rather to treat the dimensions as parts of a dimension system.

A dimension exists independently of any units. It can be referred to ostensively “that distance” or concretely “the distance from Ghent to Aix”. Any association or attribute value set relating to the same dimension is comparable. So we propose to treat any dimension as a bulk class. So a dimension system consists of a whole with bulk class parts.

Classes support methods, so the collection of methods supporting a dimension system can be packaged with the class system. General facilities such as addition or equality are optional anyway, since, for example, reals do not support equality, dates do not support addition and GIF objects do not support greater than. So it is consistent for a subclass to forbid an optional method.

Each dimension in a system is represented by any of a collection of units. It does not make sense to consider the units as subclasses of the dimension, since each dimension necessarily is represented by all of the units. On the other hand, a concrete dimension is almost always represented by a specific unit. The unit can be represented as part of the target of an association instance (USD134; 6 pounds, 5 shillings, 3 pence; EUR100) are all valid instances of the bulk class ‘money’. The currencies and values can be extracted by parsing using the formation rules. But usually the unit is factored out, so that all instances of an association have the same unit.

In this case we consider the various unit names as designating *representation* classes, as distinguished from the *concept* class which is the dimension within its system. (The distinction between concept and representation classes is more general, and is discussed in Section 1.4.1.2 below.) Associated with a concept class is a set of representation classes.

So a water supply utility would have a class “product” which is the target of an association “consumed” whose source is “customer”. Product would inherit from the bulk class “water” and also from the representation class “kilolitres” associated with the SI system class part “volume”. Similarly, there would be an association “owing” between “customer” and “amount”. The latter class would inherit from the concept class “money” and from the representation class “USD”. An automated reasoning system would navigate through the class/subclass, set/instance and whole/part structure of the ontology.

The relationship among the dimensions and among the units within a system and between systems must be represented as a series of formulas. This is similar to the need to represent assertions, integrity constraints, subclass definition predicates and derived class rules in standard UML. We propose to permit any agreed textual representation language.

The dimension issue is made more complicated by the fact that a specification that a bottle has a capacity measured in litres is not sufficient for interoperation. A program needs to know not only that the class is in litres, but how the litres are represented as numbers (integers, reals, magnitudes in powers of 10, including range and precision). Further, it needs to know how the numbers are implemented (integers as 16 bits least significant bit first, reals as IEEE double float, etc.). So the annotation approach would have to cater for at least these aspects (dimension concept, dimension unit representation, unit number representation, number implementation), making it more unwieldy, while it all fits naturally into a class with multiple inheritance (faceted) approach.

For example, we could first develop our ontology using a class system appropriate to the application, including a bare class ‘capacity. Later we could increase our ontological commitment by importing a dimension system including volume represented by litres, and assert that capacity is a subclass of litre, with the capability to navigate to volume and to the SI system. Still later we could increase commitment again by importing a number system and assert that litre is a subclass of the COBOL number with five digits, two digits after the decimal (designated in the COBOL system as ‘999.99’). Finally we could complete commitment by asserting that an appropriate superclass of ‘999.99’ is a subclass of ‘IEEE double float’. In implementation practice there would be some subtle problems of inheriting methods from multiple superclasses, but I think they are pretty well resolved in contemporary O-O systems.

The proposal so far has ignored the representation of the capacity of a bottle as an attribute function. This can be handled by importing the concept of ‘domain’ from the SQL:1999 standard. A domain is essentially a class, so the preceding mechanism applies directly. The value set of the attribute function can be specified as a subset of the domain class, thereby inheriting all the methods and constraints applicable to the class.

Using domains, there is no longer a difference between an attribute and an association. We therefore recommend that attributes not be used with ontologies. Derived attributes can be represented using either derived associations or alternate representations (as appropriate).

Note that the SQL:1999 standard has flagged deprecation of the ‘domain’ construct, probably because within a single information system no-one writes programs that violate the domain constraints anyway, so the construct tends to be redundant. It appears that for interoperability the domain construct is much more important, especially for interoperating autonomous agents.

Dimension is involved in what sorts of things one can say about a class. A question as to whether one can say 2.5 litres or 2.5 people resolves into one of two questions. First, is 2.5 a valid value for an attribute (or range of a functional association)? This is settled when the ontological commitment reaches the level of a number system. Second, does it make sense to say that a derived attribute of a class can be 2.5? Here it partly depends on whether our class is defined extentionally or intentionally. If extentional, the class is a set with a finite number of members, so the standard aggregation operations apply, including count and average of a property. If the class is intentionally defined (litre, ‘999.99’) or a bulk type (money) then aggregation operations do not apply at all since there are no individuals to aggregate.

1.4.1.2 Concept versus representation classes

We introduced the distinction between concept and representation classes with the discussion of dimension in Section 1.4.1.1. This distinction is more general. It is made for example in the conceptual modelling system Object-Role Modelling [12], where what we call a concept class is called a non-lexical object type and what we call a representation class is called a lexical object type.

We have seen that in the case of dimensions we need to distinguish a concept from its representation. Interoperability often generates this requirement. For example, in an e-commerce application, buyer and seller share a concept “product”, but they identify a given product differently. There is a concept class “product”, and two representation classes “seller product ID” and “purchaser product ID”. To enable interoperation, a correspondence between the two representation classes must be established, which could be done by the buyer, seller, both, or by an exchange.

For another example, a system supporting a coronial enquiry is keeping track of information relating to a death. The system has a concept class “murder”, but the whole business of the enquiry is to decide whether the particular death is an instance of that concept. This sort of situation is very common in expert systems applications, where the whole business is to determine whether a particular data set is an instance of one of the outcome classes.

Although most information stored in information systems refers outside the system itself, single applications often do not in practice make the distinction between concept and representation. This is partly because the relationship between the system and the world is generally mediated by the human aspects of the organization, and partly because most of the information in most information systems is institutional facts, records of speech acts, so that the concept is very close to its representations and the representations are often isomorphic. A computer record of an invoice has the same data as a paper record of the same invoice. Most widely-used modelling systems, including UML class diagrams, do not make this distinction, and not all ontologies would, either.

We therefore recommend that a class be capable of designation as a concept class, representation class, or both.

1.4.2 Instances

A shared world will often contain instances, both of classes and associations. Consider the Koechel catalogue of Mozart's music. All of the individual works are instances, as are the associations between the works and the date composed, date first performed, patron, and librettist of the operas. It is therefore necessary to include instances in the model of the ontology.

Representation classes obviously can have instances. Concept classes in principle cannot have instances since they are independent of particular representations, although for convenience one might allow a natural language representation not intended to be used other than as a description. Individual classes can have instances. Bulk classes are not sets, since their contents do not carry unity, but they can have quasi-instances, generally the target of an association with an instance of a countable class. They can also have a fixed set of possible quasi-instances, say a series of standard volumes for containers in a soft drink application.

Since classes, instances and quasi-instances are potentially all parts of an ontology model, they can all appear in the vocabulary of predicates in the constraint/ assertion/ rule languages used in the ontology.

1.5 Meta-properties

An object needs to be able to be identified in terms of its properties (an *identifying relation*). A complex object with necessary parts needs some way to tell which parts compose which whole (a *unifying relation*). A property whose values are necessarily always the same for a class is a *essential* property for that class. A property whose values are part of the definition of a subclass is a *rigid* property for that subclass. Since the ontology is defined outside any particular system, which properties constitute the identifying and unifying relations for any class must be part of the ontology, as meta-properties designated by annotations. The particular set of meta-properties here come from the OntoClean system [13].

A class *supplies* a property if it holds for that class and not for any superclasses, while a class *carries* a property it inherits from a superclass. The OntoClean method annotates properties with their formal meta-properties:

- identity +I
- unity +U
- essentiality +M
- rigidity +R

In general, if the annotation of a property is +A, where A is one of the meta-property annotations, this means that in all objects of the class the property necessarily has meta-property A. So the annotation +R on a property of a class says that the type carries rigidity.

There are weaker attributions of meta-properties. If a property is annotated –A, then the meta-property A does not necessarily hold for all objects in the class. It might hold by accident, but not necessarily. A property annotated –M is not essential for all instances, so that different instances can have different qualities. The sign '-'indicates 'not', so –R is 'not rigid', -M is 'not essential', -I is 'not identity', -U is 'not unity'. If a property is annotated ~A, then the meta-property is not necessary for any instance. A property annotated ~M can be updated, so that the value of the property associated with any object may change. The sign '~' indicates 'anti', so ~R is 'anti-rigid', ~M is 'anti-essential', ~I is 'anti-identity', ~U is 'anti-unity'. We have a concept of *strength* of a meta-property. For meta-property A, +A is stronger than –A, since A is necessary implies that A is possible. We also have that ~A is *inconsistent with* +A, since necessarily not A is inconsistent with necessarily A.

The relationship between +, -, ~ M/ R and mandatory/optional is:

- R must be mandatory (note +R -> +M)
- M must be mandatory (note ~R -> ~M, so +M cannot be ~R)
- R, -M, ~R, ~M can be optional

1.5.1 Identity of class

Given an object, one needs to be able to identify the class(es) to which it belongs. For physical objects this is done as a predicate on its properties (man is a featherless biped, water is H_2O). For institutional facts, the class is generally identified at least partly lexically. A marriage certificate has "marriage certificate" printed on it. An invoice is a record in the file called "invoices". Money is the amount in the "balance" field of the "account" record, which will be designated as being drawn from the appropriate bulk class, which is identified lexically.

Identity of subclasses to which an object belongs is determined according to the subclass definition predicate if the subclass is a defined subclass. If the subclass is an asserted subclass, then the identification process takes place outside the world, by say the human administrators. This is always lexical. The human operator enters a financial transaction in the *payment* screen rather than the *invoice* screen, so the financial transaction is of the appropriate subtype. The object always enters the world with its subclass attached to it, as it were.

Note that what are syntactically defined subclasses are often practically asserted subclasses. An e-commerce world may have the concept of a customer entitled to a special discount, represented by the value of a *discount type* attribute in the customer class. But this attribute is set as a result of a negotiation between the management of the business and the customer as a judgmental speech act rather than by a rule on other properties.

Individuals in countable classes also need to be identified in terms of the values of their properties. This facility already exists in UML with keys and candidate keys.

1.5.2 Unity

Similarly, if a relation is anti-unity (~U) for a supertype, then it can be an essential unifying relation for none of its instances, therefore it can not be an essential unifying relation (+U) for any subtype. For example, we can say that the relation *c*, that a student has passed all courses, never includes all the speech acts necessary for a student to be awarded a degree. There is always a further request to the proper authority for the granting of the credential, which must be agreed to by the authority. In some fields this requires an additional examination (bar exams, medical specialist exams, etc.). Suppose we have a supertype *Qualified* with subtypes the various qualifications and kinds of qualifications. One might be tempted to include a subtype of people who had completed the coursework and but not sat the examination (call the type *Ready for qualification*). Our relation *c* is +U for the new type. Therefore making *Ready for qualification* a subtype of *Qualified* would violate the unity constraints on subsumption, even though it satisfies the identity constraints (*student#* is +I for all subtypes).

1.6 Associations

The ontology should support relationships or associations, and these relationships should have a broader term/ narrower term structure. Also standard UML cardinality constraints. Note that relationships must be able to cross the instance/schema boundary, e.g. “Kerry likes ice-cream”. (Kerry is an instance of *person*, while *ice-cream* is a bulk class).

Associations can be n-ary.

1.6.1 Structured associations

Associations can have the same subclass, part or set-instance structure as classes**.**

1.6.1.1 Sub-associations

Associations can have sub-associations which follow the subtype/supertype model. Note that UML already includes these.

1.6.1.2 Part/Whole

There is an association “guidance” between the classes “human” and “spirit”. This association has two parts “guardian angel” and “tempter”, with the spirit instance of the former derived from the subclass “angel” of spirit and the spirit instance of the latter derived from the subclass “demon” of spirit. These parts are both mandatory.

A technology supplier maintains an association “account management” between its marketing staff and its customers' staff. Each “account management” association has two sub-associations. The first is “lead” with the staff instance in the subclass “sales

reps" and the customer staff instance in the subclass "decision maker". The second is "technical" with the staff instance drawn from the subclass "pre-sales support" and the customer staff instance drawn from the subclass "technical responsible".

1.6.1.3 Set/Instance

In real estate auctions there is a complex association between agent staff and buyers. One can model this as a ternary association among customers, agent staff and properties. For a given property there is an association between each buyer and an agent. The agent's job is to keep their buyer in the auction as long as possible.

1.7 Structured collections of terms

Hierarchies are one way of structuring a collection of terms, but there are alternatives including faceted systems and formal grammars. This is especially necessary to control ontological commitment. The proposal should permit this, even if for implementation convenience there is a hidden "top" to the system.

A faceted system has multiple top classes, where the class system supports multiple inheritance. Each top class can support a subclass structure, possibly a large hierarchy. Each top class and its subclass structure is called a *facet*. The facets can be represented as parts of the class structure inherited by a given multiply-inherited subclass.

The need for constraints expressed in formal grammars can be illustrated from SNOMED. We might want every consultation classed with the "Diagnoses" facet to also have a classification in the "Functions, symptoms, etc." facet; and every consultation classed with the "Procedures" facet also classed with the "Topography (Anatomy)" facet. These constraints could be expressed as XML DTD expressions. The medical records function of the most popular general practice software in Australia, Medical Director, works this way.

1.7.1 Indexed systems of names

The general problem of indexed systems of names is illustrated by another look at the student records example. There are many players in that system, divided into three classes: faculties, students and lecturers. The objects classes in the ontology include students (as enrolled in degree programs), courses, enrolments and grades. Responsibility for creating instances of these classes is distributed among the players. Students (as enrolled in degree programs) are created by a cooperation between the Faculties and students (as social agents). Courses are created by Faculties. Enrolments are created by students (can be thought of as either social agent or as enrolled in degree program. The integrity constraints suggest the latter is more appropriate.) Grades are created by faculties on the advice of lecturers, and so on.

Players are semi-autonomous. They each manage their local affairs constrained by their commitments to the common ontology. Each student (as social agent) can maintain an application functioning as an avatar representing them (as enrolled in a degree program), which keeps track of the courses in which they are enrolled, due dates for assignments, marks for assignments, grades for courses completed, and so on. Each

lecturer maintains applications managing the class list for each of their courses. Each course knows about the students enrolled, and has local structure – some have only grades while others have a complex structure of individual and group assignments, with marks for each aggregating into a grade for each enrolled student at the end. To a faculty there are collections of students (as enrolled in degree programs), courses, and enrolments with eventually grades.

A conventional view of this system, as would be commonly represented in a UML class diagram, is from the point of view of the University. Students and Courses are classes, with a many-to-many association representing enrolment. The association is reified so it can have attributes such as a grade. Each student is seen as a view giving a record of study, while each lecturer is seen as a view giving a class list. From the present perspective, this way of looking at things is problematic because it does not leave much room for local autonomy in the way the students or lecturers manage their affairs.

On the present perspective it makes more sense to think of a collection of student avatar classes, each of which has an association with enrolment in the Faculty, and a collection of classlist avatar classes, each of which also has an association with enrolment. The student avatars can see courses through enrolment by transitive relationship, as can the classlist avatars see students.

This way of looking at the situation gives a large number of student avatar and classlist avatar classes. These classes are in one-to-one correspondence with the populations of the students (enrolled in degree programs) and course classes maintained by the faculty. We can therefore think of the names of the student and classlist avatar classes as indexed by the instances of the faculties' student (enrolled in degree program) and course classes.

Note that if we are maintaining a distinction between classes which have members and sets which have instances, with an individual being assigned to a class by a rule, then we are talking definitely about sets and instances, not classes and members. The student avatar classes are indexed by the set of student identifiers, while the classlist avatar classes are indexed by the set of course identifiers. Both of these are sets of names, and there is no possibility of a rule by which an individual name can be determined to be an instance of either set except by looking.

What we have done is to separate the ontology from the representation. Tables and views are representational issues. After all, it is possible to represent any database in a table with scheme {Relation name, Attribute name, Attribute value}.

This kind of situation is very common in enterprise level data warehousing. The data warehouse star scheme has many hierarchically structured attributes which are used to index various levels of aggregation of many and diverse operational transaction-based systems. Different parts of the enterprise see different aspects as real. It makes sense to see the structure as possibly several layers of classes indexed by sets of names.

A complex example of this phenomenon is given by the Standard Industrial Classification (SIC) scheme maintained by the US Department of Labor. Instead of a single set of course names, the class names are indexed by a complex of sets of names, where the SIC system has parts which are a set of Division names, each Division has

parts which are sets of Major Group names, each Major Group has parts which are sets of Industry Group names, and each Industry group has parts which are sets of Industry names. All part classes are mandatory, so all of the classes whose members are individual firms/organizations are organized into subclasses four deep, even if as in the case of Forestry Services, the Industry Group class Forestry Services has only one subclass at the Industry level, also called Forestry Services.

These names of course are complex, modelled by the names model of Section 1.3. In particular, each class has besides its English language name (e.g. Forestry Services) an identifier with scope global to the whole SIC system (e.g. Industry Group Forestry Services is identified by ‘085’, while Industry Forestry Services is identified by ‘0851’).

1.8 Modules

An ontology generally derives from several distinct authorities, even if it is actually constructed by a single software development group (e.g. in SNOMED, the authority for anatomy is different from the authority for pharmaceuticals, and there are several different authorities for diseases.) One would expect there to be a large number of published ontologies for various aspects of systems, for example addresses, currencies, physical units, numbers, number representations.

Moreover, a particular ontology making use of other ontologies will typically include additional structures specializing classes from the modules and establishing relationships among classes from different modules. For example, an ontology supporting a particular e-commerce exchange may import a general e-commerce ontology having a concept “product”, but specialize it with a number of application-specific product subclasses. It may import two different dimension systems and introduce conversion rules between comparable units (English mile = American mile, English gallon = 1.25 American gallons).

An ontology should record the authority for each concept as an aid to maintenance and understanding. This could be represented as a module structure.

1.8.1 Revision

Further, because the published ontologies can be expected to be revised over time and because a particular ontology using them will at least create links among elements of diverse ontologies if not include specialisations etc., the ontology representation needs to support version control down to the atomic level. There is another OMG RFP in this area, so we will limit ourselves to a statement of requirements.

1.8.2 Extent-descriptive predicates

There is often information *about* a world which is not properly part of the ontology. Examples are the size of populations of class instances, the specificity of attributes (Relates to the frequency of occurrence of an attribute value within a class population. An identifier within a class is maximal specific, a rigid property of the class is minimally specific.), the likelihood that a class instance will appear in an optional

association or have an optional attribute, how often a class has instances added or deleted, likely values for an attribute within a range of valid values, default values, and so on.

This kind of information can be important for several reasons.

We will sometimes want to use an ontology as a model of a world for which we what to develop implementations, either conventionally or automatically. This sort of information about the world is used to inform design decisions. A class with a small population can be cached. A class whose population is never updated can be stored on CD-ROM. A rarely occurring optional association can be implemented in a separate relational table, while a rarely missing optional association may be better implemented as a foreign key attribute with null values allowed. When generating software for a within-enterprise communication application, there may be different design decisions made depending on likely values of network latency between organizational nodes.

We will sometimes want to construct complex queries by assembling information from several different systems implementing either the same or different worlds. The paradigm for this activity is a complex SQL query. In a single relational database, the database manager will decompose the query into its elements then automatically compose a query plan based on the kind of information we are describing, which is conventionally stored in the database manager's system catalogue. Where the data is stored in multiple autonomous databases, there is no central system catalogue. The proposal of this section functions as at least a partial substitute.

We may be concerned about the authoritativeness of an implementation of a world. Does the implementation store all the instances of a class, or only a fortuitous selection? Is the implementation the repository of the creators of the institutional facts stored in it, or are we seeing a copy? How often is the information updated?

To represent this sort of information, we need a system of predicates associated with the classes, associations and other structures in the ontology. We have already proposed annotations for meta-properties like identifying relations, unifying relations, rigid properties of classes, etc. However, unlike the meta-property annotations, the predicates described in this section are not necessary for the structural integrity of the world. We might call these *extent-descriptive* predicates.

Because extent-descriptive predicates are not structurally essential, it is probably best to refrain from specifying them in detail for a design language like UML. Each world or community of users will wish to design their own ontology of concepts to be used in extent-descriptive predicates. There may emerge packaged such ontologies which can be simply adopted. The UML design language needs only to have the capability for extent-descriptive predicates on all structures of the models represented. It should be possible for a structure to have several predicates from different extent-descriptive systems.

1.9 Players, roles and acts

Most of the business of an information system is to store institutional facts, that is the records of speech acts. As we have seen in the e-commerce and university examples above, in a multi-system, semantic web sort of environment, much of what goes on is

the making of speech acts, or at least the changes in the world of some of the systems as a result of changes in others, for example the change in the world of the lecturer as seen in a class list when a student makes the speech act of enrolling in the course.

Every fact is created, deleted or changed by an act. An act is performed by one or more entities who are authorized to take one of its possibly several roles. For example, a sale requires one entity to act as buyer and another as a seller. A marriage requires a bride, a groom and a celebrant. An entity authorized to take one role in one act may be authorized to take other roles in other acts. Not every entity in an ontology is able to take a role of any kind – most entities are passive.

We therefore propose a distinguished kind of entity called a *player*. A player can be a human, a corporation, a less formal organization, or possibly a computer program. After all a vending machine can take the seller role in a purchase speech act, and in the world of e-commerce there are a large number of speech acts performed by programs – consider a purchase by a personal agent of a book from Amazon.com mediated by a Visa credit card account.

A kind of fact is that a particular player is authorized to take a particular role in a particular kind of speech act. This fact is itself often created by a speech act – a marriage celebrant needs a licence.

The ontology needs a multi-adic class structure, recognizing players, roles, acts and facts as separate kinds of classes. With every fact class there must be associated act classes that create, delete or modify it. With every act class there must be associated role classes that determine who may perform it, generally together with a fact class giving the information required to perform the act (a purchase act requires a product and price, for example). With every player class there must be associated the role classes the player is authorized to take.

1.10 State

To support autonomous-agent-type applications, the ontology should have some sort of explicit representation of the state space relating to the various speech act types permitted, possibly as a finite state automaton. This is much less than OMG action semantics, since it does not necessarily prescribe how the transitions are achieved. It is also less than Microsoft's XLANG, because XLANG supports entire business processes. It may be covered by the proposed WMG WSDL, but published information is so far not detailed enough to tell. The developing area called choreography is probably where these modelling concepts lie.

As a first cut, a UML state diagram is probably adequate, where an event is identified as a class of speech act. State becomes an additional kind of class, augmenting the system of player, role, act and fact. A state transition can have a act class associated with it which is performed when the transition is executed after the recognition of the triggering act instance.

Using UML for Ontologies 2

This chapter outlines the extensions required to UML to support the representation of ontologies in Section 2.1 while Section 2.2 describes how to use existing UML constructs to support other ontology needs. Section 2.3 discusses the uses relating to the use of multiple ontologies.

2.1 Extensions to UML 2.0

To meet some of the requirements of ontologies, it is necessary to extend UML 2.0. As shown in Figure 1, the Ontology package extends the UML 2.0 Kernel package. The Ontology packages which contains a number of sub-packages that extend some of the sub-packages of UML 2.0 Kernel package.

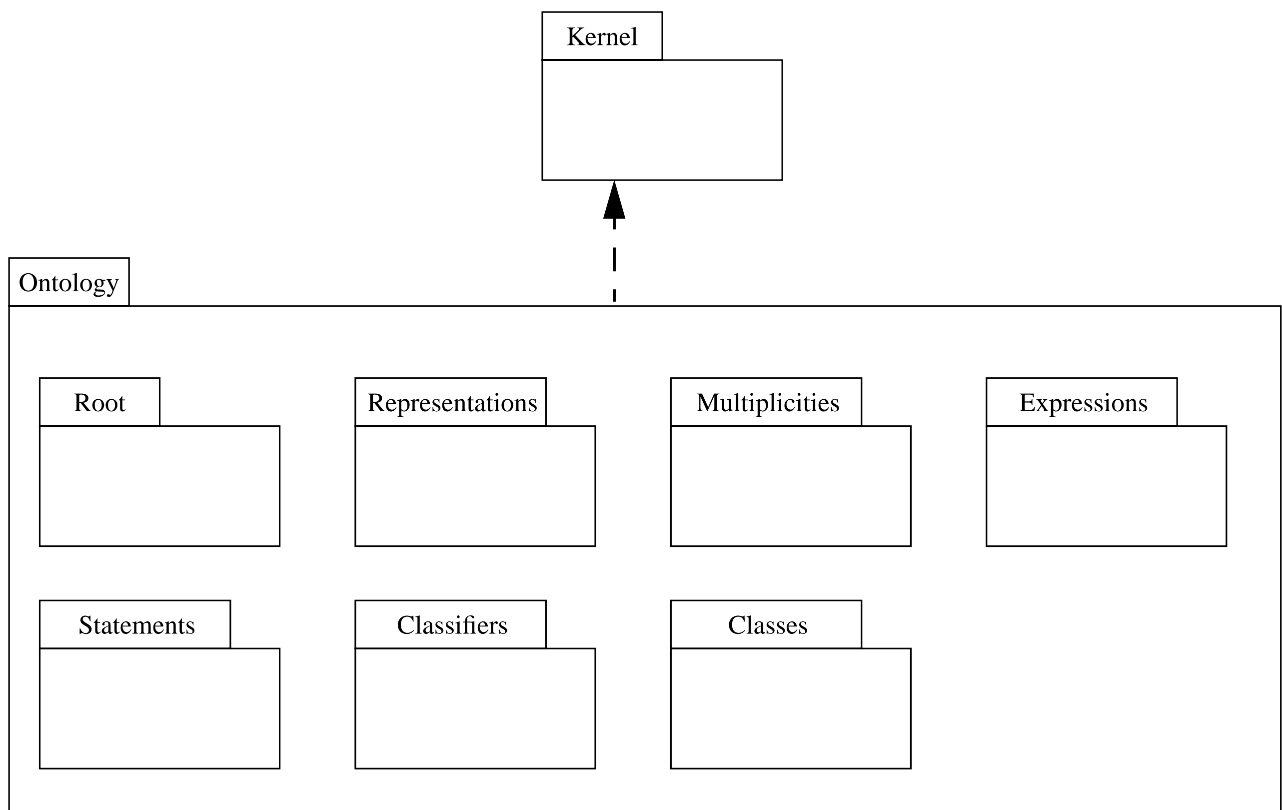

Figure 1 Ontology package and its sub-packages

2.1.1 Ontology::Root package

Figure 2 illustrates the Ontology::Root package as an the extension of the UML 2.0 Kernel::Root package.

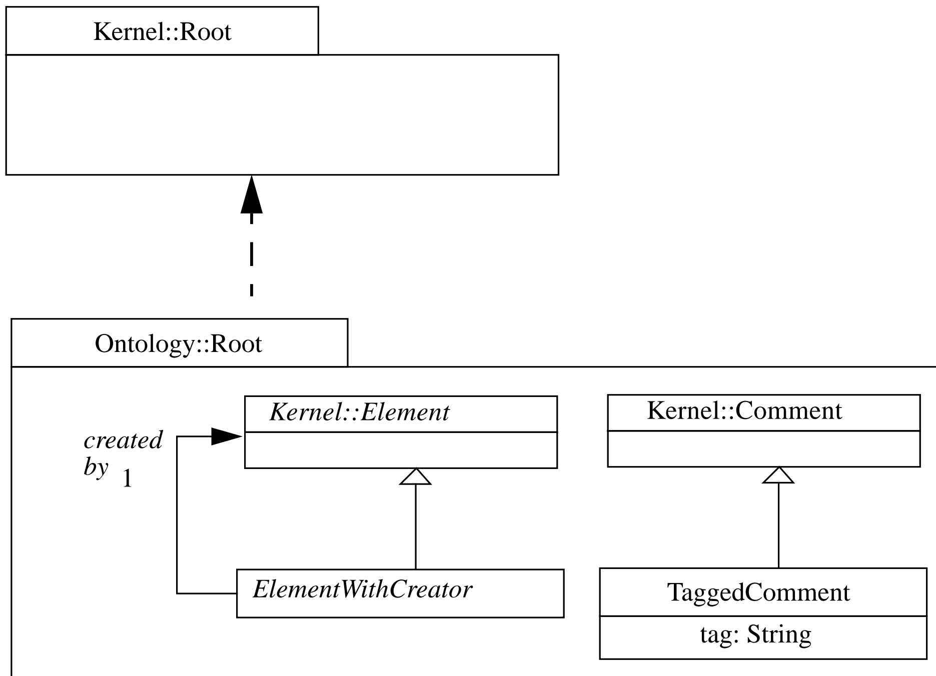

Figure 2 The Ontology::Root package

The abstract class ElementWithCreator is introduced to enable elements to be associated with a creator via the abstract association created_by. All statements and classifiers should have an associated creator.

The TaggedComment class enables comments on an element to have an associated tag. The provision of an uninterpreted "name-value" pair is extremely useful for attaching external information, and useful in transformation, especially for round-tripping purposes.

2.1.2 Ontology::Representations package

The Ontology::Representations package extends the Kernel::Namespaces package as illustrated in Figure 3.

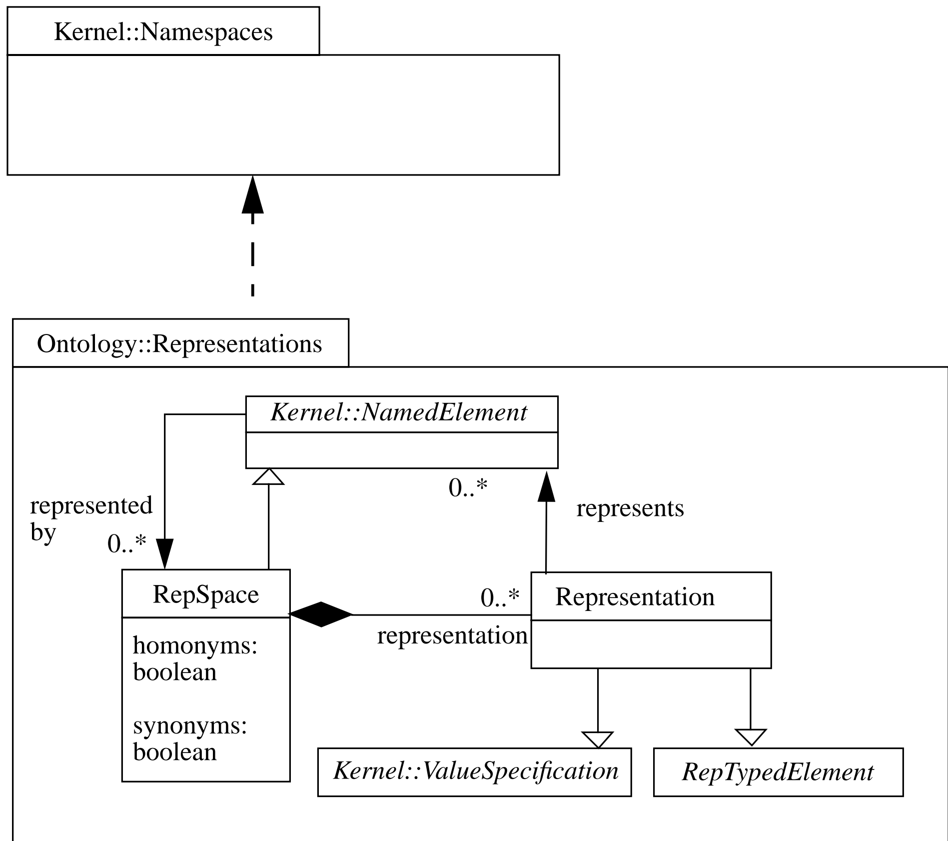

Figure 3 Ontology::Representations package

Elements in an ontology can have many representations (not restricted to strings as used in UML 2.0), both within a single representation space and across multiple representation spaces.

ValueSpecifications used as representations in a RepresentationSpace are constrained to be values which have a physical manifestation (e.g. text strings, images, line drawings, sounds). The representations used within a single RepresentationSpace are not constrained to be of the same type (although they often will be).

A represented element (a NamedElement) may or may not be aware of its representation(s). Those of which it is are aware as accessed via `represented_by`.

Since RepresentationSpaces can themselves be represented, it is possible to build complex or nested representations.

A RepresentationSpace may permit homonyms (the same name for multiple things) and synonyms (multiple names for the same thing). If homonyms and synonyms are not permitted, then the RepresentationSpace is a unique identification system.

The attribute `homonyms` is true if the RepresentationSpace allows homonyms (the same representation for multiple things). If homonyms are not permitted, then the RepresentationSpace is an identification system.

The attribute `synonyms` is true if the RepresentationSpace allows synonyms (multiple representations for the same thing). If homonyms and synonyms are not permitted, then the RepresentationSpace is an unique identification system.

2.1.3 Ontology::Multiplicities package

Figure 4 illustrates the Ontology::Multiplicities package, an extension of the UML 2.0 Kernel::Multiplicities package.

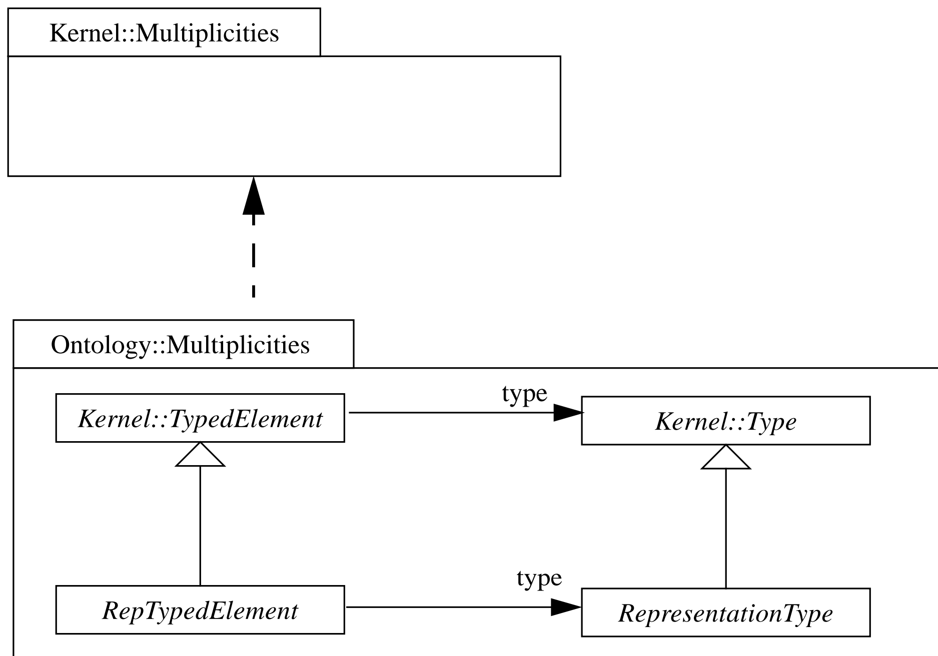

Figure 4 Ontology::Multiplicities package

RepresentationTypedElement and RepresentationType are abstract classes, which can be used to restrict types to only those classes which inherit from Ontology::Classes::RepresentationClass, i.e. those which have a physical manifestation and can be used for representation purposes (e.g. text strings, images, drawings, sounds).

2.1.4 Ontology::Expressions package

Figure 5 illustrates the Ontology::Expressions package, an extension of the UML 2.0 Kernel::Expressions package.

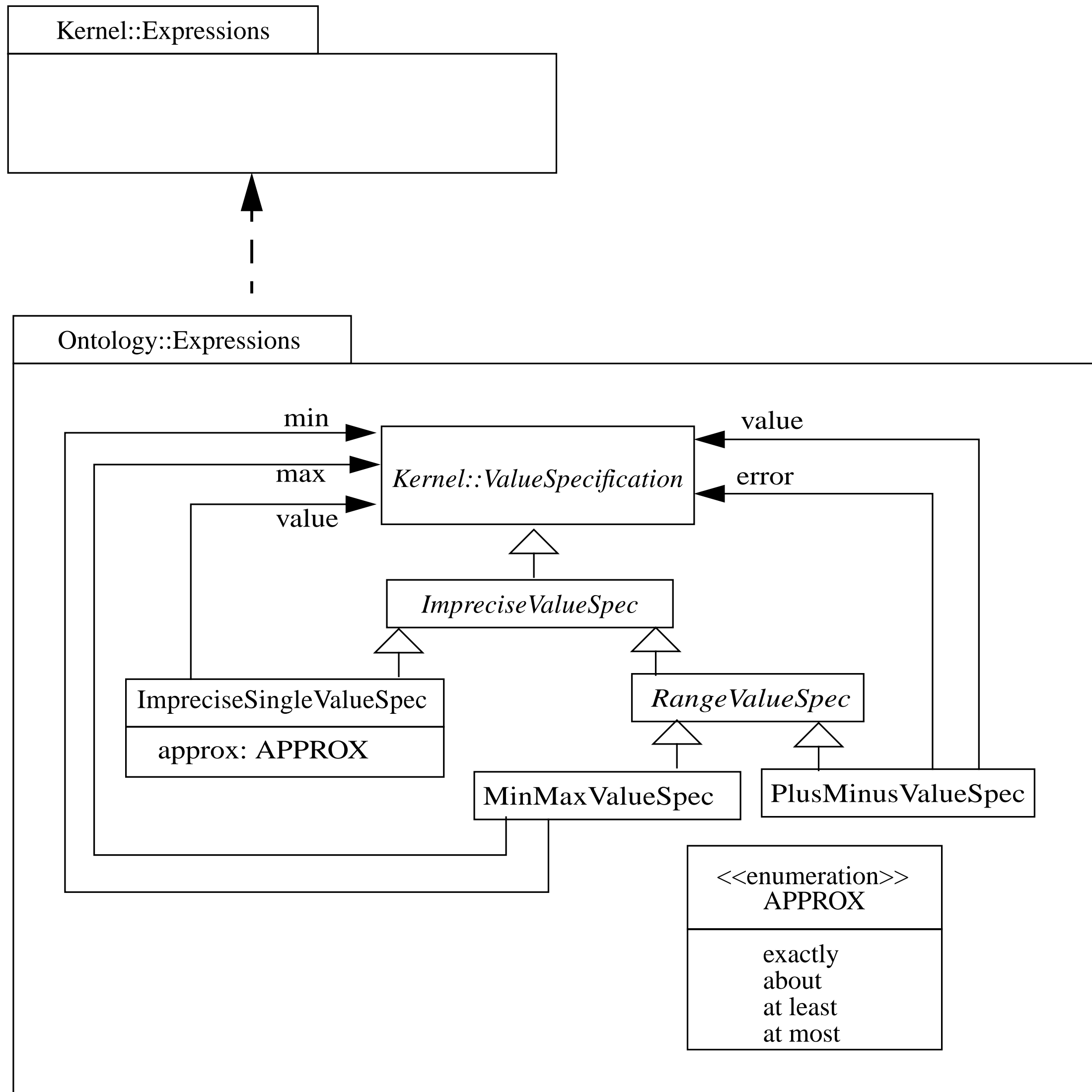

Figure 5 Ontology::Expressions package

The Ontology::Expressions package introduces the means to specify values imprecisely. It provides for 3 styles of imprecise values through the introduction of 3 concrete classes:

- ImpreciseSingleValueSpecification, which enables a value to be specified as being approximate using the decorations: “exactly”, “about”, “at least” or “at most”

- MinMaxValueSpecification, which enables a value to be specified as being within a range defined by a minimum and maximum value
- PlusMinusValueSpecification, which enables a value to be specified as being within a range defined by a base value, plus or minus an error value

2.1.5 Ontology::Statements package

Figure 6 illustrates the Ontology::Statements packages, an extension of the UML 2.0 Kernel::Instances package.

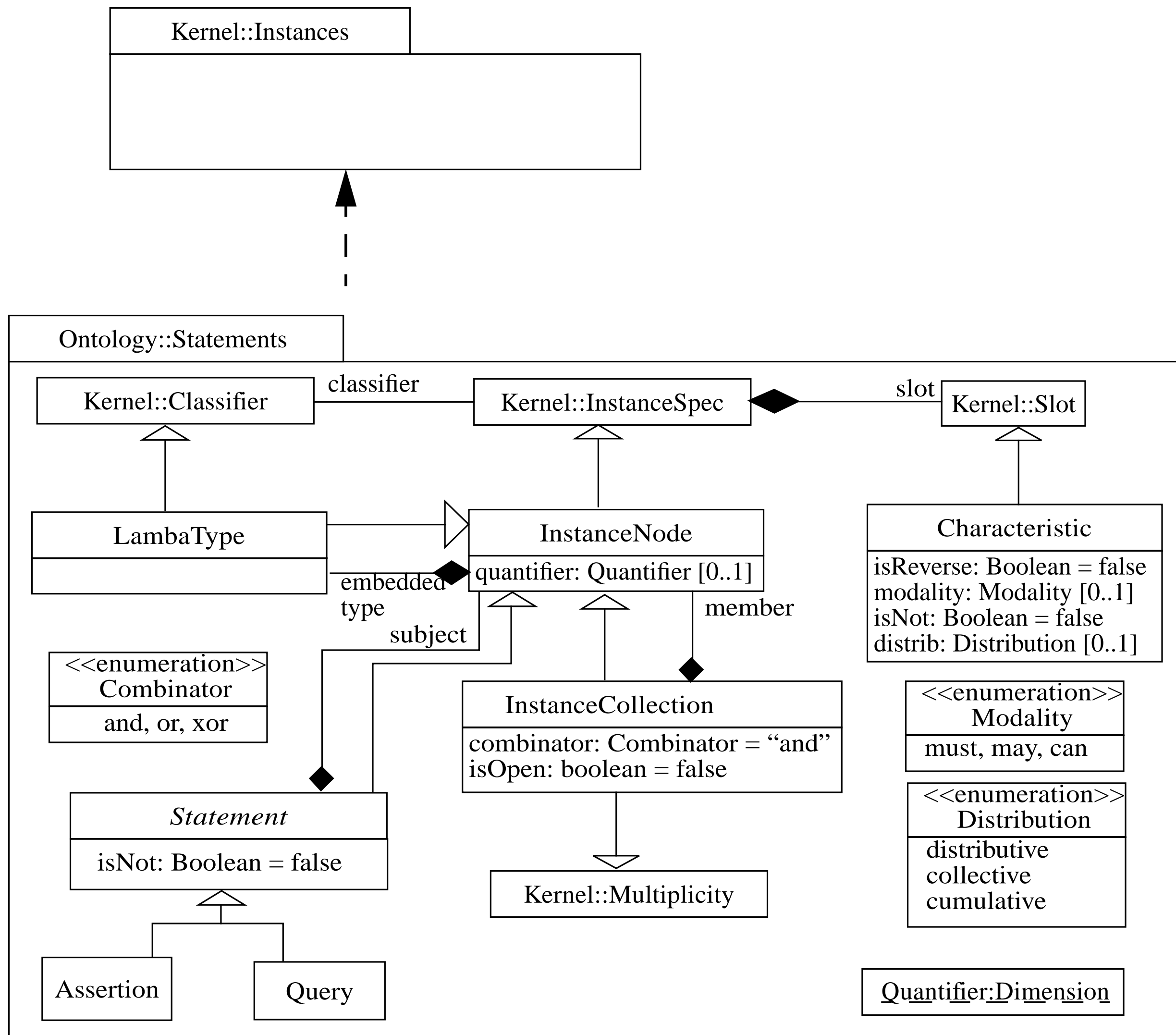

Figure 6 Ontology::Statements package

The Statements package extends UML’s instances to support the richer expression of knowledge.

The central construct is the InstanceNode which form graphs via the Characteristics.

The InstanceNode class inherits from UML's InstanceSpecification and provides the optional quantifier to indicate the extent of the population of the type to which the characteristics apply, e.g. "a", "some", "most". Quantifier is a class, which is itself an instance of the Ontology::Classes::Dimension class, which is used to define the extent of a population.

InstanceCollections specify a collection of InstanceNodes using the usual UML Multiplicity (with min, max, ordering, uniqueness criteria) with the additional properties:

- combinator, which indicates if the elements in the collection represent a union ("and") or strict alternatives ("xor") or multiple alternatives ("or"). For example, "Peter, Paul and Mary were a singing group" and "Smith, Jones xor Brown will be the next American President". The default is "and".
- isOpen, which if true indicates that the elements specified in the collection are incomplete and that other unspecified members may exist. This is similar to the UML notion of completeness in GeneralisationSets.

InstanceNodes contain Characteristics which inherit from UML's Slot and provide the following additional properties:

- The isReverse property indicates that definingFeature of the Characteristic (inherited from Slot) is not applicable to the Classifier of the owingInstance of the Slot, but is applicable to the Classifier of the Value of the Slot. For example, "The man Fred owns the dog Fido" could be also expressed in "reverse" as "The dog Fido is owned by the man Fred". Reversing characteristics is useful if the ontology does not define a relationship between two concepts in both directions.
- The optional modality property (must/may/can) introduces deontics (the strength of "binding" on the characteristic), e.g. "The man Fred can own the dog Fido" is different to "The man Fred owns the dog Fido".
- The isNegated property is true when the characteristic does not hold, e.g. "The man Fred does not own the dog Fido".
- The optional distrib property is used only for CollectionNodes (a collection of InstanceNodes) and describes whether the characteristic is:
  - distributive, meaning that it applies to the InstanceNode in the CollectionNode according to their combinator, e.g. "Fred and Mary own Fido" implies "Fred owns Fido and Mary owns Fido", while "Fred or Mary own Fido" implies "Fred owns Fido or Mary owns Fido".
  - collective, meaning that it applies to the collection as a whole and not (necessarily) to the elements, e.g. if Fred and Mary own Fido, then the ownership is joint and it cannot be said that either Fred owns Fido nor Mary owns Fido.
  - cumulative, meaning that it applies to the collection but is computed through the aggregation of some characteristic of the elements, e.g. Fred and Mary have an average age of 40 years through some computation involving the age of Fred and the age of Mary. Most statistical characteristics are cumulative.

InstanceNodes (being InstanceSpecifications) have a classifier, which may be a LambdaType which is a classifier defined within the scope of this InstanceNode. LambdaTypes are used to construct complex or specialised types which are unlikely to be sufficiently frequently used to justify their inclusion in an ontology, e.g. “people with red hair, blue eyes and freckles look good in green clothes” involves the lambda-type (class Person with a set of characteristics), which is expressed as an InstanceNode.

Note – How frequently a type is needed to justify its inclusion in the ontology is a subjective choice.

Statements are the abstract expression of knowledge with Assertion and Query being the imperative and interrogative forms. Each statement comprises a subject (an InstanceNode) with a least one characteristic required for well-formed-ness. For example, “Fred owns Fido” is an assertion, while the structurally similar “Does Fred own Fido?” is a query. However, “Fred” alone is not a statement. Statements themselves may be used as InstanceNodes, e.g. “The dogcatcher did not know that Fred owns Fido”. Statements can be expressed as a negation if the isNot property is true, e.g. “It is not true that Fred owns Fido”.

2.1.6 Extending Kernel::Classifiers

Figure 7 illustrates the Ontology::Classifiers package, an extension of the UML 2.0 Kernel::Classifiers package.

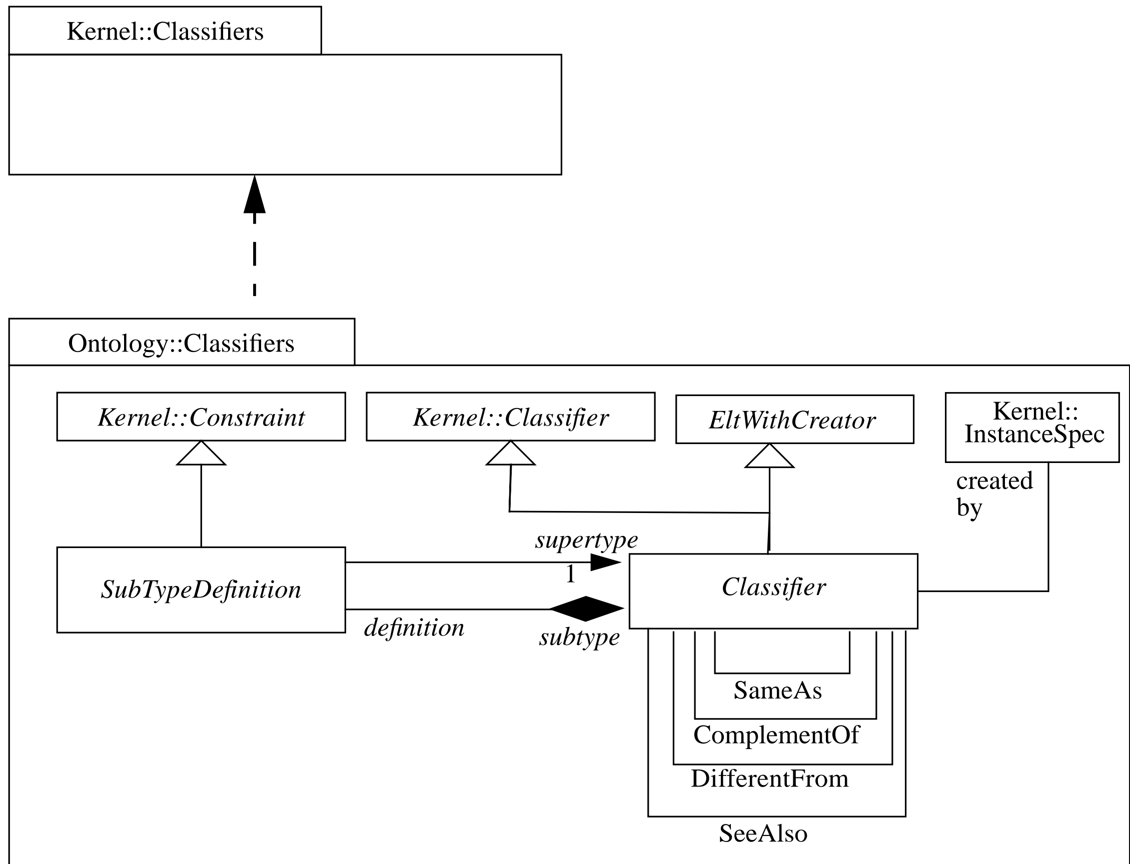

Figure 7 Ontology::Classifiers package

Types in an ontology can have subtypes, both by primitive subtyping (where one type inherits from another) or by defined subtyping (where membership of a subtype is defined by a predicate on the supertype, e.g. a rectangle (the supertype) whose length equals its width is a square. In UML, classes can explicitly inherit from other classes, yielding primitive subtyping, but UML 2.0 does not support defined subtyping.

The SubTypeDefinition (a Constraint) is applied to an instance of a “supertype” Classifier to determine if it is an instance of the subtype Classifier.

Ontologies typically represent a very large number of concepts and, in the presence of multiple ontologies, there may be the possibility of concepts being duplicated or confused with one another. The following associations are introduced to record such relationships among Classifiers.

The SameAs association is used to record duplicate Classifiers, i.e. representing the same concept. The classifiers involved may not be identical in their specification, and the concept is therefore defined by the union of their specifications. The SameAs association is transitive.

The ComplementOf association is used to record Classifiers which are the “negation” of one another, and hence every instance must be capable of being classified as being one or other but not both. Typically, one of the pair of complements will be defined as a concept in the “normal” way (by describing its properties), while its complement will be defined simply as being its “negation”. For example, a “SpatialElement” (meaning something that occupies volume in physical space) and “NotSpatialElement” (meaning something that does not occupy volume in physical space). Virtually every concept is capable of possessing a complement, but an ontology may not include complements that possess no additional properties (other than being the “negation” of another concept) as it is always possible in knowledge representation to “negate” a type “on the fly” through the use of lamba-types.

The DifferentFrom association is used to record Classifiers which, although similar in specification, are not representing the same concept and hence should not be confused with one another. Two classifiers cannot be both “same as” and “different from” one another. The DifferentFrom association is not transitive.

The SeeAlso association is used to record Classifiers, which represent similar concepts and which may be confused with one another. The SeeAlso association is transitive.

Note – Although DifferentFrom and SeeAlso are both attempting to avoid confusion by the ontology users, they are not derived from one another, as DifferentFrom is related to specification and SeeAlso is related to concepts. Both are “subjective” assessments, although tools may assist in identifying potential candidate pairs of Classifiers.

2.1.7 Ontology::Classes package

Figure 8 illustrates the Ontology::Classes package, an extension of the UML 2.0 Kernel::Classes package.

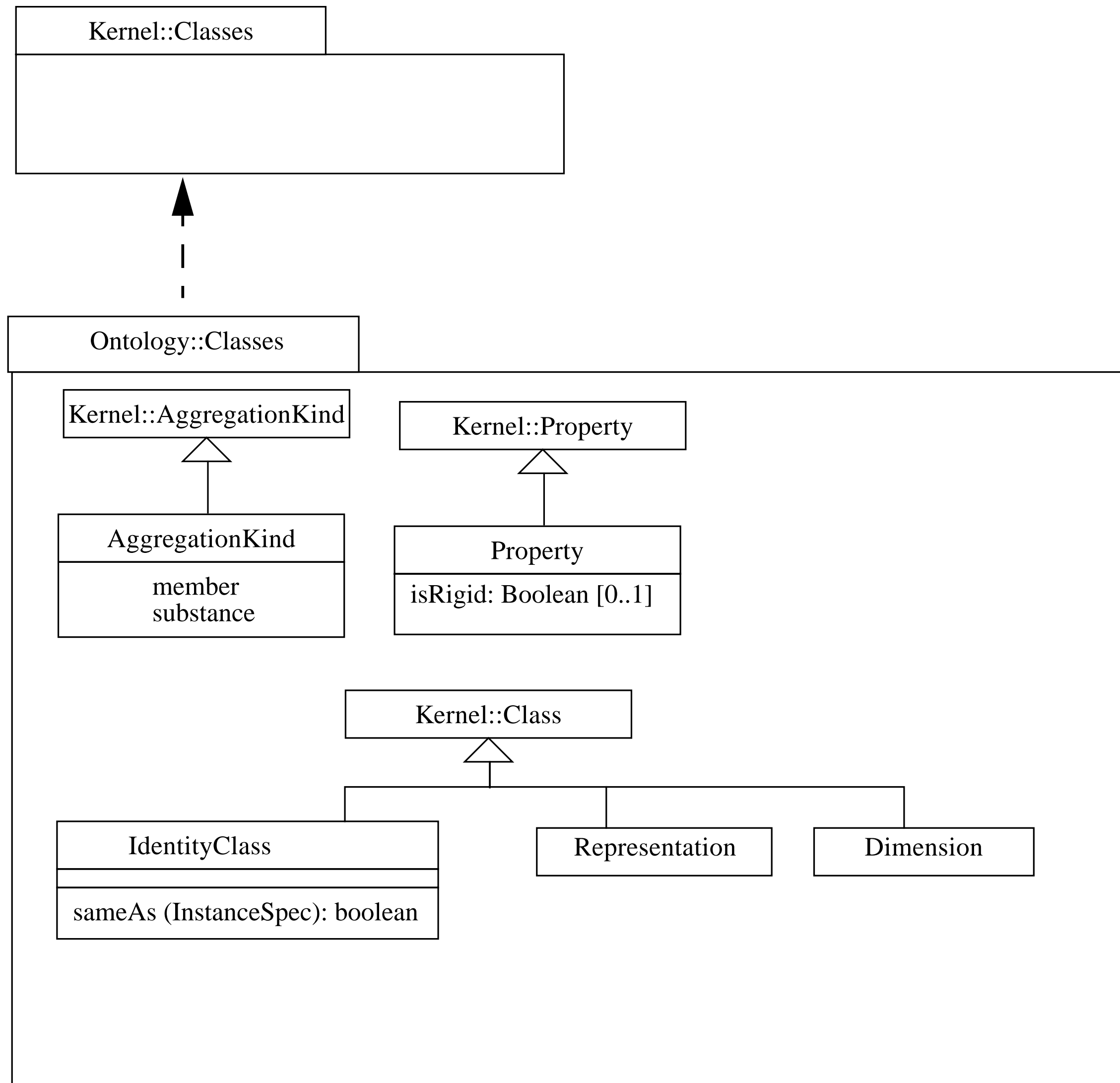

Figure 8 Ontology::Classes package

In ontologies [13], concepts can be classified by whether their instances have:

- identity (can two instances of the class always be distinguished?)
- unity (the ability of a whole to enumerate its parts)
- essentiality (properties which cannot change in their lifetime)
- rigidity (properties which cannot change and are shared by all instances of that concept)

For ontologies, we create the subclass IdentityClass for those classes which possess *identity*. IdentityClasses offer a method to test another instance for “same-ness” to itself.

AggregationKind is extended to support two new forms of aggregation. Set membership is denoted by “member” aggregation; it is not composite (no lifecycle semantics). Substance aggregation represents a form of composition in which the parts cannot be enumerated (do not possess *unity*), either because the parts do not possess identity in normal circumstances (e.g. an ocean is a composition of water, but we cannot identify individual “water”) or because the act of forming the composition transformed the parts in some manner which lost their identity (e.g. mashed potatoes). Classes which contain substances are known as “bulk classes”, as reference to their substances is generally in the form of quantitative statements (e.g. there are 5 eggs in that cake).

Essentiality is a property of an instance that cannot change, i.e. isReadOnly on class Property. *Rigidity* requires the same essential property of all instances of the class, and this is reflected by the isRigid property. Note that the isRigid property is not meaningful if isReadOnly is false.

Representation and Dimension are two new subtypes of Class, intended to the be superclass of any representation or dimension classes (respectively). It is important to note that the measurement of dimensions are handled using representations. For example, “3 feet” and “1 yard” and “approx .9 metres” are simply 3 different representations of the same instance of distance (a dimension).

2.2 Guidelines for use of UML

In ontologies, subtypes can have many supertypes. UML supports multiple inheritance.

Concepts in an ontology can be both types and instances. There appears to be no restriction in UML 2.0 on classes being instances of other classes. Classification systems can be built in this way as the specific class used for representation can itself be an instance of another (meta-) class pertaining to the classification system.

Partitioning. UML provides GeneralizationSets to enable Classifiers (and hence both Classes and Associations) to be partitioned. There is support for the notions of complete/incomplete partitions and for partitions to be disjoint (or not).

Attributes is a light-weight rendering of associations, and imply that the class that is the target (type) of the attribute is less interesting than the class that owns the attribute. While this is a valid approach when modelling an abstraction, it is not an ontological distinction. UML permits attributes, but for ontologies, the use of attributes (other than in the representation classes) is discouraged.

Support for players, roles, acts (events), facts, states can be found in UML 2.0 Superstructure and UML Enterprise Collaboration Architecture.

2.3 Supporting multiple ontologies

In enterprise distributed systems, it is inevitable that there will be many ontologies and knowledge bases available for different domains, or even the same domain. Therefore, the interworking of multiple ontologies and knowledge bases must be supported.

The use of a "starter ontology" (see Appendix A for an example) is encouraged to provide a basis for positioning one ontology related to another in broad terms. The use of associations SameAs, DifferentTo, SeeAlso etc. can be used to further assert relationships between multiple ontologies.

Divergent beliefs are possible even within a single knowledge base and almost inevitable when interworking multiple knowledge bases. In a revision submission, additional model elements will be provided to provide finer-grained control over the relationships between concepts in separate ontologies (e.g. "refinement", "overlaps") and well as relationships between assertions in knowledge bases (e.g. "supports", "is an example of", "contradicts", "is a counter-example of"). These relationships will then be used to provide the building blocks for a policy model to enable inferencing to occur in the presence of divergent beliefs and inconsistencies across multiple knowledge bases. It is anticipated that the policy model will enable the user to express their willingness to believe, prefer or ignore assertions from nominated sources over nominated subject areas, which captures the notion of some sources being "more expert" in some subject areas than others.

Relationship of OWL and UML 3

This Chapter is a commentary on the relationship between OWL and UML 2.0.

Material in plain text is from the OWL source document. Material in italics is commentary.

3.1 OWL Lite RDF Schema Features

The following OWL Lite features related to RDF Schema are included.

Class: A class defines a group of individuals that belong together because they share some properties. For example, Deborah and Frank are both members of the class Person. Classes can be organized in a specialization hierarchy using SubClassOf. There is a built-in most general class named Thing that is the class of all individuals and a superclass of all OWL classes.

> *Same as UML class.*

rdfs:subClassOf: Class hierarchies may be created by making one or more statements that a class is a subclass of another class. For example, the class Person could be stated to be a subclass of the class Mammal. From this a reasoner can deduce that if an individual is a Person, then it is a Mammal.

> *Same as UML subclass. Probably includes asserted as well as defined subclass relationships.*

rdfs:Property: Properties can be used to state relationships between individuals or from individuals to data values. Examples of properties include hasChild, hasRelative, hasSibling, and hasAge. The first three can be used to relate an instance of a class Person to another instance of the class Person (and are thus **ObjectProperties**), and the last (hasAge) can be used to relate an instance of the class Person to an instance of the datatype Integer (and is thus a **Datatype** property).

> *Conflation of UML association and attribute. ObjectProperty seems to be roughly association, while Datatype property seems to be roughly attribute. Except that a property can apply to any class, while in UML an attribute or association is connected to specified classes. Attributes can be multi-valued. More like the ORM (NIAM) concept of a fact type, but detached from any object types. All properties seem to be binary, but all n-ary associations can be made into binary associations using derived associative classes.*

rdfs:subPropertyOf: Property hierarchies may be created by making one or more statements that a property is a subproperty of one or more other properties. For example, hasSibling may be stated to be a subproperty of hasRelative. From this a reasoner can deduce that if an individual is related to another by the hasSibling property, then it is also related to the other by the hasRelative property.

> *Associations can have sub-associations. Value sets of attributes can have a subset structure. The subtypes can have a lattice structure. Presumably applies also to rdfs:subClassOf.*

rdfs:domain: A domain of a property limits the individuals to which the property can be applied. If a property relates individual to another individual, and the property has a class as one of its domains, then the individual must belong to the class. For example, the property hasChild may be stated to have the domain of Mammal. From this a reasoner can deduce that if Frank hasChild Anna, then Frank must be a Mammal. Note that rdfs:domain is called a global restriction since the restriction is stated on the property and not just on the property when it is associated with a particular class. See the discussion below on local restrictions for more information.

> *Brings property closer to association/ attribute. Can still be attached to multiple classes, but restricted to a particular set of subtypes. Note that OWL has a class lattice top "thing".*

rdfs:range: The range of a property limits the individuals that the property may have as its value. If a property relates an individual to another individual, and the property has a class as its range, then the other individual must belong to the range class. For example, the property hasChild may be stated to have the range of Mammal. From this a reasoner can deduce that if Louise is related to Deborah by the hasChild property, i.e., Deborah is the child of Louise, then Deborah is a Mammal. Range is also a global restriction as is domain above. Again, see the discussion below on local restrictions (e.g. AllValuesFrom) for more information.

> *Seems that property is thought of as directional, rather than as a relation. The two (or more) ends of an association are symmetrical, while an attribute is directional.*

Individual: Individuals are instances of classes, and properties may be used to relate one individual to another. For example, an individual named Deborah may be described as an instance of the class Person and the property hasEmployer may be used to relate the individual Deborah to the individual StanfordUniversity.

> *Instances of classes or associations or class-instance/attribute instance couples.*

3.2 OWL Lite Equality and Inequality

The following OWL Lite features are related to equality or inequality.

equivalentClass: Two classes may be stated to be equivalent. Equivalent classes have the same instances. Equality can be used to create synonymous classes. For example, Car can be stated to be equivalentClass to Automobile. From this a reasoner can deduce that any individual that is an instance of Car is also an instance of Automobile and vice versa.

Synonyms of class names.

equivalentProperty: Two properties may be stated to be equivalent. Equivalent properties relate one individual to the same set of other individuals. Equality may be used to create synonymous properties. For example, hasLeader may be stated to be the equivalentProperty to hasHead. From this a reasoner can deduce that if X is related to Y by the property hasLeader, X is also related to Y by the property hasHead and vice versa. A reasoner can also deduce that hasLeader is a subproperty of hasHead and hasHead is a subProperty of hasLeader.

Synonyms of association or attribute names.

sameIndividualAs: Two individuals may be stated to be the same. This construct may be used to create a number of different names that refer to the same individual. For example, the individual Deborah may be stated to be the same individual as DeborahMcGuinness.

The unique name assumption does not apply. So need token-level synonyms.

differentFrom: An individual may be stated to be different from other individuals. For example, the individual Frank may be stated to be different from the individuals Deborah and Jim. Thus, if the individuals Frank and Deborah are both values for a property that is stated to be functional (thus the property has at most one value), then there is a contradiction. Explicitly stating that individuals are different can be important in when using languages such as OWL (and RDF) that do not assume that individuals have one and only one name. For example, with no additional information, a reasoner will not deduce that Frank and Deborah refer to distinct individuals.

The unique name assumption does not apply. So need explicitly defined difference.

allDifferent: A number of individuals may be stated to be mutually distinct in one allDifferent statement. For example, Frank, Deborah, and Jim could be stated to be mutually distinct using the allDifferent construct. Unlike the differentFrom statement above, this would also enforce that Jim and Deborah are distinct (not just that Frank is distinct from Deborah and Frank is distinct from Jim). The allDifferent construct is particularly useful when there are sets of distinct objects and when modelers are interested in enforcing the unique names assumption within those sets of objects.

Makes the unique name assumption among a set of names.

3.3 OWL Lite Property Characteristics

There are special identifiers in OWL Lite that are used to provide information concerning properties and their values.

inverseOf: One property may be stated to be the inverse of another property. If the property P1 is stated to be the inverse of the property P2, then if X is related to Y by the P2 property, then Y is related to X by the P1 property. For example, if hasChild is the inverse of hasParent and Deborah hasParent Louise, then a reasoner can deduce that Louise hasChild Deborah.

> *Inverse association. Also inverse of attribute function. Note that attributes can be multi-valued. Note also that several classes may have the same attribute function.*

TransitiveProperty: Properties may be stated to be transitive. If a property is transitive, then if the pair (x,y) is an instance of the transitive property P, and the pair (y,z) is an instance of P, then the pair (x,z) is also an instance of P. For example, if ancestor is stated to be transitive, and if Sara is an ancestor of Louise (i.e., (Sara,Louise) is an instance of the property ancestor) and Louise is an ancestor of Deborah (i.e., (Louise,Deborah) is an instance of the property ancestor), then a reasoner can deduce that Sara is an ancestor of Deborah (i.e., (Sara,Deborah) is an instance of the property ancestor). OWL Lite (and OWL DL) impose the side condition that transitive properties (and their superproperties) cannot have a maxCardinality 1 restriction. Without this side-condition, OWL Lite and OWL DL would become undecidable languages. See the property axiom section of the OWL Abstract Syntax and Semantics document for more information.

> *Associations can be transitive (presumably they must have type-compatible domain and range). A generalization of the concept of homogeneous fact type in ORM or a homogeneous relationship in ERA.*

SymmetricProperty: Properties may be stated to be symmetric. If a property is symmetric, then if the pair (x,y) is an instance of the symmetric property P, then the pair (y,x) is also an instance of P. For example, friend may be stated to be a symmetric property. Then a reasoner that is given that Frank is a friend of Deborah can deduce that Deborah is a friend of Frank. Note that properties that are to be made symmetric may not have arbitrary domains and ranges.

> *Associations can be symmetric (presumably have the same domain and range).*

FunctionalProperty: Properties may be stated to have a unique value. If a property is a FunctionalProperty, then it has no more than one value for each individual (it may have no values for an individual). This characteristic has been referred to as having a unique property. FunctionalProperty is shorthand for stating that the property's minimum cardinality is zero and its maximum cardinality is 1. For example, hasPrimaryEmployer may be stated to be a FunctionalProperty. From this a reasoner may deduce that no individual may have more than one primary employer. This does not imply that every Person must have at least one primary employer however.

> *Particular UML cardinality constraint. One end of (binary) association is [0 - 1]. A single-valued attribute can be optional.*

InverseFunctionalProperty: Properties may be stated to be inverse functional. If a property is inverse functional then the inverse of the property is functional. Thus the inverse of the property has at most one value for each individual. This characteristic

has also been referred to as an unambiguous property. For example, hasUSSocialSecurityNumber (a unique identifier for United States residents) may be stated to be inverse functional (or unambiguous). The inverse of this property (which may be referred to as isTheSocialSecurityNumberFor) has at most one value for any individual in the class of social security numbers. Thus any one person's social security number is the only value for their isTheSocialSecurityNumberfor property. From this a reasoner can deduce that no two different individual instances of Person have the identical US Social Security Number. Also, a reasoner can deduce that if two instances of Person have the same social security number, then those two instances refer to the same individual.

Inverse of a functional association or a single-valued attribute.

3.4 OWL Lite Property Type Restriction

OWL Lite allows restrictions to be placed on how properties can be used by instances of a class. The following two restrictions limit which values can be used while the next section's restrictions limit how many values can be used.

allValuesFrom: The restriction allValuesFrom is stated on a property with respect to a class. It means that this property on this particular class has a local range restriction associated with it. Thus if an instance of the class is related by the property to a second individual, then the second individual can be inferred to be an instance of the local range restriction class. For example, the class Person may have a property called hasOffspring restricted to have allValuesFrom the class Person. This means that if an individual person Louise is related by the property hasOffspring to the individual Deborah, then from this a reasoner can deduce that Deborah is an instance of the class Person. This restriction allows the property hasOffspring to be used with other classes, such as the class Cat, and have an appropriate value restriction associated with the use of the property on that class. In this case, hasOffspring would have the local range restriction of Cat when associated with the class Cat and would have the local range restriction Person when associated with the class Person. Note that a reasoner can not deduce from an allValuesFrom restriction alone that there actually is at least one value for the property.

Range applies to a property detached from any particular domain (domain "thing"). This construct is the same as range but applied to a property only when it is attached to a particular class, or one of its subclasses.

someValuesFrom: The restriction someValuesFrom is stated on a property with respect to a class. A particular class may have a restriction on a property that at least one value for that property is of a certain type. For example, the class SemanticWebPaper may have a someValuesFrom restriction on the hasKeyword property that states that some value for the hasKeyword property should be an instance of the class SemanticWebTopic. This allows for the option of having multiple keywords and as long as one or more is an instance of the class SemanticWebTopic, then the paper would be consistent with the someValuesFrom restriction. Unlike allValuesFrom, someValuesFrom does not restrict all the values of the property to be instances of the same class. If myPaper is an instance of the SemanticWebPaper class, then myPaper is related by the hasKeyword property to at least one instance of the

SemanticWebTopic class. Note that a reasoner can not deduce (as it could with allValuesFrom restrictions) that all values of hasKeyword are instances of the SemanticWebTopic class

The range of a property attached to a particular class must intersect with a nominated class. (The previous construct allValuesFrom requires the range to be a subset of a nominated class.)

3.5 OWL Lite Restricted Cardinality

OWL Lite includes a limited form of cardinality restrictions. OWL (and OWL Lite) cardinality restrictions are referred to as local restrictions, since they are stated on properties with respect to a particular class. That is, the restrictions constrain the cardinality of that property on instances of that class. OWL Lite cardinality restrictions are limited because they only allow statements concerning cardinalities of value 0 or 1 (they do not allow arbitrary values for cardinality, as is the case in OWL DL and OWL Full).

minCardinality: Cardinality is stated on a property with respect to a particular class. If a minCardinality of 1 is stated on a property with respect to a class, then any instance of that class will be related to at least one individual by that property. This restriction is another way of saying that the property is required to have a value for all instances of the class. For example, the class Person would not have any minimum cardinality restrictions stated on a hasOffspring property since not all persons have offspring. The class Parent, however would have a minimum cardinality of 1 on the hasOffspring property. If a reasoner knows that Louise is a Person, then nothing can be deduced about a minimum cardinality for her hasOffspring property. Once it is discovered that Louise is an instance of Parent, then a reasoner can deduce that Louise is related to at least one individual by the hasOffspring property. From this information alone, a reasoner can not deduce any maximum number of offspring for individual instances of the class parent. In OWL Lite the only minimum cardinalities allowed are 0 or 1. A minimum cardinality of zero on a property just states (in the absence of any more specific information) that the property is optional with respect to a class. For example, the property has Offspring may have a minimum cardinality of zero on the class Person (while it is stated to have the more specific information of minimum cardinality of one on the class Parent).

UML cardinality [1] or [0 - 1]

maxCardinality: Cardinality is stated on a property with respect to a particular class. If a maxCardinality of 1 is stated on a property with respect to a class, then any instance of that class will be related to at most one individual by that property. A maxCardinality 1 restriction is sometimes called a functional or unique property. For example, the property hasRegisteredVotingState on the class UnitedStatesCitizens may have a maximum cardinality of one (because people are only allowed to vote in only one state). From this a reasoner can deduce that individual instances of the class USCitizens may not be related to two or more distinct individuals through the hasRegisteredVotingState property. From a maximum cardinality one restriction alone, a reasoner can not deduce a minimum cardinality of 1. It may be useful to state that certain classes have no values for a particular property. For example, instances of the

class UnmarriedPerson should not be related to any individuals by the property hasSpouse. This situation is represented by a maximum cardinality of zero on the hasSpouse property on the class UnmarriedPerson.

> *UML cardinality [0-1] or stipulation that an association does not apply to a particular class. Remember that properties are defined without being attached to classes.*

cardinality: Cardinality is provided as a convenience when it is useful to state that a property on a class has both minCardinality 0 and maxCardinality 0 or both minCardinality 1 and maxCardinality 1. For example, the class Person has exactly one value for the property hasBirthMother. From this a reasoner can deduce that no two distinct individual instances of the class Mother may be values for the hasBirthMother property of the same person. Alternate namings for these restricted forms of cardinality were discussed. Current recommendations are to include any such names in a front end system. More on this topic is available on the publicly available webont mail archives with the most relevant message at http://lists.w3.org/Archives/Public/www-webont-wg/2002Oct/0063.html.

> *In ERA modelling terms the association is either both mandatory and functional or it is forbidden. (One presumably represents surjective functions by applying these cardinality constraints to the inverse.)*

3.6 OWL Lite Class Intersection

OWL Lite has contains an intersection constructor but limits its usage.

intersectionOf: OWL Lite allows intersections of named classes and restrictions. For example, the class EmployedPerson can be described as the intersectionOf Person and EmployedThings (which could be defined as things that have a minimum cardinality of 1 on the hasEmployer property). From this a reasoner may deduce that any particular EmployedPerson has at least one employer.

A subclass can be defined as the intersection of classes.

3.7 Incremental Language Description of OWL DL and OWL FULL

Both OWL DL and OWL Full use the same vocabulary although OWL DL is subject to some restrictions. Roughly, OWL DL requires type separation (a class can not also be an individual or property, a property can not also be an individual or class). This implies that restrictions cannot be applied to the language elements of OWL itself (something that is allowed in OWL Full). Furthermore, OWL DL requires that properties are either ObjectProperties or DatatypeProperties: DatatypeProperties are relations between instances of classes and RDF literals and XML Schema datatypes, while ObjectProperties are relations between instances of two classes. The OWL Abstract Syntax and Semantics document explains the distinctions and limitations. We describe the OWL DL and OWL Full vocabulary that extends the constructions of OWL Lite below.

oneOf: (enumerated classes): Classes can be described by enumeration of the individuals that make up the class. The members of the class are exactly the set of enumerated individuals; no more, no less. For example, the class of daysOfTheWeek can be described by simply enumerating the individuals Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday. From this a reasoner can deduce the maximum cardinality (7) of any property that has daysOfTheWeek as its allValuesFrom restriction.

> *Same as UML enumerated type.*

hasValue: (property values): A property can be required to have a certain individual as a value (also sometimes referred to as property values). For example, instances of the class of dutchCitizens can be characterized as those people that have theNetherlands as a value of their nationality. (TheNetherlands itself is an instance of the class of Nationalities).

> *Constraint on a property that any class it applies to must contain a particular object. (Properties can be defined between "thing" and "thing".)*

disjointWith: OWL Full allows the statement that classes are disjoint. For example, Man and Woman can be stated to be disjoint classes. From this disjointWith statement, a reasoner can deduce an inconsistency when an individual is stated to be an instance of both and similarly a reasoner can deduce that if A is an instance of Man, then A is not an instance of Woman.

> *Disjoint subclasses.*

unionOf, complementOf, intersectionOf (Boolean combinations): OWL allows arbitrary Boolean combinations of classes and restrictions: unionOf, complementOf, and intersectionOf. For example, using unionOf, we can state that a class contains things that are either USCitizens or DutchCitizens. Using complementOf, we could state that children are not SeniorCitizens. (i.e. the class Children is a subclass of the complement of SeniorCitizens). Citizenship of the European Union could be described as the union of the citizenship of all member states.

> *Classes defined by boolean combination of subclasses (either applied to asserted classes or to the restriction predicates of defined classes).*

minCardinality, maxCardinality, cardinality (full cardinality): While in OWL Lite, cardinalities are restricted to at least, at most or exactly 1 or 0, full OWL allows cardinality statements for arbitrary non-negative integers. For example the class of DINKs ("Dual Income, No Kids") would restrict the cardinality of the property hasIncome to a minimum cardinality of two (while the property hasChild would have be restricted to cardinality 0).

> *Cardinality constraints as found in UML.*

complex classes: In many constructs, OWL Lite restricts the syntax to single class names (e.g. in subClassOf or equivalentClass statements). OWL Full extends this restriction to allow arbitrarily complex class descriptions, consisting of enumerated classes, property restrictions, and Boolean combinations. OWL also includes a special

“bottom” class with the name Nothing that is the class that has no instances. Also, OWL full allows classes to be used as instances (and OWL DL and OWL Lite do not). For more on this topic, see the “Design for Use” section of the Guide document.

Not entirely clear the implications of this. If OWL Lite allows defined classes, than the first extension simply says that one can replace the name by the class' definition predicate. The 'also' is at least 1.5 order. Note that OWL does not appear to include quantification at all.

Conformance 4

There are 6 levels of conformance defined by this specification:

- Single Ontology Conformance (Section 4.1)
- Single Knowledge Base Conformance (Section 4.2)
- Multiple Ontology Conformance (Section 4.3)
- Multiple Knowledge Base Conformance (Section 4.4)
- OWL DL Source-Mapping Conformance (Section 4.5)
- OWL DL Target-Mapping Conformance (Section 4.6)

4.1 Single Ontology Conformance

Single ontology conformance enables the definition of concepts, individuals and relationships within a single ontology.

Note – The set of classes and associations required for this conformance will be given in a revised submission.

4.2 Single Knowledge Base Conformance

Single knowledge base conformance enables the expression of statements (assertions and queries) within a single knowledge base defined by a single ontology. Single knowledge base conformance is a superset of single ontology conformance.

Note – The set of classes and associations required for this conformance will be given in a revised submission.

4.3 Multiple Ontology Conformance

Multiple ontology conformance enables the interworking of multiple ontologies. Multiple ontology conformance is a superset of single ontology conformance.

Note – The set of classes and associations required for this conformance will be given in a revised submission.

4.4 Multiple Knowledge Base Conformance

Multiple knowledge base conformance enables the interworking of multiple knowledge bases, defined by multiple ontologies, including the ability to detect and resolve inconsistency and divergent beliefs. Multiple knowledge base conformance is a superset of multiple ontology conformance and single knowledge base conformance.

Note – The set of classes and associations required for this conformance will be given in a revised submission.

4.5 OWL DL Source-Mapping Conformance

OWL DL Source conformance enables the translation of OWL DL into ODM.

Note – The mappings required for this conformance will be given in a revised submission.

4.6 OWL DL Target-Mapping Conformance

OWL DL Target conformance enables the translation of ODM into OWL DL.

Note – The mappings required for this conformance will be given in a revised submission.

References 5

[1] Object Management Group, *MDA Guide,* omg/03-06-01.

[2] Gruber, T.R., *Towards Principles for the Design of Ontologies used for Knowledge Sharing,* Technical Report KSL-93-04 Knowledge Systems Laboratory, Stanford University, 1993.

[3] Weber, R., O*ntological Foundations of Information Systems,* Coopers & Lybrand Accounting Research Methodology, Monograph No 4, Melbourne, Australia, 1997.

[4] *www.snomed.org*

[5] W.V.O. Quine, *Word and object,* MIT Press 1960.

[6] L. Wittgenstein, *Tractatus logico-philosophicus,* Routledge and Keegan Paul, 1974 sect 4.121.

[7] J. Searle, *The construction of social reality,* The Free Press 1995.

[8] *www.geneontology.org*

[9] Melton, J and Simon, A.R., *SQL:1999,* Academic Press, 2002.

[10] *www.gils.net/semantics.html*

[11] Colomb, R., *Information Spaces*, Springer 2002, p. 142.

[12] Nijssen, S. and Halpin, T., *Conceptual Schema and Relational Database Design*, Prentice-Hall 1989.

[13] Guarino, N. & Welty C., *Evaluation Ontological Decisions with Ontoclean,* CACM 45(2) pp 61-65, 2002.

References

Starter Ontology — Appendix A

A.1 Introduction

This intitial submission proposes that a starter ontology should be adopted as part of this ODM RPF. The purpose of the starter ontology is to provide a basic core of concepts, which would serve as a basis for other ontologies.

This appendix represents an example of a starter ontology, drawn from DSTC's WebKB2 knowledge representation system. The concrete syntax used in this initial submission is that of WebKB2 and is not proposed as a normative concrete syntax for ODM.

A.2 WebKB2 notation

The WebKB2 notation uses the following symbols:

Table 5-1 WebKB2 notation

Symbol	Meaning
<	subtype of
>	subtype
^	instance of
:	instance
~	exclusion
=	similar
P	part of
p	part
M	member of
m	member
S	substance of
s	substance
N	noun type of

Table 5-1 WebKB2 notation

Symbol	Meaning
n	noun type
O	object of
o	object

A.3 Starter Ontology

```
Thing (^anything that is not a relation^)
 > {Situation Entity} Thing_playing_some_role,
 ~ relation;

   Situation (^thing that "occurs" in a (real or imagined) region of time and space^)
    > {State Process} Phenomenon event Situation_playing_some_role;

       State (^situation not changing and not making a change during a given period of time^)
        > state;

       Process (^situation that makes a change during some period of time^)
        > Event Problem_solving_process act Process_playing_a_role;

           Event (^process considered instantaneous from some viewpoint; classification under this
                            category is application-dependant^);

           Problem_solving_process (^cognitive activity to solve a problem^)
            > Task;

               Task (^processes modelled in knowledge acquisition, e.g. a diagnostic^)
                > Real_life_task;

                   Real_life_task (^a task composed of more basic tasks^)
                    > Knowledge_engineering;

                       Knowledge_engineering (^making a knowledge based system^)
                        > {Environment_analysis Problem_analysis Task_analysis Function_analysis
                            Implementation_analysis} Knowledge_engineering_with_KADS;

                           Knowledge_engineering_with_KADS (^making a knowledge based system using KADS
                            methodology^);

       Phenomenon (^situation known through the senses rather than by reasoning^)
        > phenomenon;

       Situation_playing_some_role (^e.g. a causal situation^)
        > Process_playing_a_role consequence result;

   Entity (^thing that can be "involved" in a situation^)
    > {Spatial_entity Nonspatial_entity} {Undivisible_entity Divisible_entity} Entity_playing_some_role;

       Spatial_entity (^space region or thing occupying a space region^)
        > Space Physical_entity;

           Space (^point or extent in space^)
            > space;

           Physical_entity (^spatial entity made of matter^)
            > {Entity_that_can_be_alive Entity_that_cannot_be_alive} Dead_entity;

               Entity_that_can_be_alive (^e.g. an animal, a cell^)
                > Living_entity life_form cell;

                   Living_entity (^entity that is alive^)
                    ~ Dead_entity;

               Entity_that_cannot_be_alive (^e.g. a bottle^)
                > object;

               Dead_entity (^entity that is no more alive^)
                ~ Living_entity;

       Nonspatial_entity (^e.g. knowledge, motivation, language, measure^)
        > Psychological_entity Information_entity Collection;

           Psychological_entity (^feature/product of mental activity, e.g. feeling^)
            > psychological_feature;

           Information_entity (^content/element of a description^)
```

```
> {Description Description_container Attribute_or_measure};

   Description (^description of a situation^)
    > Description_content Description_medium;

       Description_content (^e.g. a narration, an hypothesis^)
        > Proposition Narration Process_representation
                 Description_with_KADS_inference_structure Role message;

           Narration (^report, story, biography, etc.^);

           Process_representation (^process history/specification^)
            > Process_history Process_specification;

               Process_history (^information describing a past process^);

               Process_specification (^information specifying the possible executions of some
                   processes, e.g. a computer program, a musical score^)
                > Kinetic_script Procedure;

                   Kinetic_script (^script on motions^);

                   Procedure (^e.g. a computer program, a schedule, a musical score^)
                    > Protocol;

                        Protocol (^e.g. a network protocol such as HTTP 4.0^)
                         > Network_protocol;

                            Network_protocol (^e.g. FTP, HTTP^)
                             > FTP HTTP;

           Description_with_KADS_inference_structure (^dataflow graph of "inferences" (tasks)
                   the inputs/outputs of which are described by "roles"^);

           Role
            > Hypothesis Observable Finding Complaint Norm Difference Discrepancy_class
                   Diagnosis_result Parameter System_model Historical_data;

       Description_medium (^e.g. a syntax, a language, a script^)
        > Language Abstract_data_type representation communication;

           Language (^natural/artificial language^)
            > {Natural_language Controlled_language};

               Controlled_language
                > Procedural_language Declarative_language Markup_language;

                   Procedural_language
                    > {C Lisp Javascript};

                   Declarative_language
                    > {Prolog KIF Conceptual_Graph_language};

                   Markup_language
                    > {XML HTML};

                        XML (^a version of the Extensible Markup Language^)
                         > XML_notation_for_RDF;

                            XML_notation_for_RDF (^an XML notation for the RDF model^)
                             : RDF_syntax_of_22-02-1999;

                                 RDF_syntax_of_22-02-1999 (^description in
                   http://www.w3.org/TR/1999/REC-rdf-syntax-19990222^);

           Abstract_data_type (^atomic or structured^)
            > {Atomic_ADT Structured_ADT} Literal;

               Atomic_ADT
                > Number Boolean;

                   Boolean (^two instances: true and false^)
                    : True False;

               Structured_ADT
                > Ordered_open_collection_ADT Ordered_closed_collection_ADT Model_ADT
                   Statement;

                   Ordered_open_collection_ADT
                    > List_ADT Stack Keyed_collection_ADT;

                   Ordered_closed_collection_ADT
                    > Array Queue;

                   Model_ADT
                    > Graph_model KADS_model;
```

```
                       Graph_model
                        > Conceptual_Graph_model RDF_model;

      Description_container (^e.g. file, image (but not disk or paper)^)
       > Document_element File_in_special_format Special_file;

         Document_element (^a part of a document^)
          > Document;

            Document (^the entire content of a document^)
             > Unretrievable_document;

         File_in_special_format
          > {Textual_file Binary_file};

            Textual_file
             > Plain_text_file Structured_text_file Encoded_text_file;

               Structured_text_file
                > HTML_file;

               Encoded_text_file
                > Postscript_file;

            Binary_file
             > GIF_file;

         Special_file
          > Software Documentation Repository;

            Software
             > Freeware Web_search_engine;

            Documentation
             > Tutorial Lecture User_manual;

            Repository
             > Web_index Yellow_Pages Home_page Cookie;

               Web_index
                > Web_index_and_search_engine;

      Attribute_or_measure (^e.g. mass, mass unit, 1 kg, frequency, [2-3] hz, color, blue, speed,
                  1 m/s^)
       > Time_measure Spatial_attribute_or_measure Physical_attribute_or_measure
                  Psychological_attribute_or_measure Process_attribute_or_measure
                  Modality_measure measure attribute property;

         Time_measure (^measure of points or intervals in time^)
          > {Time_point Time_period} time;

         Spatial_attribute_or_measure (^e.g. length measure in meters^)
          > Length_measure Area_measure Volume_measure;

            Length_measure
             > Length_measure_in_meter Length_measure_in_feet;

         Physical_attribute_or_measure (^e.g. mass/length/color measure^)
          > Mass_measure Color_measure;

         Process_attribute_or_measure (^e.g. a speed measure in km/h^)
          > Speed_measure;

            Speed_measure
             > Speed_measure_in_meter_per_second;

         Modality_measure (^e.g. "Never" might be declared as an instance^)
          > Temporal_modality_measure;

            Temporal_modality_measure
             : Never Rarely Often Always;

   Collection (^something gathering separated things (entities/situations)^)
    > group set Container Class Property Constraint_resource;

      Container
       > {Bag Seq Alt} Set(pm) Number_container(pm);

         Alt (^alternatives (exclusive or inclusive?)^)
          > Or_bag(pm) Xor_bag(pm);

            Or_bag (^bag of OR-ed elements^);

            Xor_bag (^bag of XOR-ed elements^);

         Set (^container where duplicate elements are not allowed^)
          > Or_set Xor_set;
```

```
                Or_set (^set of OR-ed elements^);

                Xor_set (^set of XOR-ed elements^);

        Class (^all classes are instance of that object^)
         : Thing(pm);

        Property (^all binary relation classes are instance of that object^)
         > Constraint_property Container_membership_property Constrained_relation_class(pm),
         : relation(pm);

            Constrained_relation_class
             > Antisymmetric_relation_class Irreflexive_relation_class Reflexive_relation_class
                        Symmetric_relation_class Transitive_relation_class
                        Weak_transitive_relation_class Many_to_many_relation_class
                        Many_to_one_relation_class One_to_many_relation_class;

                Antisymmetric_relation_class
                 > Asymmetric_relation_class Partial_order_relation_class;

                    Partial_order_relation_class
                     > Total_order_relation_class;

                Irreflexive_relation_class
                 > Asymmetric_relation_class;

                Reflexive_relation_class
                 > Equivalence_relation_class Partial_order_relation_class;

                Symmetric_relation_class
                 > Equivalence_relation_class;

                Transitive_relation_class
                 > Equivalence_relation_class Partial_order_relation_class;

        Constraint_resource
         > Constraint_property;

Undivisible_entity (^classification under this category is application-dependant^);

Divisible_entity
 > {Divisible_entity_without_discrete_parts Composite_entity};

    Composite_entity (^divisible entity with discrete parts^)
     > Collection whole;

Entity_playing_some_role (^e.g. an agent, an owner^)
 > Owned_entity Entity_part variable Process_recipient Process_object Causal_entity
                        Imaginary_entity {necessity inessential} thing anticipation;

    Owned_entity
     > possession;

    Entity_part
     > part part unit substance;

    Process_recipient (^recipient of a process^)
     > recipient;

    Process_object
     > subject;

    Causal_entity (^something (animal or software agent) able to act^)
     > Living_entity Goal_directed_agent Perhaps_goal_directed_causal_entity
                        Without_goal_causal_entity causal_agent;

        Goal_directed_agent (^goal directed causal entity (ex:a problem solver or an interactional
                        agent)^)
         > Cognitive_agent;

            Cognitive_agent (^for example an animal or an AI-agent^)
             > {Conscious_agent Non_conscious_cognitive_agent};

                Conscious_agent (^for example a person^)
                 > person;

                Non_conscious_cognitive_agent (^e.g. AI_Agent^);

        Perhaps_goal_directed_causal_entity (^e.g. supernatural forces^);

        Without_goal_causal_entity (^non conscious entity and not AI_Agent^);

    Imaginary_entity (^an entity that has been imagined^)
     > Imaginary_spatial_entity;

        Imaginary_spatial_entity (^e.g. a cartoon character^)
```

```
            > imaginary_place;

   Thing_playing_some_role (^category to classify things according to roles/viewpoints; classification
                             under this category is application-dependant^)
    > Mediation Thing_needed_for_some_process Situation_playing_some_role Entity_playing_some_role
                             relation;

       Mediation (^Peirce/Sowa's notion of "thirdness"^);

       Thing_needed_for_some_process (^e.g. something needed for an application^)
        > Thing_needed_for_knowledge_engineering;

          Thing_needed_for_knowledge_engineering
           > Thing_needed_for_KADS_knowledge_engineering;

             Thing_needed_for_KADS_knowledge_engineering
              > Knowledge_engineering_with_KADS KADS_model Description_with_KADS_inference_structure
                             Role;


relation (Thing,Thing) (^superclass of relations; instance of Property^)
 > relation_from_property relation_from_class comment is_defined_by label see_also different
                             relation_from_collection relation_to_collection relation_from_description
                             relation_from_situation relation_from_time_measure
                             relation_from_spatial_entity attributive_relation ordering_relation,
 ~ Thing;

   relation_from_property (Property,Thing)
    > range domain sub_property_of inverse;

       range (Property,Class);

       domain (Property,Class);

       sub_property_of (Property,Property);

       inverse (Property,Property);

   relation_from_class (Class,Thing)
    > sub_class_of exclusive_class wnObject wnNounType;

       sub_class_of (Class,Class);

       exclusive_class (Class,Class);

       wnObject (Class,Thing);

       wnNounType (Class,Thing);

   comment (Thing,Literal);

   is_defined_by (Thing,Thing)
    > definition;

       definition (Thing,Description);

   label (Thing,Literal);

   see_also (Thing,Thing);

   different (Thing,Thing);

   relation_from_collection (Collection,Thing)
    > member size minimal_size maximal_size percentage average sub_collection overlapping_collection
                             not_overlapping_collection;

       member (Collection,Thing);

       size (Collection,Literal) (^number of elements^);

       minimal_size (Collection,Literal);

       maximal_size (Collection,Literal);

       percentage (Collection,Literal);

       average (Number_container,Literal) (^an average on the values^);

       sub_collection (Collection,Collection);

       overlapping_collection (Collection,Collection);

       not_overlapping_collection (Collection,Collection)
        > collection_complement;

          collection_complement (Collection,Collection);
```

```
relation_to_collection (Thing,Collection)
 > parts instances subclasses;

    parts (Thing,Collection);

    instances (Class,Collection);

    subclasses (Class,Collection);

relation_from_description (Description,Thing)
 > and contextualizing_relation author authoring_time description_object description_instrument
                           description_support rhetorical_relation argumentation_relation;

    and (Description,Description);

    contextualizing_relation (Description,Thing)
     > contextualizing_logical_relation modality believer;

        contextualizing_logical_relation (Description,Thing)
         > truth or xor implication;

            truth (Description,Boolean);

            or (Description,Description);

            xor (Description,Description);

            implication (Description,Description)
             > equivalence;

                equivalence (Description,Description);

        modality (Description,Modality_measure);

        believer (Description,Causal_entity);

    author (Description,Causal_entity);

    authoring_time (Description,Time_measure);

    description_object (Description,Thing);

    description_instrument (Description,Description_medium);

    description_support (Description,Description_container);

    rhetorical_relation (Description,Description);

    argumentation_relation (Description,Description)
     > answer contribution replacement confirmation reference argument contradiction;

        answer (Description,Description);

        contribution (Description,Description);

        replacement (Description,Description);

        confirmation (Description,Description);

        reference (Description,Description);

        argument (Description,Description)
         > weak_argument strong_argument;

            weak_argument (Description,Description);

            strong_argument (Description,Description)
             > proof;

                proof (Description,Description);

        contradiction (Description,Description);

relation_from_situation (Situation,Thing)
 > relation_from_situation_to_time_measure relation_from_situation_to_situation
                           relation_from_process;

    relation_from_situation_to_time_measure (Situation,Time_measure)
     > time duration since_time until_time;

        time (Situation,Time_measure);

        duration (Situation,Time_measure);

        since_time (Situation,Time_measure);
```

```
        until_time (Situation,Time_measure);

    relation_from_situation_to_situation (Situation,Situation)
     > later_situation;

        later_situation (Situation,Situation)
         > next_situation consequence;

            next_situation (Situation,Situation)
             > termination;

                termination (Situation,Situation);

            consequence (Situation,Situation);

    relation_from_process (Process,Thing)
     > purpose triggering_event ending_event precondition postcondition agent initiator experiencer
                           instrument object result recipient sub_process method source destination
                           path process_attribute;

        purpose (Process,Situation);

        triggering_event (Process,Event);

        ending_event (Process,Event);

        precondition (Process,State);

        postcondition (Process,State);

        agent (Process,Entity);

        initiator (Process,Causal_entity);

        experiencer (Process,Causal_entity);

        instrument (Process,Entity);

        object (Process,Thing)
         > input input_output;

            input (Process,Thing)
             > material parameter;

                material (Process,Thing);

                parameter (Process,Thing);

            input_output (Process,Thing)
             > object_to_modify object_to_destroy;

                object_to_modify (Process,Thing)
                 > object_to_mute;

                    object_to_mute (Process,Thing);

                object_to_destroy (Process,Thing);

        result (Process,Thing)
         > output;

            output (Process,Thing);

        recipient (Process,Entity);

        sub_process (Process,Process);

        method (Process,Description);

        source (Process,Spatial_entity);

        destination (Process,Spatial_entity);

        path (Process,Spatial_entity);

        process_attribute (Process,Process_attribute_or_measure)
         > speed manner;

            speed (Process,Speed_measure);

            manner (Process,Process_attribute_or_measure);

relation_from_time_measure (Time_measure,Thing)
 > near_time before_time;

    near_time (Time_measure,Time_measure);
```

```
    before_time (Time_measure,Time_measure) (^instance of _total_order_relation^);

relation_from_spatial_entity (Spatial_entity,Thing)
 > on_location above_location in_location near_location interior_location exterior_location
                           before_location;

    on_location (Spatial_entity,Spatial_entity);

    above_location (Spatial_entity,Spatial_entity);

    in_location (Spatial_entity,Spatial_entity);

    near_location (Spatial_entity,Spatial_entity);

    interior_location (Spatial_entity,Spatial_entity);

    exterior_location (Spatial_entity,Spatial_entity);

    before_location (Spatial_entity,Spatial_entity);

attributive_relation (Thing,Thing)
 > owner generator attribute;

    owner (Thing,Causal_entity);

    generator (Thing,Causal_entity)
     > parent;

       parent (Thing,Causal_entity);

    attribute (Thing,Attribute_or_measure) (^e.g. [a car, color: red]^)
     > spatial_attribute physical_attribute process_attribute;

       spatial_attribute (Spatial_entity,Spatial_attribute_or_measure)
        > length area volume;

          length (Spatial_entity,Length_measure);

          area (Spatial_entity,Area_measure);

          volume (Spatial_entity,Volume_measure);

       physical_attribute (Physical_entity,Physical_attribute_or_measure)
        > mass color;

          mass (Physical_entity,Mass_measure);

          color (Physical_entity,Color_measure);

ordering_relation (Thing,Thing) (^e.g. type, sub_class_of, part, equal^)
 > partial_order_relation equivalence_relation;

    partial_order_relation (Thing,Thing) (^instance of _partial_order_relation_class^)
     > total_order_relation inferior_or_equal generalization wnSubstance wnMember member part;

       total_order_relation (Thing,Thing)
        > inferior;

          inferior (Thing,Thing) (^general class to specialize^);

       inferior_or_equal (Thing,Thing);

       generalization (Thing,Thing)
        > sub_class_of type;

          type (Thing,Class);

       wnSubstance (Thing,Thing);

       wnMember (Thing,Thing) (^member relation in WordNet^);

       part (Thing,Thing)
        > physical_part sub_collection sub_situation main_part first_part last_part;

          physical_part (Physical_entity,Physical_entity);

          sub_situation (Situation,Situation)
           > sub_process;

          main_part (Thing,Thing);

          first_part (Thing,Thing);

          last_part (Thing,Thing);

    equivalence_relation (Thing,Thing) (^instance of equivalence_relation_class^)
     > similar equal;
```

```
similar (Thing,Thing);
equal (Thing,Thing) (^"=" in KIF^);
```