NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 — "Semantic-based knowledge and content systems"

D2.1.2 The collaborative ontology design ontology (v2)

Deliverable Co-ordinator: Aldo Gangemi

Deliverable Co-ordinating Institution: Consiglio Nazionale delle Ricerche (CNR)

Other Authors: Valentina Presutti (CNR)

This deliverable presents a substantial update of the C-ODO ontology design metamodel, called codolight. Codolight is now linked to requirements and application tasks, has been used for tool descriptions, aligned to external vocabularies, is lighter in complexity, and improves association between the social and software layers of ontology design aspects.

Document Identifier:	NEON/2007/D2.1.2/v1.0	Date due:	February 28th, 2009
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	February 28th, 2009
Project start date	March 1, 2006	Version:	v1.0
Project duration:	4 years	State:	Final
		Distribution:	Public

NeOn Consortium

This document is part of the NeOn research project funded by the IST Programme of the Commission of the European Communities by the grant number IST-2005-027595. The following partners are involved in the project:

Open University (OU) – Coordinator Knowledge Media Institute – KMi Berrill Building, Walton Hall Milton Keynes, MK7 6AA United Kingdom Contact person: Martin Dzbor, Enrico Motta E-mail address: {m.dzbor, e.motta}@open.ac.uk	**Universität Karlsruhe – TH (UKARL)** Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB Englerstrasse 11 D-76128 Karlsruhe, Germany Contact person: Peter Haase E-mail address: pha@aifb.uni-karlsruhe.de
Universidad Politécnica de Madrid (UPM) Campus de Montegancedo 28660 Boadilla del Monte Spain Contact person: Asunción Gómez Pérez E-mail address: asun@fi.ump.es	**Software AG (SAG)** Uhlandstrasse 12 64297 Darmstadt Germany Contact person: Walter Waterfeld E-mail address: walter.waterfeld@softwareag.com
Intelligent Software Components S.A. (ISOCO) Calle de Pedro de Valdivia 10 28006 Madrid Spain Contact person: Jesús Contreras E-mail address: jcontreras@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SL–1000 Ljubljana Slovenia Contact person: Marko Grobelnik E-mail address: marko.grobelnik@ijs.si
Institut National de Recherche en Informatique et en Automatique (INRIA) ZIRST – 665 avenue de l'Europe Montbonnot Saint Martin 38334 Saint-Ismier, France Contact person: Jérôme Euzenat E-mail address: jerome.euzenat@inrialpes.fr	**University of Sheffield (USFD)** Dept. of Computer Science Regent Court 211 Portobello street S14DP Sheffield, United Kingdom Contact person: Hamish Cunningham E-mail address: hamish@dcs.shef.ac.uk
Universität Kolenz-Landau (UKO-LD) Universitätsstrasse 1 56070 Koblenz Germany Contact person: Steffen Staab E-mail address: staab@uni-koblenz.de	**Consiglio Nazionale delle Ricerche (CNR)** Institute of cognitive sciences and technologies Via S. Marino della Battaglia 44 – 00185 Roma-Lazio Italy Contact person: Aldo Gangemi E-mail address: aldo.gangemi@istc.cnr.it
Ontoprise GmbH. (ONTO) Amalienbadstr. 36 (Raumfabrik 29) 76227 Karlsruhe Germany Contact person: Jürgen Angele E-mail address: angele@ontoprise.de	**Food and Agriculture Organization of the United Nations (FAO)** Viale delle Terme di Caracalla 00100 Rome Italy Contact person: Marta Iglesias E-mail address: marta.iglesias@fao.org
Atos Origin S.A. (ATOS) Calle de Albarracín, 25 28037 Madrid Spain Contact person: Tomás Pariente Lobo E-mail address: tomas.parientelobo@atosorigin.com	**Laboratorios KIN, S.A. (KIN)** C/Ciudad de Granada, 123 08018 Barcelona Spain Contact person: Antonio López E-mail address: alopez@kin.es

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to the writing of this document or its parts:

- CNR

Change Log

Version	Date	Amended by	Changes
0.1	27-01-2009	Valentina Presutti	Created Outline
0.2	01-02-2009	Aldo Gangemi	Filled in Introduction and preliminary description of sections
0.3	10-02-2009	Valentina Presutti	Add section 1.2 and organized overall structure
0.4	17-02-2009	Valentina Presutti and Aldo Gangemi	revised chapters introduction, kernel, and workflows
0.5	19-02-2009	Valentina Presutti and Aldo Gangemi	introductions of chapters added, formal descriptions of entities added in chapters workflows and tools, figures added (draft sent to QA)
0.7	20-02-2009	Valentina Presutti and Aldo Gangemi	chapters revised, formal descriptions of entities added in chapter data, figures revised.
0.8	25-02-2009	Valentina Presutti and Aldo Gangemi	introduction and kernel revision.
0.9	27-02-2009	Valentina Presutti	added formal descriptions in data, projects and interfaces, added alignment tables.
1.0	28-02-2009	Valentina Presutti and Aldo Gangemi	final version

Executive Summary

This deliverable introduces the new *light* version of C-ODO network of ontologies, called **codolight**. We describe codolight modular architecture, each module separately, its alignments to commonly used and NeOn proper vocabularies, and a summary of its applications that are fully described in D2.3.2.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

When the first version of C-ODO was developed in 2006 [GLP+07], not much had been developed in order to model ontology design requirements and descriptions. As of today, this is still true, although at least one attempt [SNTM08] has been made to model ontology engineering workflows, partly by reusing the basic approach of C-ODO.

C-ODO is a set of ontologies that attempt to provide a vocabulary to talk about ontology design requirements and descriptions. When dealing with ontology design, however, most of the activities are carried out with the help of software tools, so that the worlds of ontology design and software design have a reasonable overlap, which goes well beyond the need of good software tools in order to perform good ontology design.

As the recent history of the Web of Data shows, the development of RDF datasets, their reengineering practices, the usage of OWL vocabularies to reason on them, and now also their visualization and interaction aspects are partly interrelated, and while becoming prominent, they largely overlap web design aspects.

Similarly, we found that few advancements can be made for boosting the creation and usage of elegant, efficient, and practical ontologies, if few attention is given to a healthy analysis of design activities, requirements, and the relations between human ontology design, and ontology design tools.

It is no surprise then that after the first release of C-ODO in 2006 [CGL+06, GLP+07], which was focused on describing the practices of ontology design as a mainly human-centered set of activities, we started realizing that more effort should be involved in describing the actual pieces of software that accompany those activities, and the actual data ("knowledge types") that are managed computationally in order to create, maintain, and annotate an ontology or a network of ontologies. In the first release we had already discussed the differences and complementarity between "social" and "computational" aspects of ontology design. The ontology developed at that time was however limited on the computational side. This time, we have tried to allow correspondences for each pair of aspects, e.g. ontology projects (social side) are associated with (digital) projects, like Eclipse or SourceForge ones; workflow/planning (social side) can be specialized as a computational workflow; conceptual requirements and conceptual solutions (social side) can be expressed as competency questions/queries respectively ontology design patterns (computational side).

This deliverable presents the results of the new developments on C-ODO Light (*codolight* hereafter), leveraging its application to the description of several design tools developed in NeOn as plugins to the NeOn Toolkit. The deliverable also introduces the alignments made between codolight and several commonly used and newly proposed ontologies that are related to ontology design.

Tool descriptions and alignments have played the role of "use cases" for codolight, helping its design a lot. One use case has in particular speeded up codolight design: the development of the Kali-ma tool [PPG+09], which is supposed to provide new ways of composing implemented ontology design functionalities ("capabilities") according to explicit design needs. Kali-ma is being developed as a programmatical realization of codolight in a distributed space of ontology design tools.

1.1 Collaborative Ontology Design and C-ODO Light

Codolight takes into account new *ontology requirements* that make it departing from the original C-ODO. These requirements have been acquired from experience in modeling tool descriptions, in alignments to existing vocabularies, and after user feedback:

1. Ability to formalize ontology design tool descriptions in terms of input/output data (*knowledge types*), functionalities, interface objects and interaction patterns
2. Smooth integration between human-oriented and tool-oriented descriptions of ontology design aspects
3. Alignment to existing vocabularies such as DOAP, OMV, etc.
4. Lighter axiomatization (e.g. no anonymous classes in restrictions)
5. Modular development according to pattern-based design, which reuses the ontologydesignpatterns.org practices

Besides ontology requirements, we have also addressed several *ontology application tasks* that codolight is supposed to help achieving:

i Browsing semantic data about ontology projects, tools, data, repositories, solutions, discussions, evaluations, etc.

ii Searching and selecting design components based on design aspects, knowledge types, individual needs, user profiles, etc.

iii Creating design configuration interfaces that help/automatize the previous task

iv Help collecting ontology requirements, design functionalities, and ontology application tasks for an ontology project

v Providing a shared network of vocabularies to create/query/reason on annotations and data related to ontology projects, including integration between annotations that have heterogeneous provenance, like in annotations coming from collaborative discussions, mixed with annotation produced by change management.

Part of these tasks are being implemented within NeOn in the Kali-ma tool (see [PPG+09]).
The C-ODO Light network of ontologies (Fig. 1.1) is currently organized according to a 5-layer architecture:

Pattern layer : it contains reusable content ontology design patterns (Content ODP) [PG08] that include e.g. *sequence, partof, situation, collectionentity*, etc. The patterns (Fig. 1.2) are reused in the design of the ontologies constituting the “corolla” architecture of codolight.

Core codolight layer : it contains the nine modules of the codolight core network of ontologies, organized as in a corolla, with codkernel module in the center, and the modules: coddata, codprojects, codworkflows, codarg, codsolutions, codtools, codinterfaces, and codinteraction importing codkernel (Fig. 1.3).

Plugin layer : it consists of the modules containing the descriptions of the NeOn plugins related to ontology design, formalized in OWL by reusing the codolight vocabulary and some of the alignment modules (Fig. 1.4).

Dashboard inference layer : it consists of the modules containing the definition of the *design aspects* according to which tools, knowledge types, and functionalities are organized, as well as the closure of inferences derived from the reasoners applied to the previous layers (Fig. 1.5).

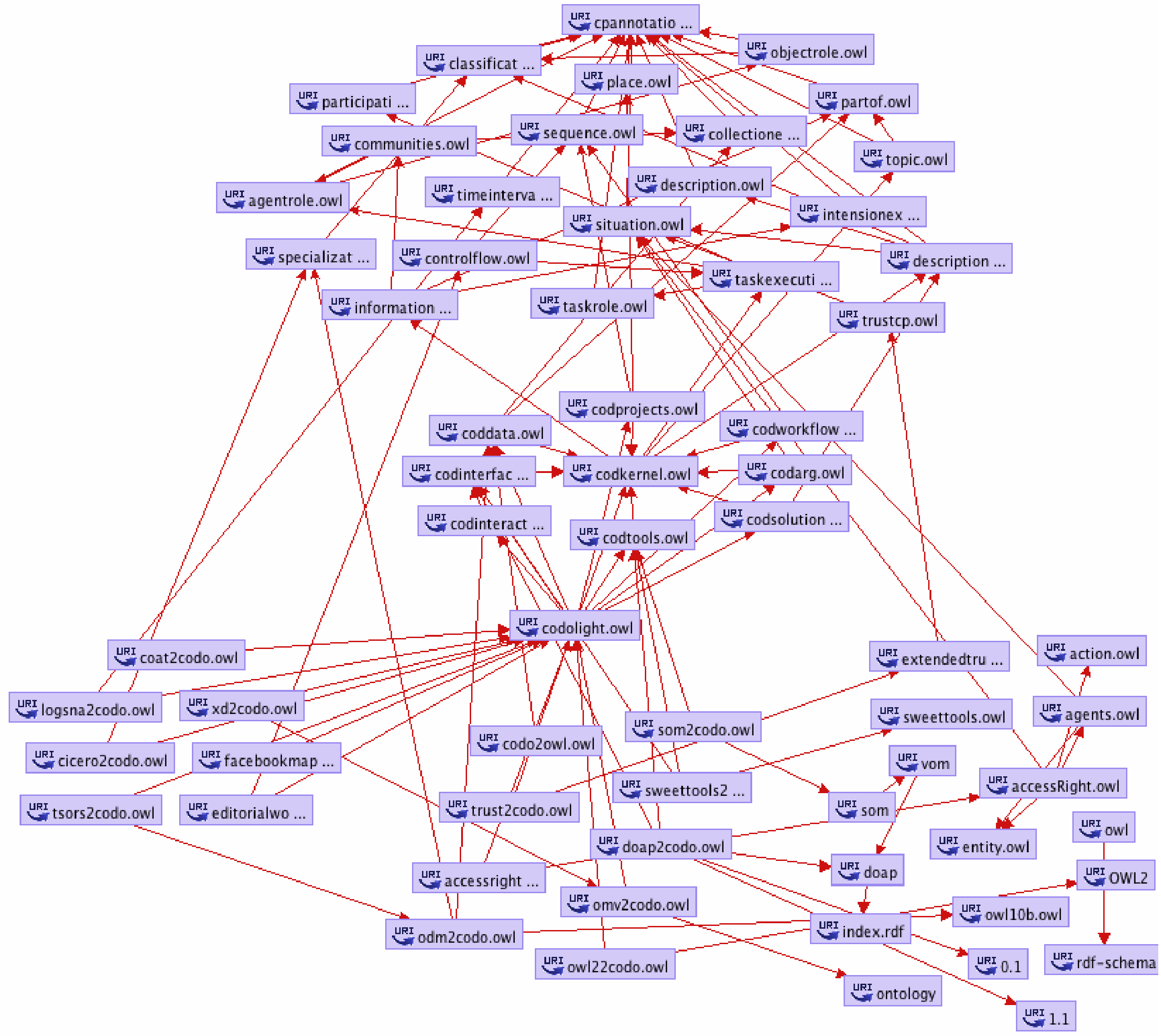

Figure 1.1: The codolight network.

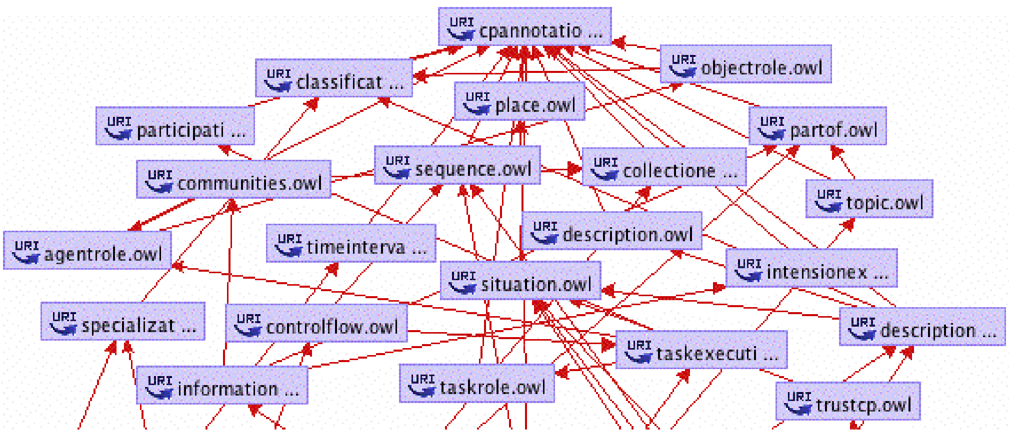

Figure 1.2: The pattern layer in the codolight network.

Alignment layer : it consists of the modules containing mapping axioms between codolight and related

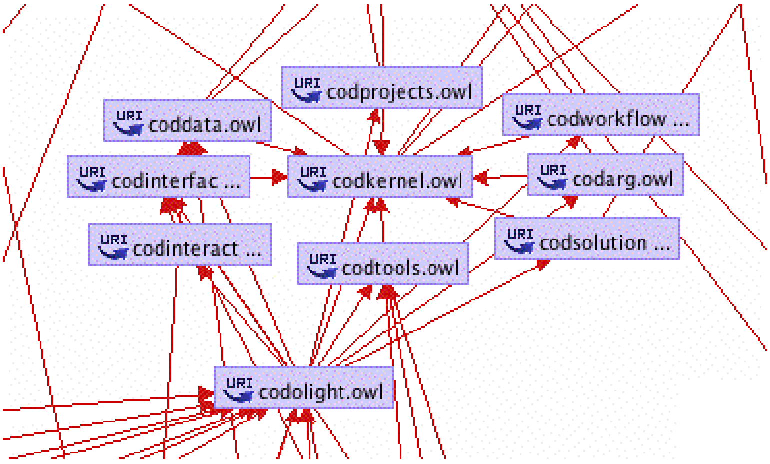

Figure 1.3: The core layer in the codolight network.

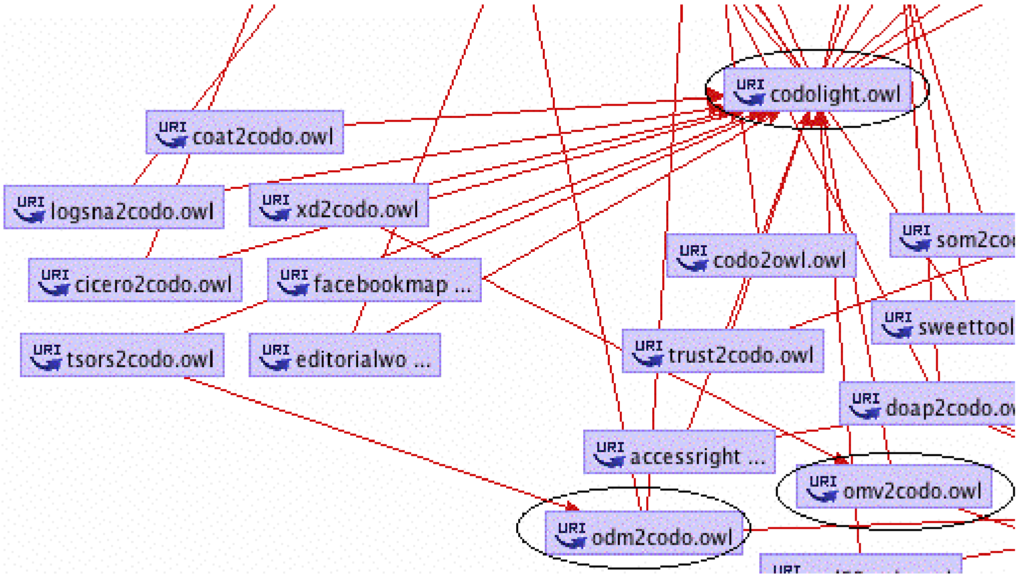

Figure 1.4: The plugin layer in the codolight network, with imports from codolight library and from alignments to external vocabularies.

vocabularies, currently: OMV, DOAP, FOAF, NeOn Access Rights ontology, NeOn OWL1.0 metamodel, NeOn OWL2 metamodel, the RDF-OWL datamodel, and the SweetTools vocabulary (Fig. 1.6).

The transitive closure of all modules in the five layers is loadable through the OWL ontology: http://www.ontologydesignpatterns.org/cpont/codo/allcodomappings.owl, which only contains owl:import axioms.

A relevant fragment of *codolight* is depicted in Fig. 1.7.

1.2 Architecture and main modules of codolight

Codolight ontology network encodes the main aspects of ontology design by following an architectural ontology design pattern called *corolla*. The corolla pattern suggests an overall (externally observable) shape for the network composed of a *kernel* module, which includes the definition of core concepts of the domain of interest, and a set of *petal* modules, each defining a specific aspect of the domain of interest. The *kernel* module defines core concepts, shared by all aspects and is imported by all *petal* modules. Petal modules

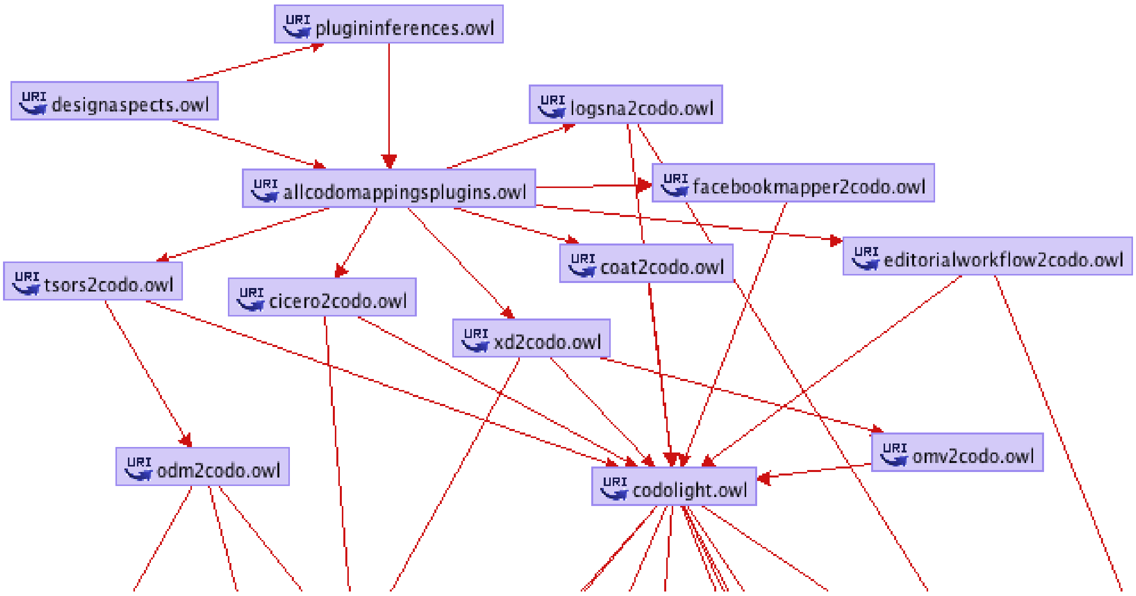

Figure 1.5: The dashboard layer in the codolight network.

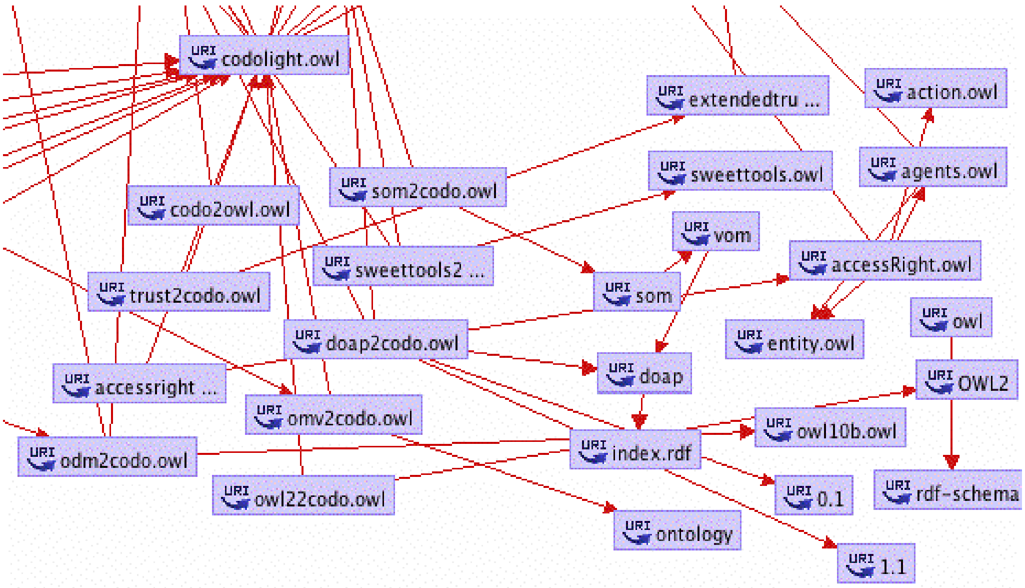

Figure 1.6: The alignment layer in the codolight network: external vocabularies are aligned to codolight library.

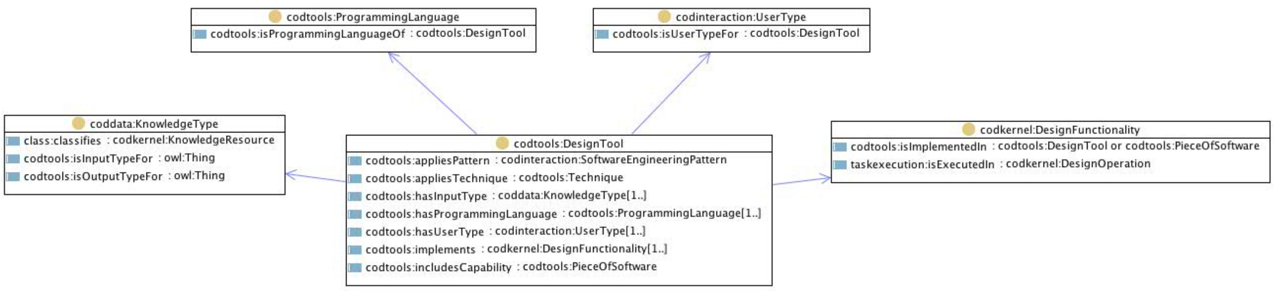

Figure 1.7: The vocabulary for the class DesignTool in codolight.

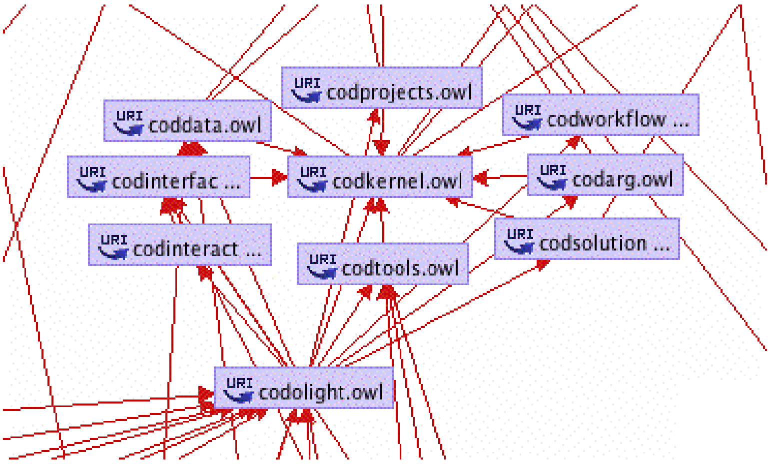

Figure 1.8: The codolight network *corolla* architecture.

refine the axiomatization of at least one of the core concepts, i.e. they add detail for at least one aspect. The corolla pattern allows to minimize dependencies between different modules of an ontology network; i.e. modules are loosely coupled, and suggests an organization of the network, by which different aspects of the domain of interest are represented by each petal module.

The general competency question for a *corolla*-based ontology network can be the following: what are the main aspects of the domain described by the ontology network?

In the case of *codolight* the following *petal* modules are defined, as depicted in Figure 1.8:

Data. This module, detailed in chapter 3, contains the main notions classifying the data managed when designing an ontology: ontologies, ontology elements, Knowledge Organization Systems (KOS), KOS elements, rules, modules, encoding syntaxes, etc. For each class of knowledge resources, a *knowledge type* instance is provided. This module has a greater detail compared to the original C-ODO one

Project. This module, detailed in chapter 4, contains the minimal vocabulary for representing ontology design projects and their executions. An ontology project is here taken as a social entity, whose computational counterpart (e.g. a “project” created in the NeOn Toolkit) is a software entity that collects resources and descriptions related to an ontology project. This module has a minor detail compared to the original C-ODO one

Workflows. This module, detailed in chapter 5, contains classes and properties to represent workflows from within ontology projects: collaborative workflows, accountable agents, need for an agent or a design functionality, etc. This module has a minor detail compared to the original C-ODO one, where the main focus was on talking about workflow collaboration types

Argumentation. This module, detailed in chapter 6, contains the basic classes and properties to represent argumentation concepts: arguments, threads, ideas, positions, rationales, etc. Also here, the module has a minor detail compared to the original C-ODO one

Solutions. This module, detailed in chapter 7, contains classes and properties to represent ontology design solutions: competency questions, ontology design patterns, ontology requirements, unit tests, etc. This module has a greater detail compared to the original C-ODO one

Tools. This module, detailed in chapter 8, contains classes and properties to represent ontology design tools: tools, pieces of code, code entities, computational tasks, input and output data relations, etc. This module has a much greater detail compared to the original C-ODO one

Interaction. This module, detailed in chapter 10, contains classes and properties that represent some typical interaction entities: user types, computational tasks and workflows, etc. This moduleis totally new with respect to the original C-ODO library

Interfaces. This module, detailed in chapter 9, contains classes and properties that represent some typical interface entities: interface objects, panes, windows, etc. This module is totally new with respect to the original C-ODO library.

1.3 Conventions for modules description

In the following chapters we describe in detail *codolight* modules. For each module, we describe what content patterns have been reused as building blocks, what entities and axioms the module defines locally, and what relations, if any, the module has with the other *codolight* petals. In this section, we define the conventions that are used in the next chapters.

1.3.1 Notation and prefixes

First of all we use the *n3* (*"turtle"*) notation [BLC08] for describing the formal definition of entities and for any additional axioms asserted in a module. In a specific module, the entities defined locally have no prefix, while the entities defined externally have a prefix according to tables 1.1, 1.2, and 1.3. For example, the class `Ontology` is referred to in the *kernel* module without any prefix because it is defined locally, while the same class in the *data* module is referred with the prefix `data:` because it is externally defined with respect to that module.

Table 1.1: Prefixes used in place of namespaces for *codolight* proper modules.

prefix	**namespace**
coddata:	http:www.ontologydesignpatterns.orgcpontcodocoddata.owl#
codkernel:	http:www.ontologydesignpatterns.orgcpontcodocodkernel.owl#
codprojects:	http:www.ontologydesignpatterns.orgcpontcodo.owl#
codworkflows:	http:www.ontologydesignpatterns.orgcpontcodo.owl#
codargumentation:	http:www.ontologydesignpatterns.orgcpontcodo.owl#
codsolutions:	http:www.ontologydesignpatterns.orgcpontcodo.owl#
codtools:	http:www.ontologydesignpatterns.orgcpontcodo.owl#
codinterfaces:	http:www.ontologydesignpatterns.orgcpontcodo.owl#
codinteraction:	http:www.ontologydesignpatterns.orgcpontcodo.owl#

1.3.2 Figures

Diagrams for each *codolight* module and for each alignment module are provided by using a new graph-based ontology visualization approach, which is briefly explained later in this sectionİ. The reason for the use of a new approach is the substantial lack of *intuitive* graphic visualization patterns for concept-level associations within ontologies. Existing tools cover the matter with many different solutions, and most of them can be grouped under four grossly defined classes. The intent of this classification, which is by no

Table 1.2: Prefixes used in place of namespaces for Content Ontology Design Patterns.

prefix	namespace
descriptionandsituation:	http:www.ontologydesignpatterns.orgcpowldescriptionandsituation.owl#
representation:	http:www.ontologydesignpatterns.orgcpowlinformationobjectsandrepresentationlanguages.owl#
taskexecution:	http://www.ontologydesignpatterns.org/cp/owl/taskexecution.owl#
topic:	http:www.ontologydesignpatterns.orgcpowltopic.owl#
objectrole:	http:www.ontologydesignpatterns.orgcpowlobjectrole.owl#
classification:	http:www.ontologydesignpatterns.orgcpowlclassification.owl#
description:	http:www.ontologydesignpatterns.orgcpowldescription.owl#
agentrole:	http:www.ontologydesignpatterns.orgcpowlagentrole.owl#
taskrole:	http:www.ontologydesignpatterns.orgcpowltaskrole.owl#
intensionextension:	http:www.ontologydesignpatterns.orgcpowlintensionextension.owl#
situation:	http:www.ontologydesignpatterns.orgcpowlsituation.owl#
partof:	http:www.ontologydesignpatterns.orgcpowlpartof.owl#
sequence:	http:www.ontologydesignpatterns.orgcpowlsequence.owl#
collection:	http:www.ontologydesignpatterns.orgcpowlcollectionentity.owl#
place:	http:www.ontologydesignpatterns.orgcpowlplace.owl#

Table 1.3: Prefixes used in place of namespaces for external vocabularies.

prefix	namespace
xsd:	http:www.w3.org2001XMLSchema#
rdfs	http:www.w3.org200001/rdf-schema#
owl:	http:www.w3.org200207/owl#

means exhaustive, is to explicitate the choices with respect to what *graph* should be visualized of an ontology content, and why we chose a custom one:

1. RDF visualizers: an OWL-RDF graph is not filtered, and all its triples are visualized as node-edge-node visual constructs. A typical example is RDF Gravity[1], which however provides very nice filtering options for hiding parts of the graph.

2. Partial-order visualizers: an OWL-RDF graph is here filtered so that only `rdfs:subClassOf` triples between `owl:Class` instances are retained, usually filtering out also `owl:Restriction` instances. The most frequent visual semantics assumes that nodes represent `owl:Class` instances, and edges (labeled or not) represent `rdfs:subClassOf` instances. A typical example of this group is OWLViz [2]. In some cases, the approach is used to visualize taxonomies of properties other than `rdfs:subClassOf`.

3. RDFS visualizers: an OWL-RDF graph is here filtered so that (typically) `rdfs:subClassOf` triples between `owl:Class` instances are retained (and represented similarly to taxonomy visualizers), while `rdfs:domain` and `rdfs:range` axioms are retained, but transformed into edges between the domain class and the range class, labeled with the `rdfs:Property` name. All the other axioms/triples are filtered out (except `disjointFrom` in some cases). A typical example is TGViz.

4. DL visualizers: an OWL-RDF graph is not filtered out, but different visual semantics are exploited in order to visualize all description logic constructs present in the graph. For example, Top Braid Composer [3] has a diagrammatic tool that used an OWL profile for UML, which e.g. represents `owl:Restriction` instances as UML objects.

[1] http://semweb.salzburgresearch.at/apps/rdf-gravity/index.html
[2] http:protegewiki.stanford.eduindex.phpOWLViz
[3] http:www.topquadrant.comproductsTB_Suite.html

Each approach in the list has severe problems for the intuitiveness of the visualization. Plain RDF visualizers make very difficult to single out the core content of an ontology, and filtering requires much effort. Partial-order and RDFS visualizers miss a lot of important information that characterizes and ontology, e.g. `owl:Restriction` instances. DL visualizers do not miss any information, but visual semantics mirrors description logics datamodel, which is not very intuitive for the average expert that wants to make sense of the basic structure of an ontology.
At least one tool, the Ontology Visualization[4] plugin for the NeOn Toolkit[5], tries to get an intuitive visualization semantics for domain experts by defining appropriate rules (hard-coded):

- Root Node (Red): A starting node, selected in Ontology Navigator.
- Mandatory Node (Blue): A node that has direct relations to an ontology node, and belongs to the same ontology as the Root Node.
- Import Node (Light blue): A node that has relations to an ontology node, and does not belong to the same ontology as the Root Node.
- Inherit Node (Light brown): A node that has relations to the parents of an ontology node.

The effect is much better in general, but yet we cannot customize the graph to be visualized, because the rules are not changeable. For example, the relations are taken only from the domains and ranges of properties, but not from the restrictions declared for the root class or one of its parents, and one cannot change this setting. if customizable, this kind of simple graphs for intuitive OWL visualization would be satisfactory.
In order to overcome those problems, we have defined an approach, based on SPARQL CONSTRUCT queries, which retains all the relevant information characterizing core parts of an ontology. It consists in firstly running a filtering query (this one worked well for this ontology project, but different ones can be written for different requirements and visualization detail):

```
CONSTRUCT { ?subject1 ?subject2 ?subject3 }
WHERE
{
 { ?subject1 rdf:type owl:Class .
   ?subject1 rdfs:subClassOf owl:Thing }
 UNION
 {
  { ?subject2 rdf:type owl:ObjectProperty }
    UNION
  { ?subject2 rdf:type owl:DataProperty }
 }
 UNION
 { ?subject2 rdfs:domain ?subject1 .
   ?subject2 rdfs:range ?subject3 .}
 UNION
 { ?subject1 rdfs:subClassOf ?z .
    ?z owl:onProperty ?subject2 .
    ?z owl:someValuesFrom ?subject3 }
 UNION
 { ?subject1 rdfs:subClassOf ?z .
    ?z owl:onProperty ?subject2 .
    ?z owl:minCardinality ?q .
```

[4] http://www.neon-toolkit.org/wiki/index.php/Plugin_for_OWL_Ontology_Visualization
[5] http://www.neon-toolkit.org

```
    ?subject2 rdfs:range ?subject3 }
 UNION
 { ?subject1 rdfs:subClassOf ?z .
    ?z owl:onProperty ?subject2 .
    ?z owl:cardinality ?q .
    ?subject2 rdfs:range ?subject3 }
}
```

Secondly, the resulting triples are asserted in a separate RDF graph.
Thirdly, the graph is visualized with a good RDF visualizer, in this case RDF Gravity [6]. Visual semantics is `owl:Class` for nodes, and `rdfs:Property` or `rdf:subClassOf` for edges (red vs. blue). In some figures, we also have triangle nodes for individuals.
Fourthly, the visualization on RDF Gravity is filtered manually for fine-tuning.
All the graphs visualized in the next sections have been produced with this approach. They lack full DL semantics (because e.g. cardinality is not visualizable in RDF Gravity), but the converse advantage is to get an intuitive and compact visualization of the entities in the ontologies, similar to concept maps. However, figures are always accompanied by actual OWL code in N3 encoding.

[6]http:semweb.salzburgresearch.atappsrdf-gravity

Chapter 2

CODO Kernel: the core concepts

The *Kernel*[1] module of *codolight* defines core entities of the ontology network, the core vocabulary of ontology design. It only contains the main classes of *codolight*, that are aligned to content ontology design patterns that have been used as its building blocks.

Core entities represent the concepts that the other *petals* share by including this module. Each petal details a specific aspect of ontology design by exploiting the core concepts defined in the kernel, without the need of depending on other petals. By means of its classes, the *kernel* module traverses all core aspects of ontology design. Informally, the situation of an ontology design team approaching the design of an ontology can be described as follows: there is a set of *knowledge resources* about a certain domain i.e. *data*, available to the designer team. Such data have to be analyzed to the aim of producing *design solutions*. The designer team creates an *ontology project*, organized by means of *workflows* including discussions and evaluations. Hence, at least a very simple *argumentation* model is actually performed in any ontology project: a person has a certain position about an idea (data and/or solution), which is supported by some motivation i.e. its *design rationale*. The goal is either to produce or find (reuse) solutions. The whole situation and its parts can be supported by specific *tools*.

The *codolight kernel* covers the basics of such situation's vocabulary by defining the following entities.

2.1 Patterns reused in *codolight kernel* module

The *kernel* module of *codolight* has been built by reusing the following Content Ontology Design Patterns (CPs) as building blocks [PG08, PGGPF07].

Description and situation. This CP represents conceptualizations i.e., descriptions, and corresponding groundings i.e., situations. The pattern is extracted from DOLCE+DnS Ultralite[2] by partial cloning of elements, and is composed of three other CPs: description, situation, and classification.

Information objects and representation languages. This CP represents possible types of representation languages and the respective information objects that can be represented by them. It is the composition of and specializes the **intension extension** and the **part of** CPs.

Task execution. This CP represents actions through which tasks are executed. It allows designers to make assertions on roles played by agents without involving the agents that play that roles, and vice versa. It allows to express neither the context type in which tasks are defined, not the particular context in which the action is carried out. Moreover, it does not allow to express the time at which the task is executed through the action (for actions that do not solely executed that certain task).

[1] http://www.ontologydesignpatterns.org/cpont/codo/codkernel.owl

[2] http://www.ontologydesignpatterns.org/ont/dul/DUL.owl

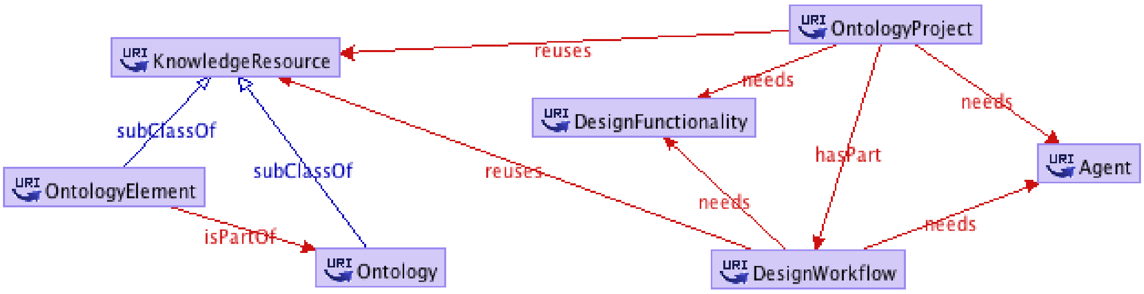

Figure 2.1: A simple graph of ontology elements from *codolight kernel* module.

Topic. This CP represents relation between things such as documents and their topic. It also represents typical relationships between topics, and between concepts and topics. Here a concept is conceptualized as a social objects, while a topic has a semiotic notion, and can be a collection of concepts.
The class encoding the conceptualization of a concept is defined in other CPs that are involved in the *codolight* ontology network, hence the following alignment axioms are included in this module.

Alignment axioms

```
classification:Concept
      owl:equivalentClass description:Concept , topic:Concept .
```

For further details on the above mentioned CPs, please refer either to [PGGPF07] or the web portal of ontology design patterns[3].
Please note that, such CPs can be in turn composed by other CPs, hence in *codolight* modules possible entities defined in additional CPs can be found. In those cases we will briefly introduce the entity reused and refer to CP where it is defined[4].

2.2 Entities of *codolight kernel* module

The following entities are defined in this module.

2.2.1 Knowledge resource

The class `KnowledgeResource` represents knowledge objects used as resources in an ontology project. Examples of knowledge resources are ontologies, thesauri, etc.

Formal definition.

```
:KnowledgeResource
      a       owl:Class ;
      rdfs:subClassOf intensionextension:InformationObject .
```

Such class is defined as a sub-class of `intensionextension:InformationObject`, a class of the CP “intension extension”, representing a piece of information, such as a musical composition, a text, a word, a picture, independently from how it is concretely realized.

[3] http://www.ontologydesignpatterns.org
[4] All CPs are available for download, the URI identifying their namespaces and listed in table 1.2 are also resolvable and allow any user to download the owl file.

2.2.2 Knowledge type

The class `KnowledgeType` identifies types of knowledge resources. It is used as the reification of the intension for any class of knowledge resources. Within codolight, knowledge types are very relevant, because they are used to annotate tools, workflows and functionalities with input and output types, to obtain awareness of design aspects covered by design tools, and finally to determine what kind of annotations are maintained across the ontology design lifecycle.

Formal definition.

```
:KnowledgeType
      a       owl:Class ;
      rdfs:subClassOf classification:Concept> .
```

The class `KnowledgeType` is formally defined as a sub-class of `classification:Concept` defined in the CP *classification* and aligned to other classes as explained in section 2.1. A concept is a social object, and is defined in some description.

2.2.3 Ontology element

The class `OntologyElement` represents (identified) elements from an ontology.

Formal definition.

```
:OntologyElement
      a       owl:Class ;
      rdfs:subClassOf representation:FormalExpression , :KnowledgeResource ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty partof:isPartOf ;
                owl:someValuesFrom :Ontology
              ] .
```

The class `OntologyElement` is formally defined as sub-class of `:KnowledgeResource`, defined in section 2.2.1, and `representation:FormalExpression`, which is a class entity of the CP "information objects and representation languages". `representation:FormalExpression` represents any information object represented in a formal language, usually having a formal interpretation, and used to formally represent any entity. Additionally, an ontology element is part of an ontology; the class `Ontology` is defined in section 2.2.4. The relation `partof:isPartOf` is defined in the CP "part of", which represents a transitive relation expressing parthood between any entities, e.g. "brain is a part of the human body".

2.2.4 Ontology

In this context, an ontology is conceptualized as a (usually complex) typed formal expression, which can be realized either analogically or as a non-executable digital object. An ontology is a typed logical theory, i.e. its characteristic elements are named after a non-logical vocabulary. Ontology is taken here independently from a particular logical language, but excludes languages that do not have a formal semantics (e.g. folksonomies, lexicons, thesauri).

Formal definition.

```
:Ontology
      a        owl:Class ;
      rdfs:subClassOf representation:FormalExpression , :KnowledgeResource ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty topic:hasTopic ;
                owl:someValuesFrom topic:Topic
              ] .
```

In codolight, the class of ontologies is a sub-class of `representation:FormalExpression` and `:KnowledgeResource`, and it has some associated topic. The class `topic:Topic`, representing a subject, argument, domain, theme, subject area, etc., and the relation `topic:hasTopic` that associates anything that can have a topic with the topic itself, come from the CP *topic*.

2.2.5 Project

Here a project is conceptualized as it is intended in ontology engineering, softwate engineering tools, or in an open source platform such as Sourceforge. In the context of *codolight*, a project collects all data related to an ontology project.

Formal definition.

```
:Project
      a        owl:Class ;
      rdfs:subClassOf owl:Thing , :KnowledgeResource .
```

Formally, a project is a knowledge resource.

2.2.6 Ontology project

The class `OntologyProject` represents any project that aims to manage the lifecycle of an ontology. As all projects, ontology projects inherit the typical characteristics and constraints of projects: teams, persons, schedules, time, funding, strategical considerations, etc.

Formal definition.

```
:OntologyProject
      a        owl:Class ;
      rdfs:subClassOf description:Description ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty partof:hasPart> ;
                owl:someValuesFrom :DesignWorkflow
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :needs ;
                owl:someValuesFrom agentrole:Agent>
              ] ;
```

```
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :reuses ;
                owl:someValuesFrom :KnowledgeResource
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :needs ;
                owl:someValuesFrom :DesignFunctionality
              ] .
```

In codolight, an ontology project is a description that needs some agent and some design functionality (see section 2.2.10). A description, defined by the class `description:Description` in the CP *description*, represents a conceptualization; it can be thought also as a “descriptive context” that defines concepts in order to see a “relational context” out of a set of data or observations. The class `agentrole:Agent`, from the CP *agent role*, represents any agentive object, either physical, or social. Furthermore, an ontology project reuses some knowledge resource (see section 2.2.1) and includes, as its part, some design workflows (see section 2.2.7). Relations `needs`, and `reuses` are defined in sections 2.2.17 and 2.2.18, respectively.

2.2.7 Design workflow

The class `DesignWorkflow` represents any workflow that guides the interaction between ontology designers.

Formal definition.

```
:DesignWorkflow
      a       owl:Class ;
      rdfs:subClassOf description:Description ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :needs ;
                owl:someValuesFrom agentrole:Agent>
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :reuses ;
                owl:someValuesFrom :KnowledgeResource
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :needs ;
                owl:someValuesFrom :DesignFunctionality
              ] .
```

`DesignWorkflow` is defined in terms of the classes `agentrole:Agent` and `description:Description`. Specifically, a `DesignWorkflow` is a description that “needs” some agent, meaning that some agent has to be involved in its description. Additionally, a `DesignWorkflow` “needs” also some `DesignFunctionality` and “reuses” some `KnowledgeResource`. The concepts of `DesignFunctionality` and `KnowledgeResource` are defined in sections 2.2.10 and 2.2.1 respectively.

2.2.8 Design rationale

Design rationales are the principles behind the motivations underlying design making, involving design operations, patterns, and rational agents (the designers). An argument (see chapter 6) is usually an application of a design rationale.
Ontology design rationales typically include different types of semantics (extensional, intensional, linguistic, approximate, etc.), best practices, etc. For example, when argumenting about the subclass axiom: `EuropeanCountry rdfs:subClassOf (hasTerritory all (hasLocation Europe))`, someone can have a negative position motivated by the counterexample argument: "Turkey is a European country but has territories outside Europe". This argument is motivated by the design rationale: "extensional semantics", by which all instances of a class must have the properties asserted as axioms for the class. Notice that a different rationale, e.g. "approximate semantics", might support the axiom, although this may lead to inconsistencies when a crisp OWL reasoner is applied to the ontology.

Formal definition.

```
:DesignRationale
      a        owl:Class ;
      rdfs:subClassOf description:Description .
```

In the context of the *kernel* module, the class `DesignRationale` is defined as sub-class of `description:Description`.

2.2.9 Design solution

The class `DesignSolution` represents structural situations (states) of a (part of an) ontology, which include only formal expressions and their relations. For example, the occurrence of a `rdfs:subClassOf` axiom (which is an ontology entity) and its elements, as included in a design solution complying to the OWL Macro: "subClassOf an intersection between a Class and a Restriction", where OWL Macros are ontology design patterns. Notice that not all states of an ontology or its parts are design solutions.

Formal definition.

```
:DesignSolution
      a        owl:Class ;
            rdfs:subClassOf situation:Situation .
```

Such class is formally defined as sub-class of `situation:Situation`, a class of the CP *situation*, representing a view on a set of entities. It can be seen as a "relational context", reifying a relation. For example, the execution of a plan is a context including some actions executed by agents according to certain parameters and expected tasks to be achieved from a plan; a diagnosed situation is a context of observed entities that is interpreted on the basis of a diagnosis, etc.

2.2.10 Design functionality

An ontology design functionality is considered here as a task to be performed within an ontology project, e.g. an "evaluation" functionality. Not all functionalities are expected to be types of specific design operations i.e.they can involve more than one type, neither computational tasks i.e. functionalities that are implemented in a tool.

Formal definition.

```
:DesignFunctionality
      a       owl:Class ;
      rdfs:subClassOf taskrole:Task .
```

It can be noticed, in its formal definition, that `DesignFunctionality` is a sub-class of `taskrole:Task`, a class of the CP *task role* identifying a piece of work to be done or undertaken. A task is assigned to only roles, such as `UserType` instances.

2.2.11 Design tool

The class `DesignTool` represents a tool that implements ontology design functionalities. It has at least one input type, a user type, implements at least one functionality, with at least one interaction pattern.

Formal definition.

```
:DesignTool
      a       owl:Class ;
      rdfs:subClassOf objectrole:Object> .
```

In the context of this module, a design tool is formally defined as an object, in terms of the class `objectrole:Object` of CP *object role*.

2.2.12 Design operation

The class `DesignOperation` represents actions carried out to accomplish some required functionality. Design operations are the prominent entities in a design making situation. In the requirement-specification-implementation cycle, ideally, each design operation should be performed, assisted, or represented by a computational operation.

Formal definition.

```
:DesignOperation
      a       owl:Class ;
      rdfs:subClassOf taskexecution:Action .
```

In this context, the class `DesignOperation` is formally defined as sub-class of `taskexecution:Action`. The latter is a class defined in the CP *task execution* and represents events with at least one agent that participates in it, and that executes a task that typically is defined in a plan, workflow, project, etc.

2.2.13 Software engineering pattern

The class `SoftwareEngineeringPattern` represents general reusable solutions to commonly occurring problems in software design.

Formal definition.

```
:SoftwareEngineeringPattern
      a       owl:Class ;
      rdfs:subClassOf description:Description .
```

In this context, a software engineering pattern is formally defined as a description.

2.2.14 Interface object

The class `InterfaceObject` identifies the visual elements of a graphical user interface (GUI).

Formal definition.

```
:InterfaceObject
      a         owl:Class ;
      rdfs:subClassOf representation:IconicObject .
```

The class is formally described in this context as a sub-class of `representation:IconicObject`, a class defined in the CP *information objects and representation languages* representing information objects expressed in terms of some iconic language.

2.2.15 Interaction pattern

An interaction pattern is a software engineering pattern that describes how some configurations of interface object(s) can be implemented for a certain computational task and user type. Some examples are provided from an online library of patterns for interaction design[5].

Formal definition.

```
:InteractionPattern
      a         owl:Class ;
      rdfs:subClassOf :SoftwareEngineeringPattern .
```

The class `InteractionPattern` is formally described as a sub-class of `SoftwareEngineeringPattern`.

2.2.16 User type

The class `UserType` includes all possible types of users, which can be involved in the design of ontologies. Examples of its instances would be "ontology designer", "domain expert", etc. It can be compared to the notion of "actor" in UML.

Formal definition.

```
:UserType
      a         owl:Class ;
      rdfs:subClassOf objectrole:Role> .
```

The class `UserType` is formally defined in terms of the class `objectrole:Role` of the CP *object role*[PGGPF07], that encodes a concept that classifies any object e.g. physical, social, or mental object, or a substance.

2.2.17 Needs

The object property `needs` (inverse `isNeededBy`) represents the relation that holds between any entity that has to be involved in the description of either an ontology project, a design workflow, or a software engineering pattern.

[5] http://www.welie.com

Formal definition.

```
:needs
      a        owl:ObjectProperty ;
      rdfs:domain _:b1 ;
      rdfs:range owl:Thing ;
      rdfs:subPropertyOf descriptionandsituation:describes ;
      owl:inverseOf :isNeededBy .

_:b1  a        owl:Class ;
      owl:unionOf (:OntologyProject :DesignWorkflow :SoftwareEngineeringPattern) .
```

In codolight, the `needs` object property has the class `owl:Thing` as domain, and the union of classes `OntologyProject`, `DesignWorkflow`, and `SoftwareEngineeringPattern` as range.

2.2.18 Reuses

The object property `reuses` (inverse `isReusedBy` encodes the relation that can occur between an existing knowledge resource that is involved in the execution of either an ontology projects or design workflows as reusable object.

Formal definition.

```
:reuses
      a        owl:ObjectProperty ;
      rdfs:domain _:b2 ;
      rdfs:range :KnowledgeResource ;
      rdfs:subPropertyOf descriptionandsituation:describes ;
      owl:inverseOf :isReusedBy .

_:b2  a        owl:Class ;
      owl:unionOf (:OntologyProject :DesignWorkflow) .
```

In codolight, the object property `reuses` has the class `KnowledgeResource` as domain and the union of classes `OntologyProject` and `DesignWorkflow` as range.

Chapter 3

The *Data* module

The *Data*[1] module of *codolight* contains a minimal set of classes and properties to represent the data, or *knowledge resources* that are typically involved in ontology projects: ontologies and their entities, mappings, modules, non-ontological resources, etc.
Data-related aspects of ontology design are mandatory, because designing and applying ontologies without them is not possible.
In [CGL+06] the vocabulary for talking about data was sketchy. On the contrary, codolight tries to extend the support for ontology design data description, which is needed by its applications in NeOn [PPG+09]. In chapter 11, several alignments to other vocabularies describing knowledge resources add scope to this module.
The key classes and properties in *codolight data* module are illustrated by means of a simple labeled graph 3.2. The central notion is `codkernel:KnowledgeResource`. Knowledge resources are input or output data for design tools, workflows, or functionalities; each class of knowledge resource has an associated knowledge type; some of them are represented in a logical language. Notable subclasses of `codkernel:KnowledgeResource` are: `Ontology`, `OntologyElement`, `KOS`, `NetworkOfOntologies`, etc.
For example, by using entities defined in this module, it is possible to relate an ontology element such as the class `foaf:Person` to the ontology it belongs to e.g. "the FOAF ontology".
An example of a codolight description for knowledge resources is given in figure 3.1. The picture shows a fragment of the *codolight*-based description of the "Open Rating System" tool as given in [PPG+09]. The tool is extensively described in [SCd+09]. The classes `Review` and `ReviewRank` are formally described as sub-classes of the class `Annotation`. All of them are subclasses of `KnowledgeResource`. Furthermore, a review rank is about a review, which in turn is about a certain ontology.
For additional, more detailed examples the reader can refer to [PPG+09].
In section 3.2, entities defined in this module are described in detail, while next section 3.1 describes CPs reused in *codolight data* module. Axioms added to *kernel* entities are described in section 3.3.

3.1 Patterns reused in *codolight data* module

The *data* module of *codolight* has been built by reusing the following Content Ontology Design Patterns (CPs) as building blocks [PG08, PGGPF07].

Collection. This CP, also called "membership", aims at representing any container for entities that share one or more common properties and relations between the container and its entities. E.g. "stone objects", "the nurses", "the Louvre Aegyptian collection". A collection is not a logical class: a collection is a first-order entity, while a class is a second-order one. A relation between collections and entities is a non-transitive

[1] http://www.ontologydesignpatterns.org/cpont/codo/coddata.owl

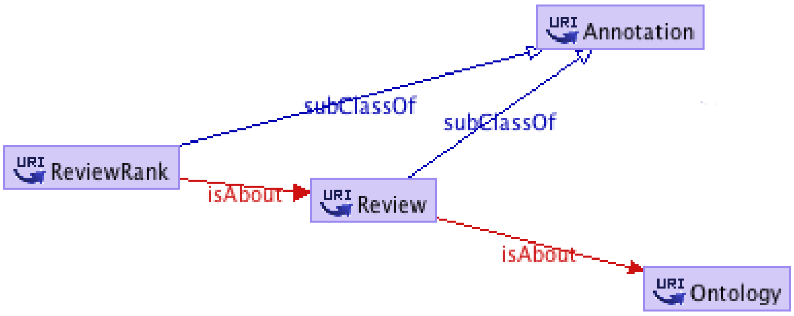

Figure 3.1: An example of a codolight description for knowledge resources in the context of the “Open Rating System” tool as described in [PPG+09].

relation (opposed to the “part of relation” described later in this section), e.g. “my collection of saxophones includes an old Adolphe Sax original alto” (i.e. my collection has member an Adolphe Sax alto).

Part of. This CP aims at representing a transitive relation expressing parthood between any entities, e.g. “the human body has a brain as part”.

3.2 Entities of *codolight data* module

The following entities are defined in this module.

3.2.1 Ontology mapping

The class `OntologyMapping` represents any set of axioms that include ontology elements from two different ontologies.

Formal definition.

```
:OntologyMapping
      a        owl:Class ;
      rdfs:subClassOf collectionentity:Collection , codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:onProperty intensionextension:isAbout ;
                owl:someValuesFrom :NetworkOfOntologies
              ] ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:onProperty collectionentity:hasMember ;
                owl:someValuesFrom :OntologyAxiom
              ] ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :OntologyMappingKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

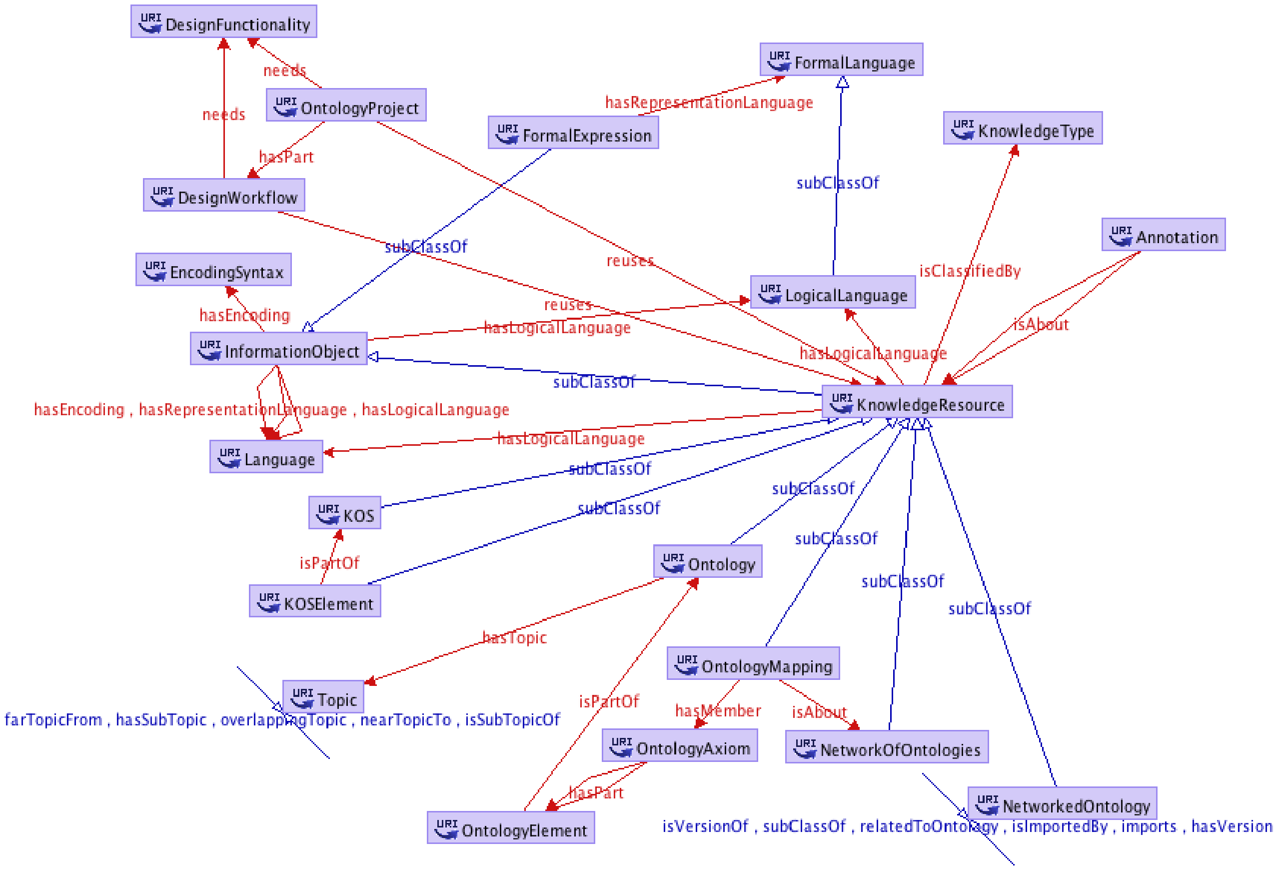

Figure 3.2: A simple graph of ontology elements from *codolight data* module.

In codolight, an ontology mapping is a knowledge resource about some network of ontologies. It is also a collection including some ontology axiom. Furthermore, the intensional meaning of the class `OntologyMapping` is encoded by the knowledge type `OntologyMappingKType`, formally described by the following axiom as an instance of the class textttcodkernel:KnowledgeType:

```
:OntologyMappingKType
      a        codkernel:KnowledgeType ;
```

3.2.2 Ontology module

The class `OntologyModule` represents ontologies that are considered modularly, i.e. within a compositional architecture. Each module in a compositional network is supposed to cover a domain aspect, a task, etc.

Formal definition.

```
:OntologyModule
      a        owl:Class ;
      rdfs:subClassOf :DataStructure ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :OntologyModuleKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

In codolight, an ontology module is a data structure; an ontology module is classified by the knowledge type `OntologyModuleKType`, formally described by the following axiom:

```
:OntologyModuleKType
      a       codkernel:KnowledgeType ;
```

3.2.3 Networked ontology

The class `NetworkedOntology` represents ontologies that are member of an ontology network. In practice, an ontology is networked when it has some relation to other ontologies. An ontology can be networked because it started a relation after its creation, or because it emerges from a (qualified) relation between other ontologies, as in the case of qualified ontology networks. In the second case, it can be called a "inherently networked" ontology. For example, two ontologies O_1 and O_2 that have an equivalence relation, are networked ontologies. More interestingly, an ontology O_3 that has been designed by reusing components from two ontologies O_4 and O_5 is also a (distributed) networked ontology, like *codolight*. Modules and content design patterns are also networked ontologies.

Formal definition.

```
:NetworkedOntology
      a       owl:Class ;
      rdfs:subClassOf codkernel:Ontology ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :NetworkedOntologyKType ;
                owl:onProperty classification:isClassifiedBy
              ] ;
      owl:equivalentClass
              [ a       owl:Restriction ;
                owl:onProperty :isPartOfNetwork ;
                owl:someValuesFrom :NetworkOfOntologies
              ] .
```

In codolight, a networked ontology is an ontology belonging to a network of ontologies, meaning that it is a component part of it. Furthermore, the class of networked ontologies is classified by the knowledge type `NetworkedOntologyKType`, formally described by the following axiom:

```
:NetworkedOntologyKType
      a       codkernel:KnowledgeType ;
      rdfs:comment "The NetworkedOntology knowledge type."^^xsd:string ;
      rdfs:label "Networked ontology KType"@en .
```

3.2.4 Network of ontologies

The class `NetworkOfOntologies` represents networks of ontologies (or ontology network). A network of ontologies is a set of ontologies with a specified unifying criterion (description), provided by (reifying a) relation such as "being a version of", "imports", "identical to", "equivalent to", "is clone of", "mapped to", "contains module", etc.

Formal definition.

```
:NetworkOfOntologies
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :NetworkOfOntologiesKType ;
                owl:onProperty classification:isClassifiedBy
              ] ;
     owl:equivalentClass
              [ a        owl:Restriction ;
                owl:maxCardinality "2"^^xsd:int ;
                owl:onProperty :hasNetworkedOntology
              ] ;
      owl:equivalentClass
              [ a        owl:Restriction ;
                owl:allValuesFrom codkernel:Ontology ;
                owl:onProperty :hasNetworkedOntology
              ] .
```

A network of ontologies is a knowledge resource composed of at least two ontologies. Also ontology elements can be part of an ontology network, however they do not characterize it, while the object property `hasNetworkedOntology` (cf. sect. 3.2.16) identifies only ontologies that are part of the network. The intensional meaning of this class is represented by the knowledge type `NetworkOfOntologiesKType` formally described by the following axiom:

```
:NetworkOfOntologiesKType
      a        codkernel:KnowledgeType ;
```

3.2.5 Ontology library

A member of the class `OntologyLibrary` represents an ontology repository, a collection of ontologies. An example can be a simple file system-based repository of ontologies.

Formal definition.

```
:OntologyLibrary
      a        owl:Class ;
      rdfs:subClassOf collection:Collection ;
      owl:equivalentClass
              [ a        owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty collectionentity:hasMember
              ] ;
      owl:equivalentClass
              [ a        owl:Restriction ;
                owl:allValuesFrom codkernel:Ontology ;
                owl:onProperty collectionentity:hasMember
              ] .
```

In codolight, the class `OntologyLibrary` is a sub-class of `collection:Collection` that has only ontologies as members, and has at least one member.

3.2.6 Ontology axiom

The class `OntologyAxiom` represents axioms from within an ontology.

Formal definition.

```
:OntologyAxiom
      a       owl:Class ;
      rdfs:subClassOf codkernel:OntologyElement ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty partof:hasPart ;
                owl:someValuesFrom codkernel:OntologyElement
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :OntologyAxiomKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

The class `OntologyAxiom` includes ontology elements composed of some other ontology element. Furthermore, the intensional meaning of this class is encoded by the knowledge type `OntologyAxiomKType`, as formally defined by the following axiom:

```
:OntologyAxiomKType
      a       codkernel:KnowledgeType ;
```

3.2.7 Ontology topic

The topic to be covered by an ontology.

Formal definition.

```
:OntologyTopic
      a       owl:Class ;
      rdfs:subClassOf topic:Topic .
```

The class `OntologyTopic` is formally described as a sub-class of `topic:Topic`. An ontology topic is related to an ontology by the object property `topic:hasTopic`, and every ontology has at least one ontology topic (see section 3.3).

3.2.8 Data structure

Any data structure, including databases, schemas, lexica, knowledge organizations systems, etc.

Formal definition.

```
:DataStructure
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :DataStructureKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

The class `DataStructure` is defined as sub-class of `codkernel:KnowledgeResource`. Furthermore, the class of data structures is intensionally represented by the knowledge type `DataStructureKType`, formally defined as follows:

```
:DataStructureKType
      a        codkernel:KnowledgeType ;
```

3.2.9 Knowledge Organization System (KOS)

Any knowledge organization system such as: thesauri, terminologies, classification schemes, subject hierarchies, etc.

Formal definition.

```
:KOS  a        owl:Class ;
      rdfs:subClassOf :DataStructure ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :KOSKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

In codolight, a KOS is defined as a data structure. The intensional meaning of the class `KOS` is represented by the knowledge type `KOSKType`, formally defined by the following axiom:

```
:KOSKType
      a        codkernel:KnowledgeType ;
```

3.2.10 KOS element

An (identified) element from a KOS.

Formal definition.

```
:KOSElement
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :KOSElementKType ;
                owl:onProperty classification:isClassifiedBy
```

```
                ] ;
        rdfs:subClassOf
                [ a        owl:Restriction ;
                  owl:onProperty partof:isPartOf ;
                  owl:someValuesFrom :KOS
                ] .
```

A KOS element is formally described as a knowledge resource that is part of some KOS. The intensional meaning of the class `KOSElement` is represented by the knowledge type `KOSElementKType`, formally described by the following axiom:

```
:KOSElementKType
        a        codkernel:KnowledgeType ;
```

3.2.11 Logical language

The logical language, in which a knowledge resource is expressed.

Formal definition.

```
:LogicalLanguage
        a        owl:Class ;
        rdfs:subClassOf resentation:FormalLanguage .
```

In codolight, a logical language is described as a formal language.

3.2.12 Encoding syntax

The syntax used for encoding a knowledge resource or in general a logical language; e.g. OWL-RDF.

Formal definition.

```
:EncodingSyntax
        a        owl:Class ;
        rdfs:subClassOf representation:Language .
```

In codolight, an encoding syntax is a special kind of language.

3.2.13 Annotation

Any knowledge resource used to talk about another existing knowledge resource.

Formal definition.

```
:Annotation
        a        owl:Class ;
        rdfs:subClassOf codkernel:KnowledgeResource ;
        rdfs:subClassOf
                [ a        owl:Restriction ;
                  owl:hasValue :AnnotationKType ;
                  owl:onProperty classification:isClassifiedBy
```

```
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty intensionextension:isAbout ;
                owl:someValuesFrom codkernel:KnowledgeResource
              ] .
```

In codolight, an annotation is a knowledge resource that is about some other knowledge resource. This class is associated with the knowledge type `AnnotationKType`, formally described by the following axiom:

```
:AnnotationKType
      a       codkernel:KnowledgeType ;
```

3.2.14 Query

A query is a request for information from a database, a knowledge base, a search engine, etc.

Formal definition.

```
:Query
      a       owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :QueryKType ;
                owl:onProperty classification:isClassifiedBy
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty resentation:hasRepresentationLanguage
              ] ;
```

A query is described as a knowledge resource that has at least one representation language. It is associated with the knowledge type `QueryKType`, formally defined by the following axiom:

```
:QueryKType
      a       codkernel:KnowledgeType ;
```

3.2.15 Rule

A rule is an axiom that is asserted independently from a specific ontology element, i.e. not within the axioms that are proper to that element. Depending on the particular “style” of a logical language and its reasoning system, rules (and axioms in general) can be considered within or outside other elements’ characterization.

Formal definition.

```
:Rule
      a       owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
```

```
            [ a       owl:Restriction ;
              owl:minCardinality "1"^^xsd:int ;
              owl:onProperty :hasLogicalLanguage
            ] ;
      rdfs:subClassOf
            [ a       owl:Restriction ;
              owl:hasValue :RuleKType ;
              owl:onProperty classification:isClassifiedBy
            ] .
```

In this model, a rule is a knowledge resource that has at least one logical representation language. Furthermore, the intensional meaning of the class `Rule` is represented by the knowledge type `RuleKType`, formally described by the following axiom:

```
:RuleKType
      a       codkernel:KnowledgeType ;
```

3.2.16 Has networked ontology

A non transitive relation between a network of ontologies and its component parts, which are networked ontologies.

Formal definition.

```
:hasNetworkedOntology
      a       owl:ObjectProperty ;
      rdfs:domain :NetworkOfOntologies ;
      rdfs:range codkernel:Ontology ;
      rdfs:subPropertyOf partof:hasPart ;
      owl:inverseOf :isPartOfNetwork .
```

The object property `hasNetworkedOntology` (inverse `isPartOfNetwork`) is formally described as a sub-property of `partof:hasPart`. As such, it is non-transitive but implies the transitive part of relation between its related entities. The class `NetworkOfOntologies` is its domain, while its range is the class `codkernel:Ontology`.

3.2.17 Has encoding

A relation between a knowledge resource e.g. an ontology, or a logical language e.g., Description Logics, and a syntactic language e.g. RDF-XML, or N3.

Formal definition.

```
:hasEncoding
      a       owl:ObjectProperty ;
      rdfs:domain
            [ a       owl:Class ;
              owl:unionOf (codkernel:KnowledgeResource :LogicalLanguage)
            ] ;
      rdfs:range :EncodingSyntax ;
      rdfs:subPropertyOf representation:hasRepresentationLanguage ;
      owl:inverseOf :isEncodingOf .
```

In codolight, this relation is described by a sub-property of representation:hasRepresentationLanguage. Its domain is the union of the classes codkernel:KnowledgeResource and LogicalLanguage, while its range is the class EncodingSyntax.

3.2.18 Has logical language

A relation between a knowledge resource e.g., an ontology, and a logical language e.g., OWL-DL.

Formal definition.

```
:hasLogicalLanguage
      a        owl:ObjectProperty ;
      rdfs:domain codkernel:KnowledgeResource ;
      rdfs:range :LogicalLanguage ;
      rdfs:subPropertyOf representation:hasRepresentationLanguage ;
      owl:inverseOf :isLogicalLanguageOf .
```

In codolight, this relation is described by a sub-property of representation:hasRepresentationLanguage. Its domain is the class codkernel:KnowledgeResource, while its range is the class LogicalLanguage.

3.2.19 Related to ontology

Any relation between two (networked) ontologies.

Formal definition.

```
:relatedToOntology
      a        owl:ObjectProperty , owl:SymmetricProperty ;
      rdfs:domain :NetworkedOntology ;
      rdfs:range :NetworkedOntology ;
      owl:inverseOf :relatedToOntology .
```

This relation is formally described as a symmetric object property between networked ontologies.

3.2.20 Has version

A relation between two different versions of an ontology. This assumes that an ontology abstracts from its versions.

Formal definition.

```
:hasVersion
      a        owl:ObjectProperty ;
      rdfs:subPropertyOf :relatedToOntology ;
      owl:inverseOf :isVersionOf .
```

In codolight, this relation is described by the object property hasVersion (inverse isVersionOf), a specialization of relatedToOntology.

3.2.21 Is about ontology project

A relation between a project and the ontology project it is about.

Formal definition.

```
:isAboutOntologyProject
      a       owl:ObjectProperty ;
      rdfs:domain :Project ;
      rdfs:range codkernel:OntologyProject ;
      rdfs:subPropertyOf intensionextension:isAbout .
```

3.3 Axioms extending *kernel* entities

The *codolight kernel* module defines four classes that are further formally described in the *data* module. Additionally, a class from the CP *information objects and representation languages* is formally characterized in this context. In the following paragraphs such additional axioms are shown.

Extending axioms for `codkernel:KnowledgeResource` In the *data* module, the class of knowledge resources is further described formally by the following axioms.

```
codkernel:KnowledgeResource
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty classification:isClassifiedBy ;
                owl:someValuesFrom codkernel:KnowledgeType
              ] .
```

Extending axioms for `codkernel:Project` A Project is a data structure.

```
codkernel:Project
      rdfs:subClassOf :DataStructure .
```

Extending axioms for `representation:LinguisticObject` The class `representation:LinguisticObject` is classified by the `LinguisticKType` knowledge type.

```
representation:LinguisticObject
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :LinguisticKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

A knowledge resource is classified by only knowledge types, meaning that the intensional meaning of knowledge resource classes is represented by knowledge types.

Extending axioms for codkernel:Ontology

```
codkernel:Ontology
      rdfs:subClassOf :DataStructure;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty topic:hasTopic
              ] ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:allValuesFrom :OntologyTopic ;
                owl:onProperty topic:hasTopic
              ] ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :OntologyKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

An ontology is formally described as a data structure, covering at least one ontology topic, and classified by the knowledge type `OntologyKType`, which is formally described by the following axiom:

```
:OntologyKType
      a        codkernel:KnowledgeType ;
```

Extending axioms for codkernel:OntologyElement

```
codkernel:OntologyElement
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :OntologyElementKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

The class of ontology elements is classified by the knowledge type `OntologyElementKType`, which in turn is formally described by the following axiom:

```
:OntologyElementKType
      a        codkernel:KnowledgeType ;
```

Extending axioms for codkernel:KnowledgeType

```
codkernel:KnowledgeType
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:allValuesFrom codkernel:KnowledgeResource ;
                owl:onProperty classification:classifies
              ] .
```

A knowledge type classifies only knowledge resources.

Extending axioms for `representation:LinguisticObject` The class `representation:LinguisticObject` is classified by the `LinguisticKType` knowledge type.

```
representation:LinguisticObject
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :LinguisticKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

The knowledge type `LinguisticKType` is formally defined by the following axiom:

```
:LinguisticKType
      a       codkernel:KnowledgeType ;
```

Chapter 4

The *Projects* module

The *Projects*[1] module of *codolight* contains a minimal set of classes and properties to represent the classes of design project-related entities, and their relations.

Project aspects of ontology design are not mandatory, since, although each activity of ontology design can be seen in the context of a project, not all design activities are contextualized with specific plans. However, specific software support for ontology project execution has appeared with recent Eclipse-based tools, and with approaches to ontology design that are analogous to software projects, specially in the open source realm.

In [NeOnD2.1.1] the vocabulary for talking about projects was more sophisticated than the one proposed here.

The key classes and properties in *codolight projects* module are illustrated by means of a simple labeled graph 4.1.

The central notion is `codkernel:OntologyProject`. Ontology projects are abstract descriptions ("plans") of actual projects executions (that are expected to satisfy the project), in which designers typically envisage the resources and procedures needed to achieve a certain goal. Ontology projects need design functionalities and agents, have ontologies and other knowledge resources as intended output, have design workflows as parts, and are described by project descriptions.

Consider for example the ontology project that aims at producing a network of ontologies for the FSDAS system of the NeOn case study in the fishery domain[2]. The FSDAS ontology project can be described as an instance of the class `OntologyProject`, and it can be associated with some report describing the plan and expected results of the project, i.e. it is `expressed by` a `ProjectDescription`, has typical workflows that FAO experts are used to perform for taking decisions i.e. `DesignWorkflows`, `reuses` XSD `KnowledgeResources` that `need` a reengineering `DesignFunctionality`, etc. Moreover, several (digital) `Projects` have been created on NeOn Toolkit and other tools in order to maintain the workspace of ontologies, annotations, documents, diagrams, etc. that are in the context of the `OntologyProjectExecution`.

In section 4.2, entities defined in this module are described in detail, while next section 4.1 describes CPs reused in *codolight projects* module. Axioms added to *kernel* entities are described in section 4.3.

4.1 Patterns reused in *codolight projects* module

Place. This CP, also called "location", aims at representing locations and the relations between things and their locations. Location is here intended in a very generic sense: a political geographic entity (Roma, Lesotho), a location determined by the presence of other entities ("the area close to Roma"), pivot events or signs ("the area where the helicopter fell"), complements of other entities ("the area under the table"), as well as physical objects conceptualized as locations as their main identity criterion ("the territory of Italy"). In

[1] http://www.ontologydesignpatterns.org/cpont/codo/codprojects.owl

[2] http://ontologydesignpatterns.org/cp/owl/fsdas/fsdasnetwork.owl

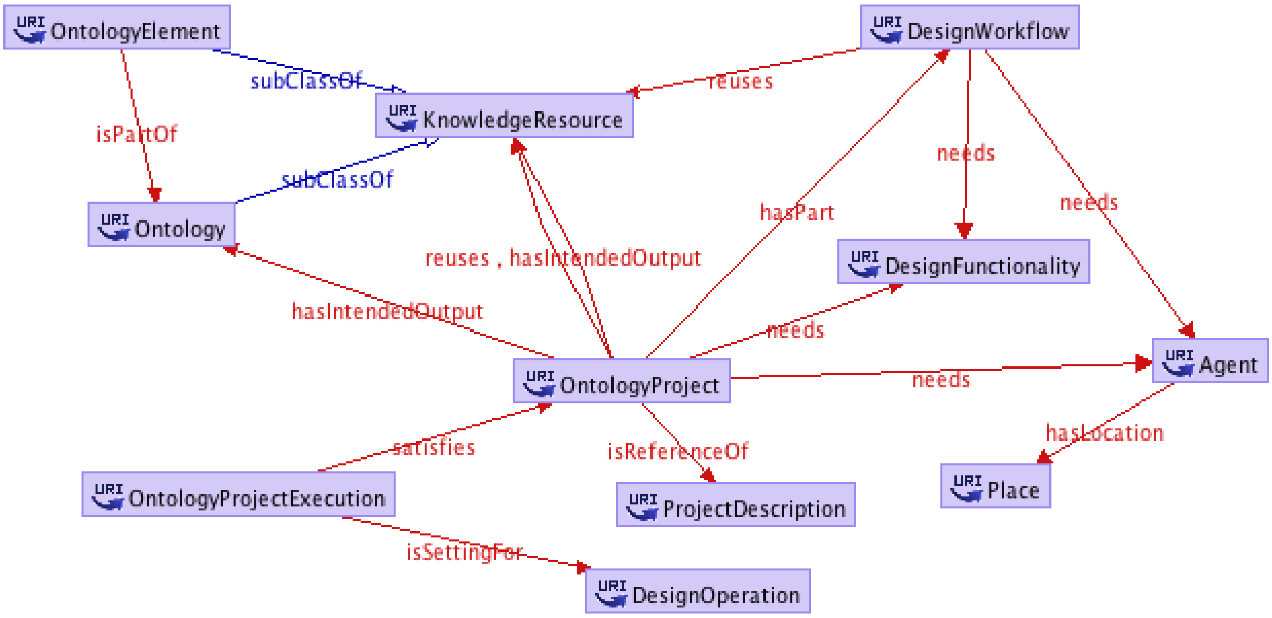

Figure 4.1: A simple graph of ontology elements from *codolight projects* module.

this generic sense, a place is an "approximate", relative location. Formally, a Place is defined by the fact of having something located in it; a place is located in itself. The relation between entities and their location is defined as a generic, relative localization, holding between any entities. E.g. "the cat is on the mat", "Omar is in Samarcanda", "the wound is close to the femural artery".

4.2 Entities of *codolight projects* module

4.2.1 Project description

A project description is an information object describing an ontology project. For example, a document describing an ontology project in natural language.

Formal definition.

```
:ProjectDescription
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :ProjectDescriptionKType ;
                owl:onProperty classification:isClassifiedBy
              ] ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:onProperty intensionextension:isAbout ;
                owl:someValuesFrom codkernel:OntologyProject
              ] .
```

In codolight, a project description is a knowledge resource about some ontology project. It is classified by the knowledge type `ProjectDescriptionKType` formally described by the following axiom.

```
:ProjectDescriptionKType
      a       codkernel:KnowledgeType ;
      rdfs:label "Project description KType"^^xsd:string .
```

4.2.2 Ontology project execution

An execution of an ontology project (its schema). Execution can be more or less precisely specified according to the constraints, preferences, and resources declared in the ontology project (schema).

Formal definition.

```
:OntologyProjectExecution
      a       owl:Class ;
      rdfs:subClassOf situation:Situation ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty situation:isSettingFor ;
                owl:someValuesFrom codkernel:DesignOperation
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty descriptionandsituation:satisfies ;
                owl:someValuesFrom codkernel:OntologyProject
              ] .
```

In codolight, an ontology project execution is a situation that satisfies some ontology project and that includes some design operation in its setting.

4.2.3 Has intended output

A relation between an ontology project and the knowledge resources it is supposed to produce.

Formal definition.

```
:hasIntendedOutput
      a       owl:ObjectProperty ;
      rdfs:domain codkernel:OntologyProject ;
      rdfs:range codkernel:KnowledgeResource ;
      rdfs:subPropertyOf descriptionandsituation:describes ;
      owl:inverseOf :isIntendedOutputOf .
```

In codolight, this relation is described by the object property `hasIntendedOutput` (inverse `isIntendedOutputOf`), sub-property of `descriptionandsituation:describes` from the CP *description and situation*. The property has the class `codkernel:OntologyProject` as domain, and the class `codkernel:KnowledgeResource` as its range.

4.3 Axioms extending *kernel* entities

The *kernel* module defines two classes that are further characterized in the *codolight projects* module by the following axioms.

Extending axioms for `codkernel:Project` A project is in created in the context of some ontology project execution.

```
codkernel:Project
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty situation:hasSetting ;
                owl:someValuesFrom :OntologyProjectExecution
              ] .
```

Extending axioms for `codkernel:OntologyProject` An ontology project is the reference of some project description and has an ontology as intended output.

```
codkernel:OntologyProject
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty intensionextension:isReferenceOf ;
                owl:someValuesFrom :ProjectDescription
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :hasIntendedOutput ;
                owl:someValuesFrom codkernel:Ontology
              ] .
```

Extending axioms for `codkernel:DesignWorkflow` A design workflow is a description having some ontology project as its part.

```
codkernel:DesignWorkflow
      rdfs:subClassOf description:Description ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty partof:isPartOf ;
                owl:someValuesFrom codkernel:OntologyProject
              ] .
```

Chapter 5

The *Workflows* module

The *Workflow*[1] module of *codolight* contains a minimal set of classes and properties to represent workflows from within ontology projects: collaborative workflows, accountable agents, need for an agent, etc.
As for all aspects of ontology design, except *data-related* ones, workflow aspects of ontology design are not mandatory. They are usually taken into account in the following cases:

- when an ontology project is explicitly created and reusable plans are envisaged for its execution
- when a method that requires several, interdependent tasks is used
- when a team or an informal group needs collaboration procedures
- when an individual needs to keep track of her own activities in the development of an ontology or a network of ontologies

In [CGL+06] a detailed analysis of collaborative aspects and a more in-depth vocabulary for talking about them has been presented. The support in codolight is however basic, in order to provide a lightweight entry-level to the description of ontology design workflows.
In [SNTM08], a project for adding workflow support to Protégé, partly based on our previous work on C-ODO, is described, and a preliminary ontology is also sketched. In http://www.ontologydesignpatterns.org/cpont/codo/protege2codo.owl, and alignment between that ontology and codolight is provided (section 11.6).
The key classes and properties in codworkflows.owl are illustrated by means of a simple labeled graph 5.1. The central notion is `codkernel:DesignWorkflow`. Design workflows are part of some ontology project, include at least one functionality, reuse at least one knowledge resource, and need at least one agent. An important subclass of `codkernel:DesignWorkflow` is `CollaborativeWorkflow`. Fully computational workflows (like Protégé ones) are included in this class.
Consider for example the codolight model of the CiceroWiki tool presented in [PPG+09]. The model has axioms that assert e.g. that CiceroWikiWorkflow is a `CollaborativeWorkflow`, that it `includesFunctionalities` such as `ProvideArgument`, `ProposeSolution`, `TakeDecision`, etc., that the task `DiscussDesignRationale precedes DecideOnSolution`, etc. In section 5.2, entities defined in this module are described in detail, while next section 5.1 describes CPs reused in *codolight workflows* module. Axioms added to *kernel* entities are described in section 5.3.

5.1 Patterns reused in *codolight workflows* module

The *workflows* module of *codolight* has been built by reusing the following CP as building block [PG08, PGGPF07].

[1] http://www.ontologydesignpatterns.org/cpont/codo/codworkflows.owl

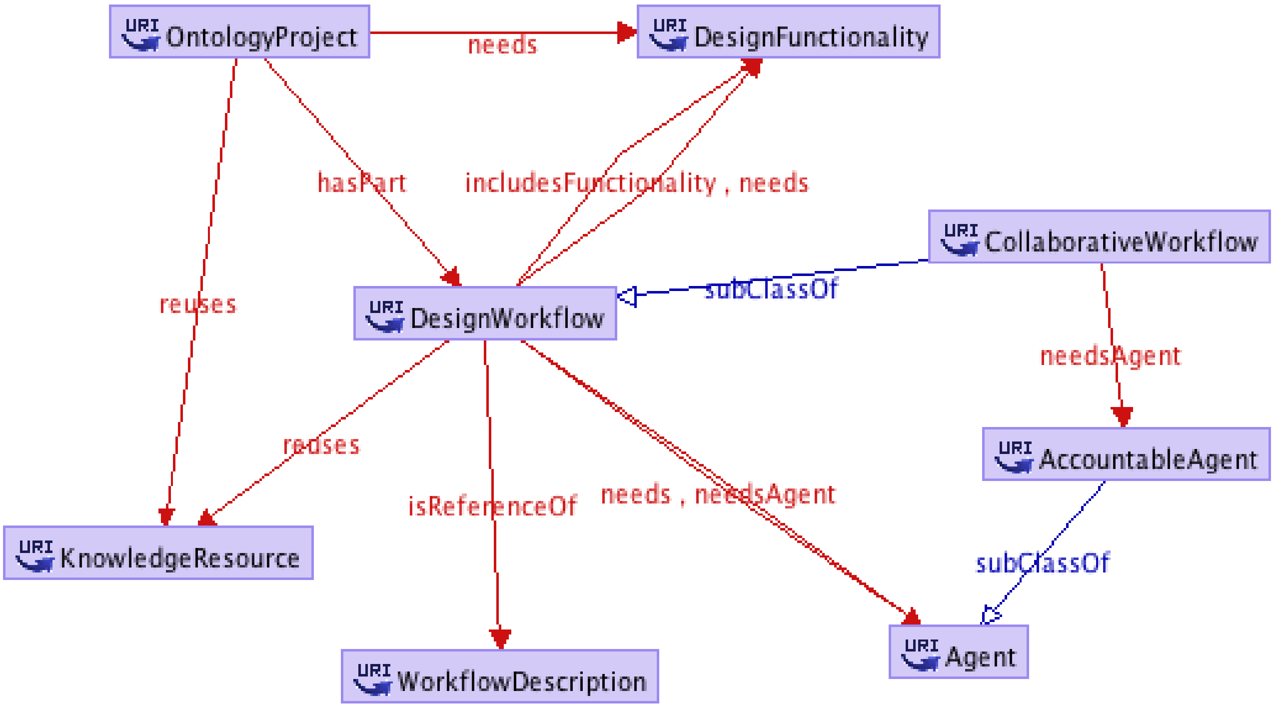

Figure 5.1: A simple graph of ontology elements from *codolight workflows* module.

Sequence. This CP is aimed at representing sequence schemas. It defines the notion of transitive and intransitive precedence and their inverses. It can then be used between tasks, processes, time intervals, spatially locate objects, situations, etc. It is also referred to as "ordering" or "precedence". Details about this CP can be found in the semantic wiki of ontology design patterns[2] or in [PGGPF07].

5.2 Entities of *codolight workflow* module

The following entities are defined in this module.

5.2.1 Workflow description

A workflow description is a set of entities that are involved in the definition of a workflow schema and the relations between them. Typically a workflow description includes knowledge resources, tasks, roles, etc.

Formal definition.

```
:WorkflowDescription
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:onProperty intensionextension:isAbout> ;
                owl:someValuesFrom codkernel:DesignWorkflow
              ] ;
      rdfs:subClassOf
```

[2] http://www.ontologydesignpatterns.org

```
                [ a       owl:Restriction ;
                  owl:hasValue :WorkflowDescriptionKType ;
                  owl:onProperty classification:isClassifiedBy>
                ] .
```

In codolight, a workflow description is a knowledge resource about some design workflow (see section 2.2.7), that is classified by a specific knowledge type i.e. `WorkflowDescriptionKType`. Codolight defines a set of knowledge types that represent the intensional meaning of knowledge resources, `WorkflowDescriptionKType` is one of them. The aim of such entities is explained in chapter 3.

5.2.2 Collaborative workflow

The class `CollaborativeWorkflow` represents design workflows, where all main participating agents are accountable for the sake of the project, where the workflow is executed. Note that the concept of design workflow is defined in the *kernel* module, but it is in this context further characterized by additional axioms as shown in section 5.3 later in this chapter.

Formal definition.

```
:CollaborativeWorkflow
      a       owl:Class ;
      rdfs:subClassOf codkernel:DesignWorkflow ;
      rdfs:subClassOf
                [ a       owl:Restriction ;
                  owl:onProperty :needsAgent ;
                  owl:someValuesFrom :AccountableAgent
                ] ;
      rdfs:subClassOf
                [ a       owl:Restriction ;
                  owl:onProperty description:usesConcept ;
                  owl:someValuesFrom codkernel:DesignFunctionality
                ] ;
      rdfs:subClassOf
                [ a       owl:Restriction ;
                  owl:minCardinality "2"^^xsd:int ;
                  owl:onProperty :needsAgent
                ] .
```

In codolight, a collaborative workflow is a design workflow that needs at least two accountable agents, and uses some design functionality.

5.2.3 Accountable agent

The class `AccountableAgent` represent any rational agent that adopts the goal of the collaborative workflow where it is needed (this axiom cannot be expressed in OWL).

Formal definition.

```
:AccountableAgent
      a       owl:Class ;
      rdfs:subClassOf agentrole:Agent ;
      owl:disjointWith :NonAccountableAgent .
```

In codolight, the class `AccountableAgent` is defined as sub-class of `agentrole:Agent` and is disjoint with `NonAccountableAgent`.

5.2.4 NonAccountableAgent

The class `NonAccountableAgent` represents rational agents that do not necessarily adopt the goal of the collaborative workflow in which they are involved (this axiom cannot be expressed in OWL).

Formal definition.

```
:NonAccountableAgent
      a       owl:Class ;
      rdfs:subClassOf agentrole:Agent ;
      owl:disjointWith :AccountableAgent .
```

In codolight, the class `NonAccountableAgent` is defined as sub-class of `agentrole:Agent` and is disjoint with `AccountableAgent`.

5.2.5 Needs agent

The object property `needsAgent` relates a design workflow to agents needed in order to describe it. It is a more specific property with respect to the `codkernel:needs` property. By this relation a design workflow or an ontology project are specifically associated with an agent rather than a generic thing.

Formal definition.

```
:needsAgent
      a       owl:ObjectProperty ;
      rdfs:domain _:b1 ;
      rdfs:range agentrole:Agent> ;
      rdfs:subPropertyOf codkernel:needs ;
      owl:inverseOf :isAgentNeededBy .

_:b1  a       owl:Class ;
      owl:unionOf (codkernel:OntologyProject codkernel:DesignWorkflow) .
```

Formally, the object property `needsAgent` (inverse `isAgentNeededBy`) specializes the `codkernel:needs` object property and its range is the class `agentrole:Agent`.

5.2.6 Is involved in the design of

This relation associates an agent with a knowledge resource it contributed to design.

Formal definition.

```
:isInvolvedInTheDesignOf
      a       owl:ObjectProperty ;
      rdfs:domain agentrole:Agent ;
      rdfs:range codkernel:KnowledgeResource ;
      owl:inverseOf :isInvolvedInDesignOperationsBy .
```

The object property `isInvolvedInTheDesignOf` (inverse `isInvolvedInDesignOperationsBy`) formally holds between the class `agentrole:Agent`, its domain and the class `codkernel:KnowledgeResource`, its range.

5.2.7 Includes functionality

The `includesFunctionality` object property relates design workflows and the functionalities involved in their description.

Formal definition.

```
:includesFunctionality
      a       owl:ObjectProperty ;
      rdfs:domain codkernel:DesignWorkflow ;
      rdfs:range codkernel:DesignFunctionality ;
      rdfs:subPropertyOf description:usesConcept ;
      owl:inverseOf :isFunctionalityIncludedIn .
```

In codolight, this object property is defined as the specialization of `description:usesConcept`. Its domain is the class `codkernel:DesignWorkflow` and its range is the class `codkernel:DesignFunctionality`.

5.3 Axioms extending *kernel* entities

The *codolight workflows* module is mainly about design workflows and its related entities. In fact, in this module, typical entities that depend on and are in general related to design workflows are defined. The class `codkernel:DesignWorkflow` is defined in the *kernel* module in order to comply to the design choices of building *codolight* with the architectural shape of a *corolla* (see section 1.2). In this module, such class is further described by the following additional axioms.

```
codkernel:DesignWorkflow
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty intensionextension:isReferenceOf ;
                owl:someValuesFrom :WorkflowDescription
              ] .
```

A design workflow is the reference of some workflow description, meaning that it is associated to some description that in turn can be used e.g. by some tools in order to support its execution. Reference here is meant as the relation that holds between information objects and any entity (including information objects). It can be used to talk about e.g. entities that are references of proper nouns: “the proper noun ’Leonardo da Vinci’ is about the person Leonardo da Vinci”, as well as to talk about sets of entities that can be described by a common noun: “the common noun ’person’ is about the set of all persons in a domain of discourse”. In this case, given a design workflow, there is some workflow description that refers to it.

Chapter 6

The *Argumentation* module

The *Argumentation*[1] module of *codolight* contains a minimal set of classes and properties to represent the classes of argumentation entities and their relations.
Argumentation aspects of ontology design are not mandatory, since, although some form of argumentation is always present in ontology design, only recently annotation of the discussions and decisions has become an object of study in ontology engineering.
In [NeOnD2.1.1] the vocabulary for talking about argumentation was quite extensive. After some attempt to apply it to different argumentation theories, evidence from successful experiences such as Compendium [SSS$^+$01] and DILIGENT [PST04] brought us to the decision of providing a much lighter vocabulary in codolight.
The key classes and properties in *codolight argumentation* module are illustrated by means of a simple labeled graph 6.1.

The central notion is `Position`. Positions are situations where agents provide ideas, arguments to ideas, and motivate arguments with possible design rationales. Arguments can be organized into threads. A thread can support a design solution (cf. chapter 7).
As an example, let's consider again the codolight model of the CiceroWiki tool presented in [PPG$^+$09]. The codolight argumentation notions of `ArgumentationThread`, `Idea`, `Argument`, etc. are specialized in Cicero as respectively `CiceroIssue`, `CiceroSolutionProposal`, `CiceroArgument`. Specific instances of issues or arguments appear in actual CiceroWiki sessions.

In section 6.2, entities defined in this module are described in detail, while next section 6.1 describes CPs reused in *codolight argumentation* module.

6.1 Patterns reused in *codolight argumentation* module

The *argumentation* module of *codolight* has been built by reusing the CP *situation* as building block [PG08, PGGPF07].

Situation. This CP allows to represent a view on a set of entities. It can be seen as a "relational context", reifying a relation. For example, a plan execution is a context including some actions executed by agents according to certain parameters and expected tasks to be achieved from a plan. The pattern is extracted from DOLCE+DnS Ultralite[2] by partial cloning of elements.

[1] http://www.ontologydesignpatterns.org/cpont/codo/codarg.owl
[2] http://www.ontologydesignpatterns.org/ont/dul/DUL.owl

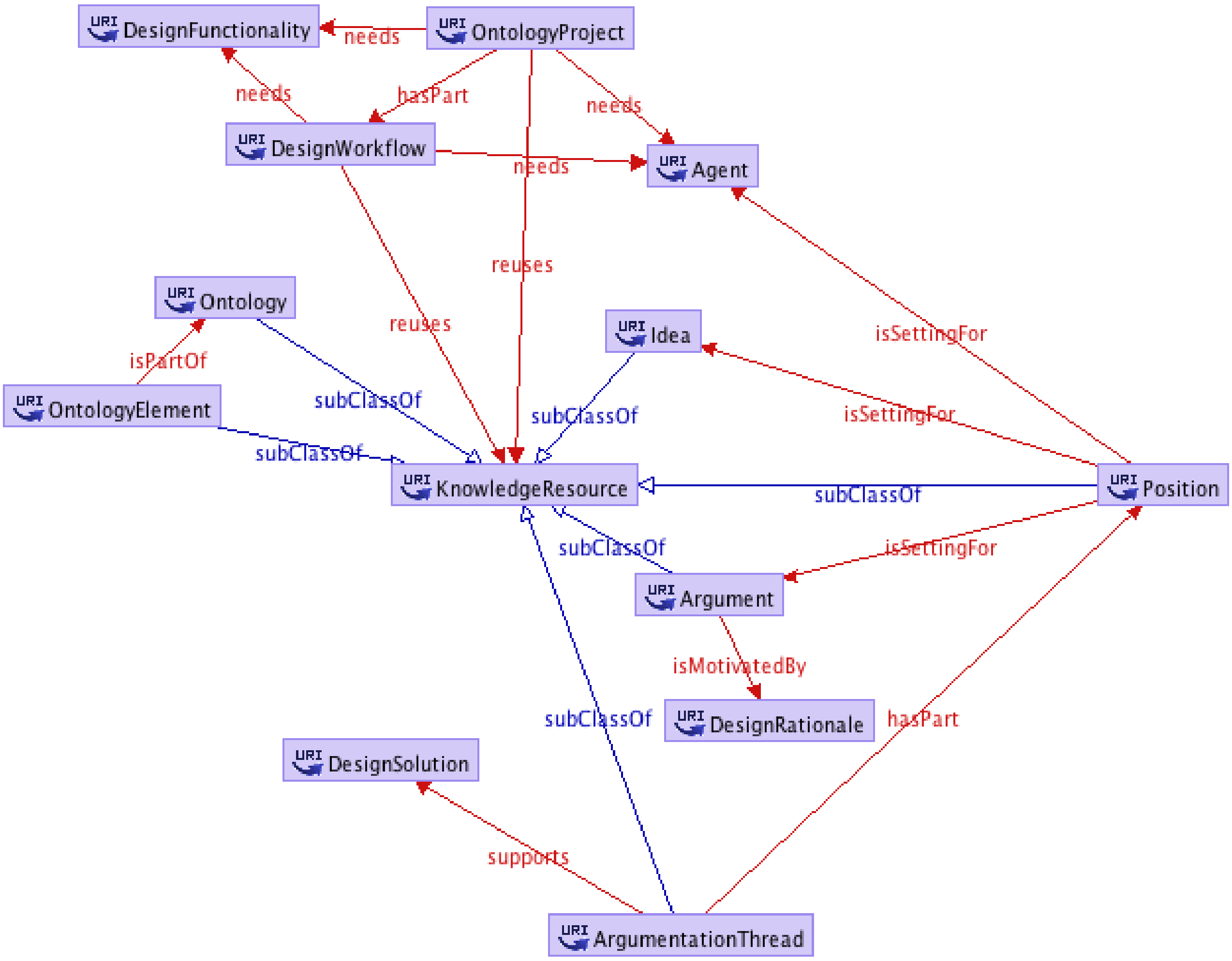

Figure 6.1: A simple graph of ontology elements from *codolight argumentation* module.

6.2 Entities of *codolight argumentation* module

6.2.1 Argument

An Argument is a situation in which a rationale is provided for a position towards an idea. For example, “I disagree with idea i” is a position that states the position type “disagreement” made by an agent that disagrees, the Idea i, the time at which the position statement occurs, etc. However, the position can include also a rationale that justifies the position. For example, the position can be: “your idea conflicts with the basic assumptions of our theory”. In this case, an agent is providing a rationale for his/her position, and this rationale is called here (based on e.g. IBIS model) “Argument”. On their turn, arguments are usually motivated by design rationales, intended as principles or best practices for modelling.

Formal definition.

```
The following entities are defined in this module.
:Argument
      a        owl:Class ;
      rdfs:subClassOf situation:Situation , codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
```

```
                owl:onProperty situation:hasSetting ;
                owl:someValuesFrom :Position
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :isMotivatedBy ;
                owl:someValuesFrom codkernel:DesignRationale
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :ArgumentKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

In codolight, the class `Argument` is described as sub-class of `situation:Situation` and `codkernel:KnowledgeResource`, that includes in its setting some position and is motivated by some design rationale. Furthermore, its intended meaning is formally represented by the knowledge type `ArgumentKType`, which is described by the following axiom:

```
:ArgumentKType
      a       codkernel:KnowledgeType ;
```

6.2.2 Argumentation thread

A complex (information) situation that includes a certain amount of positions organized as a thread that can have one or more ideas as subjects.

Formal definition.

```
:ArgumentationThread
      a       owl:Class ;
      rdfs:subClassOf situation:Situation , codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty partof:hasPart ;
                owl:someValuesFrom :Position
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :ArgumentationThreadKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

The class `ArgumentationThread` is formally described as sub-class of `situation:Situation` and `codkernel:KnowledgeResource`, having some position as its parts. Additionally, the knowledge type `ArgumentationThreadKType` represents the intensional meaning of this class and is formally described by the following axiom:

```
:ArgumentationThreadKType
      a       codkernel:KnowledgeType ;
```

6.2.3 Idea

An Idea is a description of something, either expressed by a formal expression, or by any other, informal information object. All kinds of ontology axioms are the typical subjects of ontology design argumentation, and they can be considered as ideas when they assume that role. Ideas are typically discussed by agents that provide an argument, i.e. a rationale that either challenges or justifies a position during an argumentation session. For example, an agent A can provide a counter-example (argument): “Turkey has territories outside Europe” that clarifies the rationale for its (negative) position P towards a `rdfs:subClassOf` axiom: `EuropeanCountry subClassOf (hasTerritory all (hasLocation Europe))`.

Formal definition.

```
:Idea
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :IdeaKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

An idea is a knowledge resource. The intensional meaning of the class `Idea` is represented by the knowledge type `IdeaKType`, formally described by the following axiom:

```
:IdeaKType
      a        codkernel:KnowledgeType ;
```

6.2.4 Position

A position is a situation, in which an agent provides an argument that responds to some idea, conveyed in a knowledge resource, either formal (e.g. an ontology element), or informal (e.g. a claim made in Italian). E.g., according to the “IBIS” model, an argument, possibly including a rationale, can respond to an idea, either by supporting it, or objecting to it. Arguments are argued by agents within a position.

Formal definition.

```
:Position
      a        owl:Class ;
      rdfs:subClassOf situation:Situation , codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:onProperty situation:isSettingFor ;
                owl:someValuesFrom :Idea
              ] ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:onProperty situation:isSettingFor ;
                owl:someValuesFrom agentrole:Agent
              ] ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
```

```
                owl:onProperty situation:isSettingFor ;
                owl:someValuesFrom :Argument
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :PositionKType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

In codolight, the class `Position` is a sub-class of the classes `situation:Situation` and `codkernel:KnowledgeResource`, it includes in its setting some ideas, some agent and some argument. Additionally, its intended meaning is represented by the knowledge type `PositionKType`, formally described by the following axiom:

```
:PositionKType
      a       codkernel:KnowledgeType ;
```

6.2.5 Motivates

Design rationales motivate arguments: given an idea, someone can have a position including an argument, motivated by a design rationale.

Formal definition.

```
:motivates
      a       owl:ObjectProperty ;
      rdfs:domain codkernel:DesignRationale ;
      rdfs:range :Argument ;
      owl:inverseOf :isMotivatedBy .
```

The domain of the `motivates` object property (inverse `isMotivatedBy`) is the class `codkernel:DesignRationale`, while its range is the class `Argument`.

6.2.6 Supports

After the exchange of some positions towards an idea, such thread can be said to support a certain design solution.

Formal definition.

```
:supports
      a       owl:ObjectProperty ;
      rdfs:domain :ArgumentationThread ;
      rdfs:range codkernel:DesignSolution ;
      owl:inverseOf :isSupportedBy .
```

The domain of the `supports` object property (inverse `isSupportedBy`) is the class `ArgumentationThread`, while its range is the class `codkernel:DesignSolution`

Chapter 7

The *Solutions* module

The *Solutions*[1] module of *codolight* contains a minimal set of classes and properties to represent the classes of design solution-related entities, and their relations.
Solution aspects of ontology design are not mandatory, since, although some form of solution is usually provided in ontology design, only recently repositories of reusable solutions have become an object of study in ontology engineering.
In [NeOnD2.1.1] the vocabulary for talking about solutions was quite limited. On the contrary, codolight tries to extend the support for ontology design solution description, which is needed by its applications in NeOn (cf. [PPG+09]). NeOn applications have being developed that reuse this vocabulary, including the XD (eXtreme Design) plugin and some functionalities from the ODP (Ontology Design Patterns) portal.
The key classes and properties in *codolight solutions* module are illustrated by means of a simple labeled graph 7.1.

The central notion is `codkernel:DesignSolution`. Design solutions are design situations, in which designers typically apply ontology design patterns (reusable models, good practices) to ontology elements that have been previously selected, reengineered, argumented, etc. Design solutions are matched against ontology requirements, which are typically expressed by competency questions. Patterns and competency questions are knowledge resources just like ontologies and ontology elements.
As an example, let's consider the XD plugin description[2], which specializes several subclasses of `codsolutions:OntologyDesignPatterns`, such as `LogicalPattern`, `ReengineeringPattern`, `ContentPattern`, etc., and instantiates them in the management of the ODP portal filesystem, which is controlled by means of an appropriate codolight module [PPG+09].

In section 7.1, entities defined in this module are described in detail. Axioms added to *kernel* entities are described in section 7.2.

7.1 Entities of *codolight solutions* module

This module defines the following entities.

7.1.1 Ontology requirement

The requirements expected to be fulfilled by an ontology. They are usually expressed by competency questions.

Formal definition.

[1] http://www.ontologydesignpatterns.org/cpont/codo/codsolutions.owl
[2] http://www.ontologydesignpatterns.org/cpont/codo/xd2codo.owl

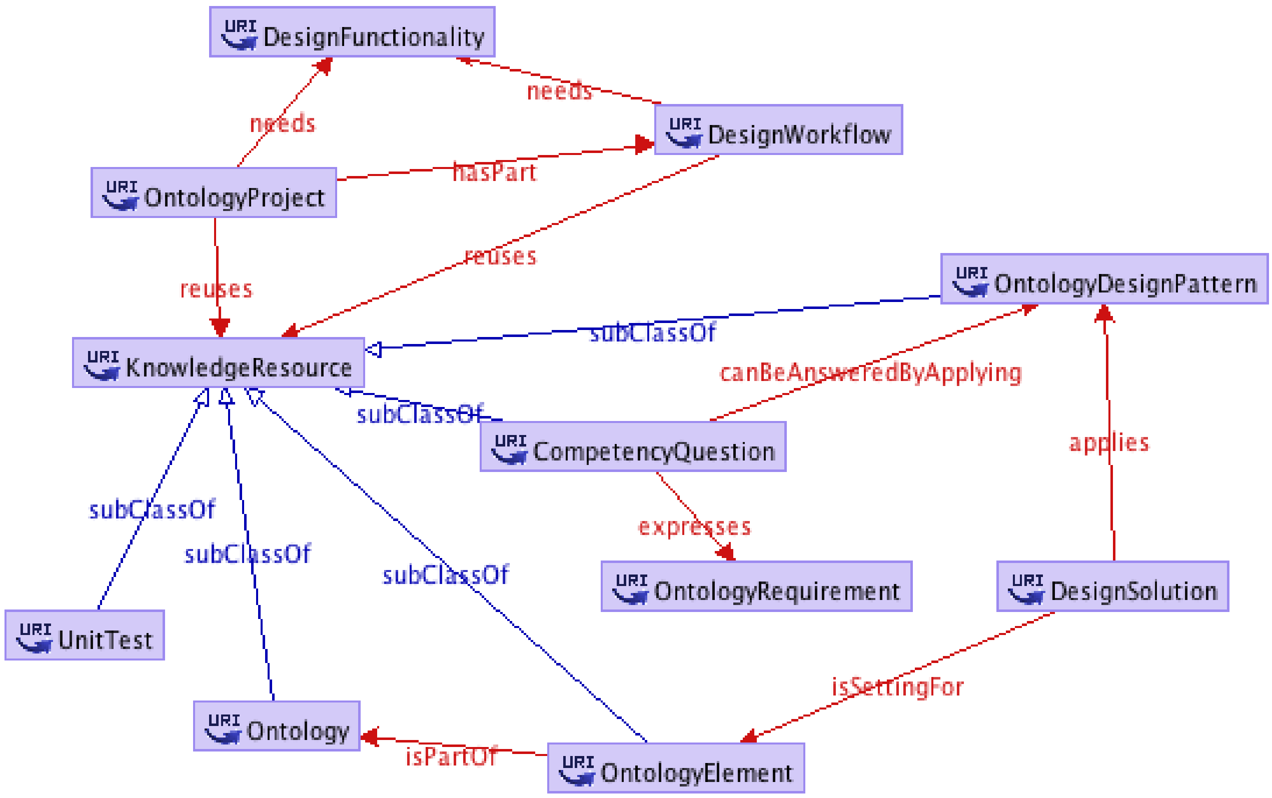

Figure 7.1: A simple graph of ontology elements from *codolight solutions* module.

```
:OntologyRequirement
      a        owl:Class ;
      rdfs:subClassOf taskrole:Task .
```

In codolight, an ontology requirement is a task.

7.1.2 Competency question

Queries (either in natural language or some query language) that express a requirement for an ontology to be fulfilled.

Formal definition.

```
:CompetencyQuestion
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:onProperty intensionextension:expresses ;
                owl:someValuesFrom :OntologyRequirement
              ] .
```

A competency question is formally described as a knowledge resource that expresses some ontology requirement.

7.1.3 Ontology design pattern

A class for holding together different kinds of solutions to ontology design.

Formal definition.

```
:OntologyDesignPattern
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource .
```

An ontology design pattern is a knowledge resource.

7.1.4 Unit test

A unit test is any formal expression that can be used (e.g. adding a pattern, submitting a query, etc.) to an existing ontology, in order to measure its fitness to some task. Unit tests are closely related to 'qoods' and especially to design pattern schemas that can be used to formalize ontology design patterns and use them as assembly components.

Formal definition.

```
:UnitTest
      a        owl:Class ;
      rdfs:subClassOf codkernel:KnowledgeResource ;
      rdfs:subClassOf
             [ a        owl:Restriction ;
               owl:allValuesFrom representation:FormalLanguage ;
               owl:onProperty resentation:hasRepresentationLanguage
             ] .
```

A unit test is formally described as a knowledge resource that has only formal language as representation language.

7.1.5 Fits

A relation between an ontology design pattern and the competency questions it addresses.

Formal definition.

```
:fits
      a        owl:ObjectProperty ;
      rdfs:domain :OntologyDesignPattern ;
      rdfs:range :CompetencyQuestion ;
      owl:inverseOf :canBeAnsweredByApplying .
```

The domain of the object property `fits` (inverse `canBeAnsweredByApplying`) is the class `OntologyDesignPattern`, while its range is the class `CompetencyQuestion`.

7.1.6 Applies

A relation between a design solution and the ontology design pattern it applies.

Formal definition.

```
:applies
      a       owl:ObjectProperty ;
      rdfs:domain codkernel:DesignSolution ;
      rdfs:range :OntologyDesignPattern ;
      rdfs:subPropertyOf descriptionandsituation:satisfies ;
      owl:inverseOf :isAppliedIn .
```

In codolight, `applies` is described as an object property (inverse `isAppliedIn`) having the class `codkernel:DesignSolution` as its domain and the class `OntologyDesignPattern` as its range.

7.2 Axioms extending *kernel* entities

The *kernel* module defines one class that is extended in the *codolight solutions* module by the following axioms.

```
codkernel:DesignSolution
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty situation:isSettingFor ;
                owl:someValuesFrom codkernel:OntologyElement
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty :applies ;
                owl:someValuesFrom :OntologyDesignPattern
              ] .
```

A design solution applies some ontology design pattern, and includes some ontology element in its setting.

Chapter 8

The *Tools* module

The *Tools*[1] module of *codolight* contains a minimal set of classes and properties to represent workflows from within ontology projects: collaborative workflows, accountable agents, need for an agent, etc.
As for all aspects of ontology design, except *data-related* ones, tool-related aspects of ontology design are not mandatory, but on the Semantic Web and in semantic technologies in general, designing and applying ontologies without using tools is de facto impossible.
In [CGL+06] the vocabulary for talking about tools was very limited. On the contrary, codolight tries to extend the support for design tool description, which is needed by its applications in NeOn (cf. [PPG+09]). In chapter 11, several alignments to other vocabularies describing tools and software projects or solutions add scope to this module.
The key classes and properties in codtools.owl are illustrated by means of a simple labeled graph 8.1. The central notion is `codkernel:DesignTool`. Design tools have knowledge types as their input and output types, have user types, have a programming language, and include capabilities from the pieces of software that are used in the tool. `PieceOfSoftware` is a class for all pieces of software, be them independent software modules, or just segments of them. Pieces of software apply code "entities", and apply software engineering techniques and patterns, notably interaction patterns.
Examples of a codolight descriptions for design tools are included for many NeOn Toolkit plugins in the deliverable [PPG+09]: they are quite detailed in representing relations between tools and workflows, workflows and functionalities, functionalities and knowledge/user types, etc. In addition, a section in [PPG+09] explains how tools are classified according to the knowledge types they have in input and/or output.

In section 8.1, entities defined in this module are described in detail. Axioms added to *kernel* entities are described in section 8.2.

8.1 Entities of *codolight tools* module

The following entities are defined in this module.

8.1.1 Technique

The class `Technique` represents the way a particular software task or procedure is carried out.

Formal definition.

```
:Technique
      a        owl:Class ;
      rdfs:subClassOf description:Description> .
```

[1] http://www.ontologydesignpatterns.org/cpont/codo/codtools.owl

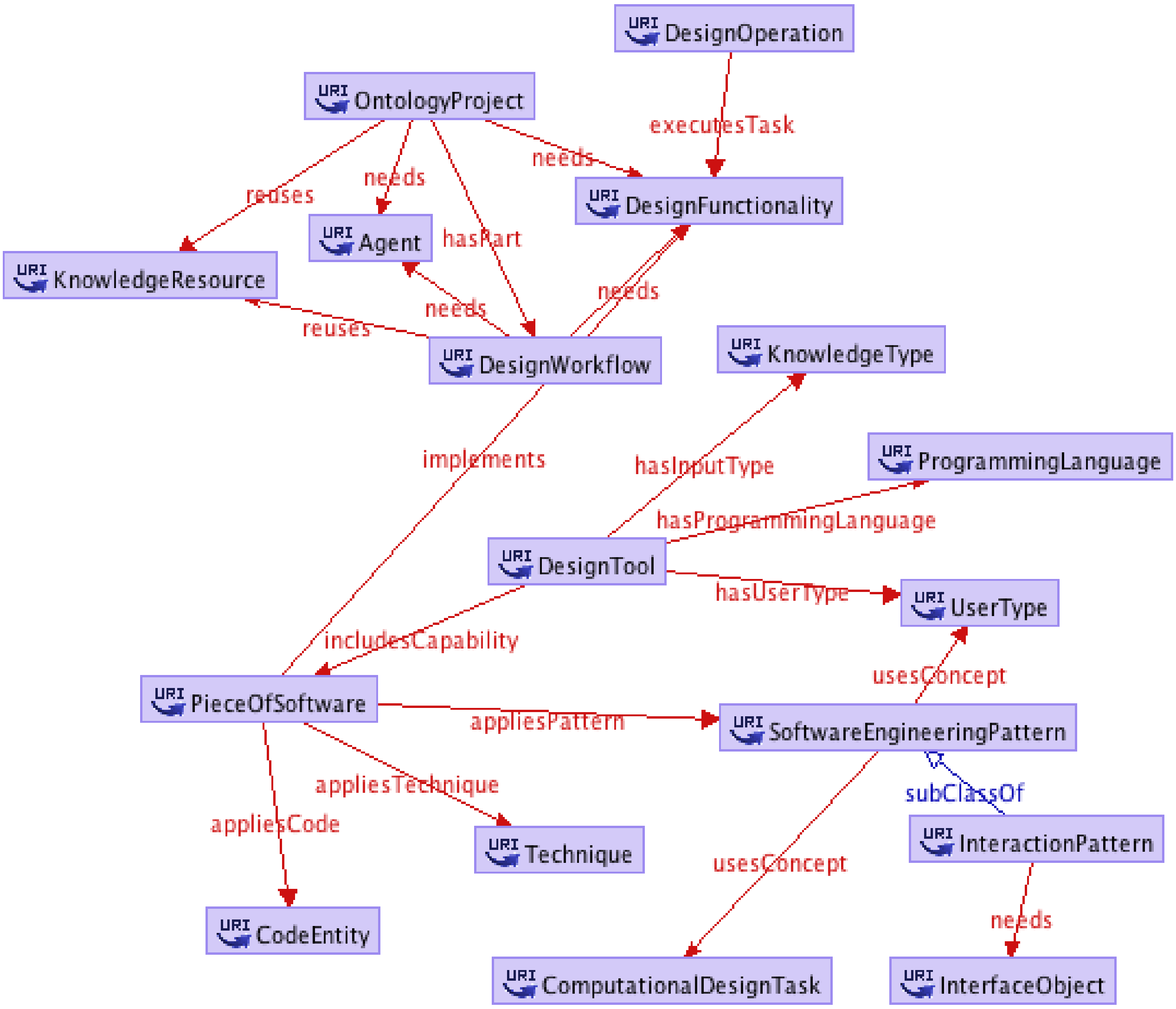

Figure 8.1: A simple graph of ontology elements from *codolight tools* module.

In codolight, a technique is defined as sub-class of `description:Description`.

8.1.2 Piece of software

A piece of software is a program or a library that enables a computer to perform a specific task. In this design context, it implements exactly one functionality by applying techniques or patterns, with specific code entities (behavioral, structural, or containers).

Formal definition.

```
:PieceOfSoftware
      a       owl:Class ;
      rdfs:subClassOf objectrole:Object ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty :appliesCode
```

```
               ] ;
      rdfs:subClassOf
               [ a        owl:Restriction ;
                 owl:cardinality "1"^^xsd:int ;
                 owl:onProperty :implements
               ] ;
      rdfs:subClassOf
               [ a        owl:Restriction ;
                 owl:onProperty :implements ;
                 owl:someValuesFrom codkernel:DesignFunctionality
               ] ;
      rdfs:subClassOf
               [ a        owl:Restriction ;
                 owl:minCardinality "0"^^xsd:int ;
                 owl:onProperty :appliesPattern
               ] ;
      rdfs:subClassOf
               [ a        owl:Restriction ;
                 owl:minCardinality "0"^^xsd:int ;
                 owl:onProperty :appliesTechnique
               ] .
```

In codolight, a piece of software is defined as an object that applies at least a piece of code e.g. a class, method, etc., and that implements exactly one functionality and can apply some software engineering pattern and/or technique.

8.1.3 Ontology application task

The class `OntologyApplicationTask` represent the tasks of an ontology within an application (e.g. retrieval, extraction, matching, etc.). It is distinguished from an ontology requirement, which is the content-oriented task that must be supported by an ontology. An ontology application task is also distinguished from a design functionality, which is a task that an application must support in order to perform ontology design operations.

Formal definition.

```
:OntologyApplicationTask
      a        owl:Class ;
      rdfs:subClassOf taskrole:Task ;
      owl:disjointWith codkernel:DesignFunctionality .
```

In codolight, an ontology application task is described as a task, the class is disjoint with the class of design functionalities.

8.1.4 Programming language

A programming language is a machine-readable language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that specify the behavior of a machine, to express algorithms precisely, or as a mode of human communication.

Formal definition.

```
:ProgrammingLanguage
      a       owl:Class ;
      rdfs:subClassOf representation:Language .
```

In codolight, a programming language is described as a special kind of language.

8.1.5 Code entity

The class `CodeEntity` represents pieces of code with a proper identity (class, method, function, file, attribute, etc.).

Formal definition.

```
:CodeEntity
      a       owl:Class ;
      rdfs:subClassOf objectrole:Object .
```

In codolight, a code entity is defined as an object.

8.1.6 Has input type

The object property `hasInputType` (inverse `isInputTypeFor`) defines a relation between tools, tasks, workflows, etc., and types of information objects that they take as input.

Formal definition.

```
:hasInputType
      a       owl:ObjectProperty ;
      rdfs:domain owl:Thing ;
      rdfs:range codkernel:KnowledgeType ;
      owl:inverseOf :isInputTypeFor .
```

In codolight, the domain of `hasInputType` is the class `owl:Thing`, while its range is the class `codkernel:KnowledgeType`.

8.1.7 Has output type

The object property `hasOutputType` (inverse `isOutputTypeFor`) defines a relation between tools, tasks, workflows, etc., and types of information objects

Formal definition.

```
:hasOutputType
      a       owl:ObjectProperty ;
      rdfs:domain owl:Thing ;
      rdfs:range codkernel:KnowledgeType ;
      owl:inverseOf :isOutputTypeFor .
```

In codolight, the domain of `hasOutputType` is the class `owl:Thing`, while its range is the class `codkernel:KnowledgeType`.

8.1.8 Applies technique

The object property `appliesTechnique` (inverse `isTechniqueAppliedIn`) represents the relation between a design tool or a piece of software and the techniques that they apply.

Formal definition.

```
:appliesTechnique
      a       owl:ObjectProperty ;
      rdfs:domain _:b1 ;
      rdfs:range :Technique ;
      owl:inverseOf :isTechniqueAppliedIn .

_:b1  a       owl:Class ;
      owl:unionOf (codkernel:DesignTool :PieceOfSoftware) .
```

This object property has the class `Technique` as range, while its range is the union of classes `codkernel:DesignTool` and `PieceOfSoftware`.

8.1.9 Applies code

The relation between a piece of software and the code it applies.

Formal definition.

```
:appliesCode
      a       owl:ObjectProperty ;
      rdfs:domain :PieceOfSoftware ;
      rdfs:range :CodeEntity ;
      owl:inverseOf :isCodeAppliedBy .
```

The domain of the object property `appliesCode` (inverse `isCodeAppliedBy`) is the class `PieceOfSoftware`, while its range is the class `CodeEntity`.

8.1.10 Has output data

A relation between tools, tasks, workflows, etc., and information objects representing their output data.

Formal definition.

```
:hasOutputData
      a       owl:ObjectProperty ;
      rdfs:domain owl:Thing ;
      rdfs:range intensionextension:InformationObject ;
      owl:inverseOf :isOutputDataFor .
```

The domain of the object property `hasOutputData` (inverse `isOutputDataFor`) is the class `owl:Thing`, while its range is the class `intensionextension:InformationObject`.

8.1.11 Has input data

A relation between tools, tasks, workflows, etc., and information objects representing their input data.

Formal definition.

```
:hasInputData
      a       owl:ObjectProperty ;
      rdfs:domain owl:Thing ;
      rdfs:range intensionextension:InformationObject ;
      owl:inverseOf :isInputDataFor .
```

The domain of the object property `hasInputData` (inverse `isInputDataFor`) is the class `owl:Thing`, while its range is the class `intensionextension:InformationObject`.

8.1.12 Implements

A relation between either a design tool or a piece of software and the logics it implements. Such logics in this context can be either a design functionality or a design workflow.

Formal definition.

```
:implements
      a       owl:ObjectProperty ;
      rdfs:domain _:b2 ;
      rdfs:range _:b3 ;
      owl:inverseOf :isImplementedIn .

_:b2  a       owl:Class ;
      owl:unionOf (codkernel:DesignTool :PieceOfSoftware) .

_:b3  a       owl:Class ;
      owl:unionOf (codkernel:DesignFunctionality codkernel:DesignWorkflow) .
```

In codolight, two anonymous classes are defined as domain and range of the object property `implements` (inverse `isImplementedIn`). Its domain is the union of the classes `codkernel:DesignTool` and `PieceOfSoftware`, while its range is the union of `codkernel:DesignFunctionality` and `codkernel:DesignWorkflow`.

8.1.13 Has user type

A relation between anything (typically a tool in this context) and its target user type. It is important for describing interaction situations involving users and tools with their interfaces.

Formal definition.

```
:hasUserType
      a       owl:ObjectProperty ;
      rdfs:domain owl:Thing ;
      rdfs:range codkernel:UserType ;
      owl:inverseOf :isUserTypeFor .
```

The object property `hasUserType` (inverse `isUserTypeFor`) has the class `owl:Thing` as domain and the class `UserType` as range.

8.1.14 Applies pattern

A relation between a software and software design patterns applied for implementing it.

Formal definition.

```
:appliesPattern
      a       owl:ObjectProperty ;
      rdfs:domain _:b4 ;
      rdfs:range codkernel:SoftwareEngineeringPattern ;
      owl:inverseOf :isPatternAppliedIn .

_:b4  a       owl:Class ;
      owl:unionOf (codkernel:DesignTool :PieceOfSoftware) .
```

The domain of the object property `appliesPattern` (inverse `isPatternAppliedIn`) is the union of the classes `codkernel:DesignTool` and `PieceOfSoftware`, while its range is the class `codkernel:SoftwareEngineeringPattern`.

8.1.15 Has programming language

A relation between a design tool and the programming language used for encoding its implementation.

Formal definition.

```
:hasProgrammingLanguage
      a       owl:ObjectProperty ;
      rdfs:domain codkernel:DesignTool ;
      rdfs:range :ProgrammingLanguage ;
      owl:inverseOf :isProgrammingLanguageOf .
```

The domain of the object property `hasProgrammingLanguage` (inverse `isProgrammingLanguageOf`) is the class `codkernel:DesignTool` while its range is the class `ProgrammingLanguage`.

8.1.16 Includes capability

A relation between a design tool and the software parts that its implementation is composed of.

Formal definition.

```
:includesCapability
      a       owl:ObjectProperty ;
      rdfs:domain codkernel:DesignTool ;
      rdfs:range :PieceOfSoftware ;
      owl:inverseOf :isCapabilityIncludedIn .
```

The domain of the object property `includesCapability` (inverse `isCapabilityIncludedIn`)is the class `codkernel:DesignTool` while its range is the class `PieceOfSoftware`.

8.2 Axioms extending *kernel* entities

The *codolight kernel* module defines three classes that are further formally described in the *tools* module; in the following paragraphs such additional axioms are shown.

Extending axioms for `codkernel:DesignFunctionality`

```
codkernel:DesignFunctionality
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:allValuesFrom codkernel:DesignOperation ;
                owl:onProperty taskexecution:isExecutedIn
              ] .
```

In this context, the description of design functionality is further detailed by an axiom asserting that design functionalities can be executed in only design operations.

Extending axioms for `codkernel:DesignOperation`

```
codkernel:DesignOperation
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty taskexecution:executesTask ;
                owl:someValuesFrom codkernel:DesignFunctionality
              ] .
```

A design operation is an action that executes some design functionality.

Extending axioms for `codkernel:DesignTool`

```
codkernel:DesignTool
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty :implements
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty :hasInputType
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty :hasUserType
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:minCardinality "1"^^xsd:int ;
                owl:onProperty :hasProgrammingLanguage
              ] .
```

In the *tool* module, the class representing design tools is further described. From the *kernel* module we only know that a design tool is an object. Furthermore, a design tool takes at least one input type, is implemented in at least one programing language, is targeted at at least one type of users and implements at least one design functionality or design workflow.

Chapter 9

The *Interfaces* module

The *Interfaces*[1] module of *codolight* contains some sample classes and properties to talk about interface objects, with exemplar instances.
Interface aspects of ontology design are not mandatory in the description of ontology design, but due the de facto dependency of design on tools, associating adequate interface objects to knowledge types and interaction patterns used in tools, it is relevant to have a vocabulary with which one can talk explicitly of those associations. A future alignment with W3C Fresnel vocabulary [BPKL06] is planned.
In [NeOnD2.1.1] vocabulary there was no coverage for talking about interfaces.
The key classes and properties in *codolight interfaces* module are illustrated by means of a simple labeled graph 9.1.

The central notion is `codkernel:InterfaceObject`. Interface objects are iconic objects that appear in actually running applications; they are classified as interface object types (that are actually used in tool descriptions), can have typical attributes, like having positions in a window, can have parts, and even a representation language. Sample subclasses include windows, tabs, widgets, buttons, item lists, etc.

In section 9.1, entities defined in this module are described in detail

9.1 Entities of *codolight interfaces* module

In this module several entities useful for interfaces are formally described, as it is shown in Figure 9.1. Several types of interface object are included as well as specific relations between them. Such interface objects are representatives for typical elements of a GUI and each of them is associated with a concept representing its intensional meaning. Such concepts are called “interface object types” and are formally defined by the following axioms:

9.1.1 Interface object type

A type of interface object. Used as the reification for the intension for any class of interface objects.

Formal definition.

```
:InterfaceObjectType
      a       owl:Class ;
      rdfs:subClassOf classification:Concept ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:allValuesFrom codkernel:KnowledgeResource ;
```

[1] http://www.ontologydesignpatterns.org/cpont/codo/codinterfaces.owl

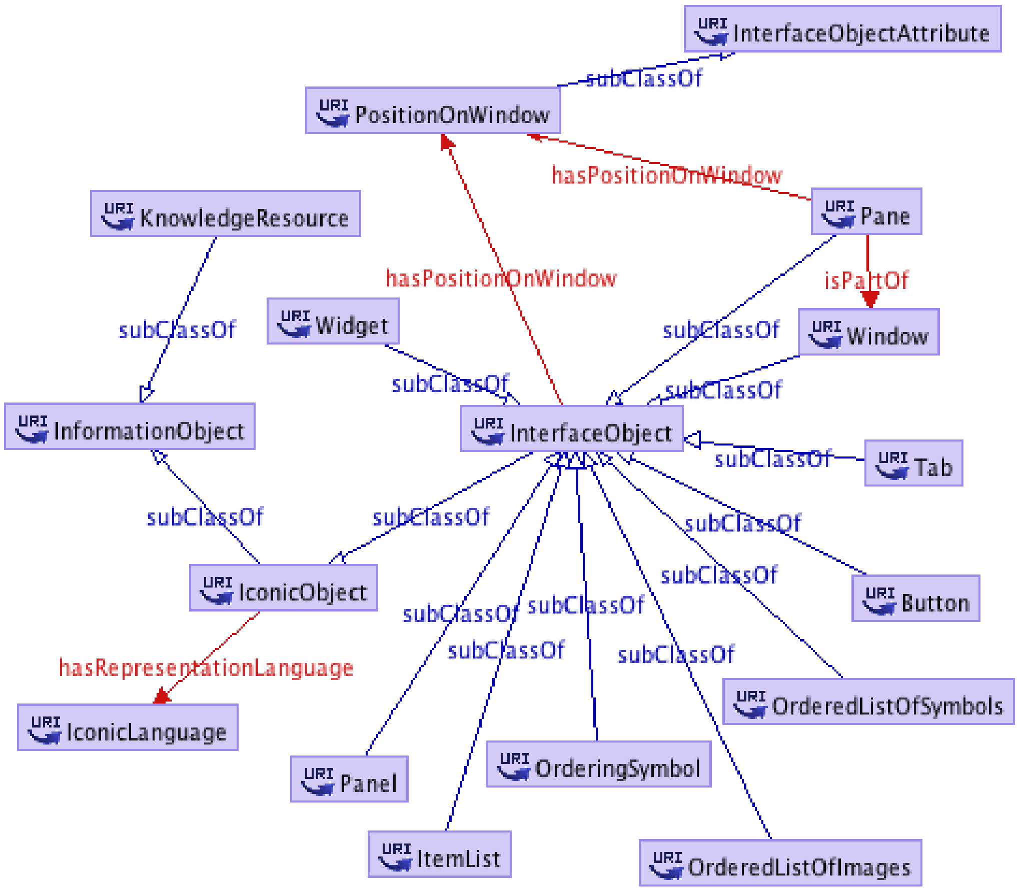

Figure 9.1: A simple graph of ontology elements from *codolight interfaces* module.

```
            owl:onProperty classification:classifies
          ] .
```

In codolight, an interface object type is described as a concept that classifies a knowledge resource. Among the others, in the *codolight interface* module, the classes `Button` and `Widget` are formally defined by the following axioms. We report their formal description as a sample of special types of interface objects, the others are defined analogously:

9.1.2 Button and Widget

A button, a GUI element. It is classified by the concept `ButtonInterfaceObjectType`.

```
:Button
      a        owl:Class ;
      rdfs:subClassOf codkernel:InterfaceObject ;
      rdfs:subClassOf
              [ a        owl:Restriction ;
                owl:hasValue :ButtonInterfaceObjectType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

A widget, a GUI element. It is classified by the concept `WidgetInterfaceObjectType`.

```
:Widget
```

```
      a       owl:Class ;
      rdfs:subClassOf codkernel:InterfaceObject ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:hasValue :WidgetInterfaceObjectType ;
                owl:onProperty classification:isClassifiedBy
              ] .
```

9.1.3 Interface object attribute

Formal definition.

```
:InterfaceObjectAttribute
      a       owl:Class ;
```

This class represents attributes of interface objects, such as colour, position, etc. For example, we report here an entity of the interface object attribute taxonomy i.e. the class `Colour`.

```
:Colour
      a       owl:Class ;
      rdfs:subClassOf :InterfaceObjectAttribute .
```

Chapter 10

The *Interaction* module

The *Interaction*[1] module of *codolight* contains some sample classes and properties to talk about interaction patterns, with exemplar instances taken from a reference software engineering site, `welie.com`.
Interaction aspects of ontology design are not mandatory in the description of ontology design, but due the de facto dependency of design on tools, with (currently most implicit) associations of knowledge types and design workflows with interaction patterns used in tools, it is relevant to have a vocabulary with which one can talk explicitly of those associations. Recent initiatives have actually created a scientific area for interaction aspects of semantic technologies [HHT09].
In [CGL+06] vocabulary there was no coverage for talking about interaction.
The key classes and properties in *codolight interaction* module are illustrated by means of a simple labeled graph 10.1.

The central notion is `codkernel:InteractionPattern`. Interaction patterns are software engineeering patterns that are used in the design of tools; they use concepts such as `UserType` and `ComputationalDesignTask`, and need interface objects. An interface object is linked to ontology projects through the class `codkernel:DesignFunctionality`: the computational task used by an interaction pattern is a design functionality, and in this way a design functionality is made computationally meaningful in the context of an interaction pattern. In other words, a system designer (or integrator or assembler) can (1) gather the requirements of an ontology project in terms of design functionalities, user types, and knowledge types, and (2) convert functionalities in computational tasks, and devise the best interaction pattern and interface objects for the task, the user types, and the knowledge types (cf. Fig. 10.2).
Sample interaction patterns are included in the OWL file of this module, e.g. `Accordion`, `Breadcrumbs`, etc.

In section 10.1, entities defined in this module are described in detail. Axioms added to *kernel* entities are described in section 10.2.

10.1 Entities of *codolight interaction* module

In order to make the ontology clearer, in this module are defined a set of instances of the class `codkernel:InteractionPattern` taken from a repository patterns for interaction design[2]. An example is given by the element `Slideshow` that is briefly explained as follows:

- Problem: the user wants to view a series of images/photos;
- Solution: Show each image for some seconds and provide controls to manually navigate back and forward, pause/resume and stop/return;

[1] http://www.ontologydesignpatterns.org/cpont/codo/codinteraction.owl
[2] http://www.welie.com

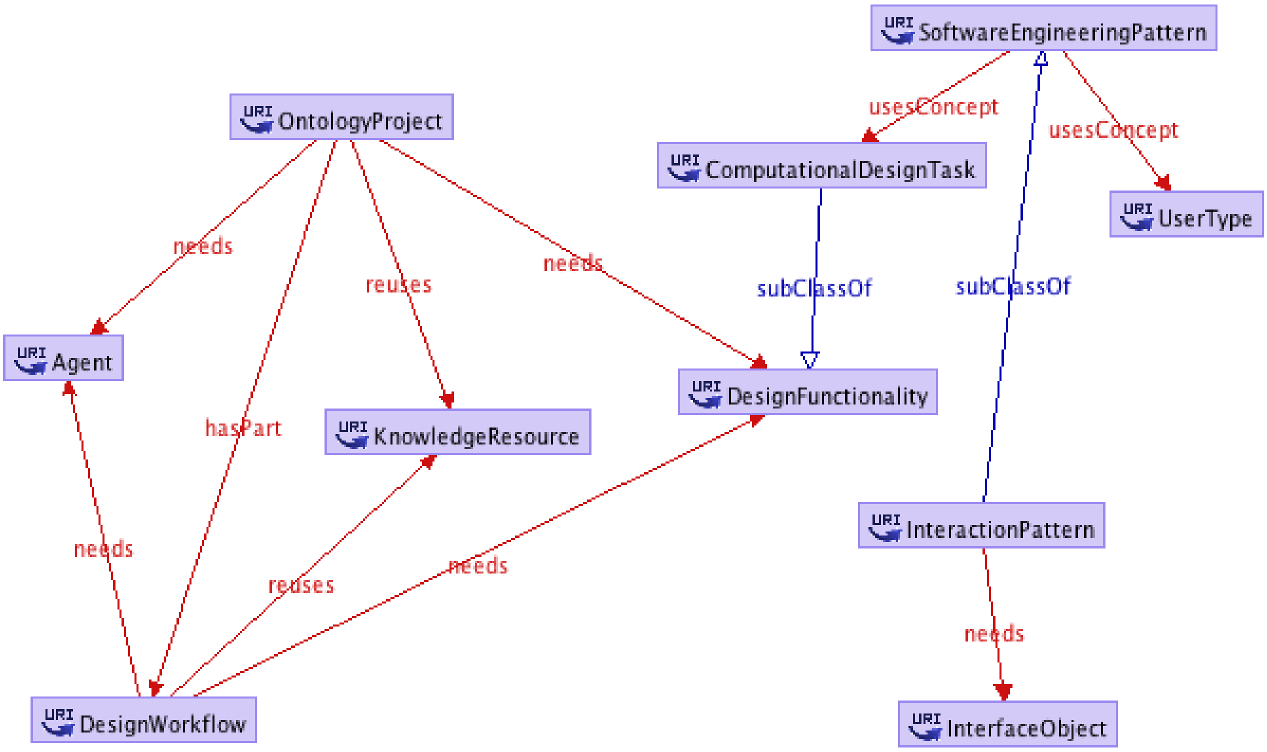

Figure 10.1: A simple graph of ontology elements from *codolight interaction* module.

- How: Usually the screen-estate is maximized for the photos and only a minimal representation of the required controls are used. The controls must be placed either at the top of the photo or below it. Make sure the following aspects are covered:
  - The controls should fade out in time if they are placed over the image;
  - The time between the photos must be configurable;
  - The user must be able to exit the slideshow mode;
  - Use a nice transition between photos! It make it a lot nicer;
  - Consider adding captions for the image title or comments.

The `Slideshow` element is formally described by the following axiom:

10.1.1 Slideshow

Formal definition.

```
:Slideshow
      a       codkernel:InteractionPattern .
```

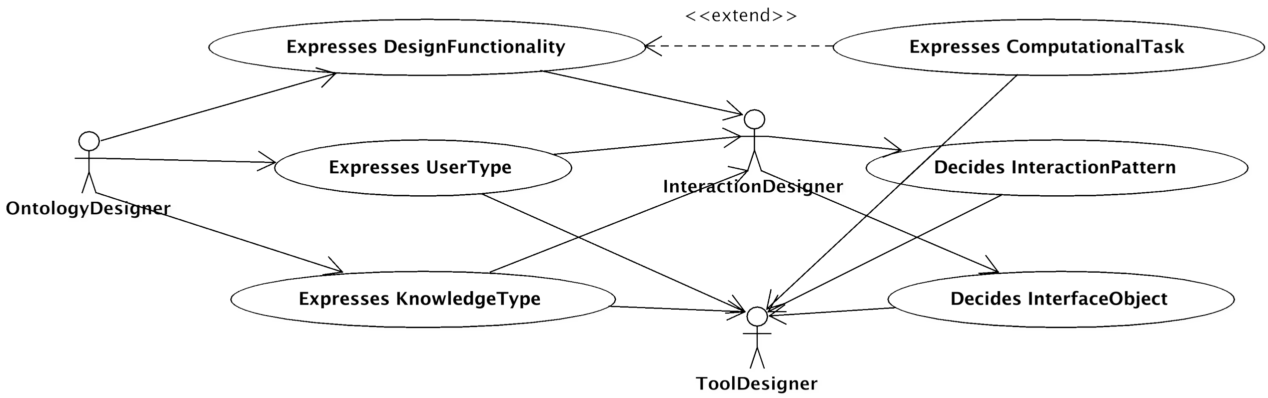

Figure 10.2: Matching requirements and tools through interaction pattern specification.

Additionally, in this module the class `ComputationalDesignTask` is formally described by the following axioms:

10.1.2 Computational design task

Formal definition.

```
:ComputationalDesignTask
      a        owl:Class ;
      rdfs:subClassOf codkernel:DesignFunctionality .
```

A computational design task is any type of design operation (i.e. a functionality) that needs to be performed with tool support.

10.2 Axioms extending *kernel* entities

The *kernel* module defines two classes that are further characterize in the *codolight interaction* module. They are formally described by the following axioms.

10.3 Extending axioms for `codkernel:SoftwareEngineeringPattern`

A software engineering pattern is a description that uses some computational design task and some user type.

```
codkernel:SoftwareEngineeringPattern
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty description:usesConcept ;
                owl:someValuesFrom :ComputationalDesignTask
              ] ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty description:usesConcept ;
                owl:someValuesFrom codkernel:UserType
              ] .
```

10.4 Extending axioms for `codkernel:InteractionPattern`

An interaction pattern needs some interface object.

```
codkernel:InteractionPattern
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty codkernel:needs ;
                owl:someValuesFrom codkernel:InterfaceObject
              ] .
```

Chapter 11

Alignments

In this chapter, we present some alignments that we have made between codolight, and other vocabularies that are widely used on the Semantic Web, or have been recently introduced by NeOn. We firstly present alignments to OWL metamodels (11.1), then to the Ontology Metadata Vocabulary (OMV) (11.2), to Description Of A Project (DOAP) (11.3), to the Access Rights ontology (11.4), to the Sweet Tools MIT vocabulary (11.5), to the Protégé workflow ontology (11.6), and finally to the Software Ontology Model (11.7). The prefixes for the aligned vocabularies are listed in Table 11.1.

Table 11.1: Prefixes used for the aligned ontologies.

Prefix	namespace	
omv:	http://omv.ontoware.org/2005/05/ontology	Ontology Metadata Vocabulary
access-rights:	http://www.uni-koblenz.de/ bercovici/owl/2008/7/accessRight.owl	Access Rights Ontology
ar-agents:	http://www.uni-koblenz.de/ schwagereit/owl/agents.owl	Access Rights (agents module)
ar-entity:	http://www.uni-koblenz.de/ bercovici/owl/2008/7/entity.owl	Access Rights (entity module)
ar-action:	http://www.uni-koblenz.de/ bercovici/owl/2008/8/action.owl	Access Rights (action module)
foaf:	http://xmlns.com/foaf/0.1/	FOAF ontology
doap:	http://usefulinc.com/ns/doap/	DOAP ontology
workflow:	http://protege.stanford.edu/rdf/workflow/	Protégé Workflow Ontology
owlodm1:	http://www.ontologydesignpatterns.org/ont/odm/owl10b.owl	OWL 1 Metamodel
owlodm2:	http://owlodm.ontoware.org/OWL2	OWL 2 Metamodel
owl:	http://www.w3.org/2002/07/owl/	OWL
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns	RDF
rdfs:	http://www.w3.org/2000/01/rdf-schema/	RDF Schema
sweet-tools:	http://www.ontologydesignpatterns.org/ont/sweettools.owl	Sweet Tools Ontology
som:	http://www.ifi.unizh.ch/ddis/evoont/2008/02/som	Software Ontology Model

11.1 Alignments to OWL

In this section, we report the alignments made between codolight and OWL entities. We have considered three different vocabularies:

- the original W3C RDF, RDFS, and OWL vocabularies 11.2
- the OWL1 metamodel designed by Peter Haase for NeOn [Haa06], and subsequently revised and enriched by Aldo Gangemi (this is the reason why it has a non-UKARL namespace) 11.3
- the OWL2 metamodel designed by Peter Haase for NeOn [HP09] 11.4

The reason why so many different vocabularies talk about entities from a same language is mainly due to pragmatical evolution of semantic technologies. The original vocabularies by W3C are not extremely detailed in distinguishing the constructs available in OWL (and RDF, RDFS); for example, it is difficult to talk explicitly about "existential restrictions", because these are just instances of `owl:Restriction`. On the other hand, W3C vocabularies are implemented in all APIs and tools for ontology engineering, and a ontology design vocabulary like codolight must be aligned to the main data vocabularies for maximal interoperability.
The OWL metamodels developed in NeOn try to overcome the referential coarseness of OWL constructs, e.g. by providing a class owlodm1:ExistentialRestriction. On the other hand, these metamodels are not a substitute for W3C OWL datamodel.

Table 11.2: Alignments between codolight and OWL

codolight entity	**type of alignment**	**OWL entity**
rdf:Statement	rdfs:subClassOf	codkernel:OntologyElement
rdfs:Container	rdfs:subClassOf	codkernel:OntologyElement
owl:DataRange	rdfs:subClassOf	codkernel:OntologyElement
owl:Ontology	rdfs:subClassOf	codkernel:Ontology
rdf:Property	rdfs:subClassOf	codkernel:OntologyElement
owl:AllDifferent	rdfs:subClassOf	codkernel:OntologyElement
rdfs:Class	rdfs:subClassOf	codkernel:OntologyElement
rdf:XMLLiteral	rdfs:subClassOf	codkernel:OntologyElement
rdf:List	rdfs:subClassOf	codkernel:OntologyElement
rdfs:Literal	rdfs:subClassOf	codkernel:OntologyElement

Table 11.3: Alignments between OWL 1 and codolight.

OWL 1 entity	**type of alignment**	**codolight entity**
owlodm1:OntologyProperty	rdfs:subClassOf	codkernel:OntologyElement
owlodm1:AnnotationProperty	rdfs:subClassOf	codkernel:OntologyElement
owlodm1:Annotation	rdfs:subClassOf	coddata:Annotation
owlodm1:DataRange	rdfs:subClassOf	codkernel:OntologyElement
owlodm1:URI	rdfs:subClassOf	intensionextension:InformationObject
owlodm1:OntologyElement	rdfs:subClassOf	codkernel:OntologyElement
owlodm1:AllDifferent	rdfs:subClassOf	codkernel:OntologyElement
owlodm1:Ontology	rdfs:subClassOf	codkernel:Ontology

11.2 Alignment to OMV

In this section, we report the alignments made between codolight and OMV (Ontology Metadata Vocabulary) entities [HSH$^+$05]. OMV is vocabulary for annotating ontologies with time, authors, tools, languages, etc. and it is used to provide support with ontology registries. However, from a design viewpoint the metadata provided by OMV have a semantics that is potentially compatible to that of other metamodels, and this alignment helps with metadata interoperability.

11.3 Alignment to DOAP

In this section, we report the alignments made between codolight and DOAP (Description Of A Project) vocabulary[1]. DOAP is a vocabulary for creating profiles of software projects with time, authors, FOAF (Friend

[1] http:trac.usefulinc.comdoap

Table 11.4: Alignments between OWL 2 and codolight entity.

OWL 2 entity	type of alignment	codolight entity
owlodm2:URI	rdfs:subClassOf	intensionextension:InformationObject
owlodm2:DataPropertyExpression	rdfs:subClassOf	codkernel:OntologyElement
owlodm2:DataRange	rdfs:subClassOf	codkernel:OntologyElement
owlodm2:AbbreviatedURI	rdfs:subClassOf	intensionextension:InformationObject
owlodm2:Axiom	rdfs:subClassOf	coddata:OntologyAxiom
owlodm2:Ontology	rdfs:subClassOf	codkernel:Ontology
owlodm2:ObjectPropertyExpression	rdfs:subClassOf	codkernel:OntologyElement
owlodm2:ClassExpression	rdfs:subClassOf	codkernel:OntologyElement
owlodm2:Annotation	rdfs:subClassOf	coddata:Annotation
owlodm2:Constant	rdfs:subClassOf	http://www.w3.org/2002/07:Thing
owlodm2:Constant	rdfs:subClassOf	intensionextension:InformationObject
owlodm2:FullURI	rdfs:subClassOf	intensionextension:InformationObject
owlodm2:SubObjectProperty	rdfs:subClassOf	codkernel:OntologyElement
owlodm2:FacetConstantPair	rdfs:subClassOf	intensionextension:InformationObject
owlodm2:Entity	rdfs:subClassOf	codkernel:OntologyElement
owlodm2:Individual	rdfs:subClassOf	codkernel:OntologyElement

Table 11.5: Alignments between OMV classes and codolight classes.

OMV class	type of alignment	codolight class
omv:OntologyDomain	rdfs:subClassOf	coddata:OntologyTopic
omv:OntologyEngineeringTool	rdfs:subClassOf	codkernel:DesignTool
omv:OntologySyntax	rdfs:subClassOf	coddata:EncodingSyntax
omv:LicenseModel	rdfs:subClassOf	description:Description
omv:OntologyTask	rdfs:subClassOf	codtools:OntologyApplicationTask
omv:OntologyEngineeringMethodology	rdfs:subClassOf	codkernel:DesignWorkflow
omv:KnowledgeRepresentationParadigm	rdfs:subClassOf	collection:Collection
omv:Location	rdfs:subClassOf	place:Place
omv:OntologyLanguage	rdfs:subClassOf	coddata:LogicalLanguage
omv:Ontology	rdfs:subClassOf	coddata:DataStructure
omv:FormalityLevel	rdfs:subClassOf	classification:Concept
omv:Party	rdfs:subClassOf	agentrole:Agent
omv:OntologyType	rdfs:subClassOf	classification:Concept

Of A Friend) vocabulary profiles, etc. The `doap:Project` notion addressed here is computational, and not social, therefore it has been aligned to `codkernel:Project`.

11.4 Alignment to Access Rights model

In this section, we report the alignments made between codolight and OWL entities. We have considered three different vocabularies:

Table 11.6: Alignments between OMV properties and codolight properties.

OMV property	type of alignment	codolight property
omv:contributesToOntology	rdfs:subPropertyOf	codworkflows:isInvolvedInTheDesignOf
omv:hasFormalityLevel	rdfs:subPropertyOf	classification:isClassifiedBy
omv:isIncompatibleWith	rdfs:subPropertyOf	coddata:relatedToOntology
omv:isSubDomainOf	rdfs:subPropertyOf	topic:isSubTopicOf
omv:hasSyntax	rdfs:subPropertyOf	coddata:hasEncoding
omv:hasOntologySyntax	rdfs:subPropertyOf	coddata:hasEncoding
omv:hasPriorVersion	rdfs:subPropertyOf	coddata:relatedToOntology
omv:hasContributor	rdfs:subPropertyOf	codworkflows:isInvolvedInDesignOperationsBy
omv:isOfType	rdfs:subPropertyOf	classification:isClassifiedBy
omv:hasDomain	rdfs:subPropertyOf	topic:hasTopic
omv:useImports	rdfs:subPropertyOf	coddata:imports
omv:isLocatedAt	rdfs:subPropertyOf	place:hasLocation
omv:hasOntologyLanguage	rdfs:subPropertyOf	coddata:hasLogicalLanguage
omv:endorsedBy	rdfs:subPropertyOf	codworkflows:isInvolvedInDesignOperationsBy
omv:hasCreator	rdfs:subPropertyOf	codworkflows:isInvolvedInDesignOperationsBy
omv:isBackwardCompatibleWith	rdfs:subPropertyOf	coddata:relatedToOntology
omv:usedOntologyEngineeringMethodology	rdfs:subPropertyOf	codprojects:isIntendedOutputOf
omv:hasLicense	rdfs:subPropertyOf	descriptionandsituation:isDescribedBy
omv:createsOntology	rdfs:subPropertyOf	codworkflows:isInvolvedInTheDesignOf
omv:endorses	rdfs:subPropertyOf	codworkflows:isInvolvedInTheDesignOf

Table 11.7: Alignments between Description Of A Project (DOAP) classes and codolight classes.

DOAP class	type of alignment	codolight class
doap:Repository	rdfs:subClassOf	collectionentity:Collection
doap:Project	rdfs:subClassOf	codkernel:Project
doap:Version	rdfs:subClassOf	coddata:Annotation
foaf:Document	rdfs:subClassOf	intensionextension:InformationObject

11.5 Alignment to Sweet Tools model

In this section, we report the alignments made between codolight and the Sweet Tools vocabulary, designed by Mike Bergman, which is used for a constantly updated repository of semantic web tools, available at [2].

11.6 Alignments to Protégé workflow model

In this section, we report the alignments made between codolight and the Protégé Workflow ontology [SNTM08]. As with the `doap:Project` alignment, the notion of `workflow:Project` addressed here is computational, and not social, therefore it has been aligned to `codkernel:Project`.

11.7 Alignment to Software Ontology Model

In this section, we report the alignment made between codolight and the SOM (Software Ontology Model) vocabulary, designed in order to represent Object-Oriented entities in the EvoOnt project [3]. Since SOM

[2] http:www.mkbergman.com/?page_id=325
[3] http://www.ifi.uzh.ch/ddis/evo/

Table 11.8: Alignments between Description Of A Project (DOAP) properties and codolight properties.

DOAP property	type of alignment	codolight property
foaf:topic	rdfs:subPropertyOf	topic:hasTopic
foaf:member	rdfs:subPropertyOf	collection:hasMember
foaf:page	rdfs:subPropertyOf	topic:isTopicOf

Table 11.9: Alignments between Access Rights classes and codolight classes.

Access rights class	type of alignment	codolight class
access-rights:Right	rdfs:subClassOf	description:Description
ar-entity:Module	rdfs:subClassOf	coddata:OntologyModule
ar-agents:Agent	rdfs:subClassOf	agentrole:Agent
ar-entity:Document	rdfs:subClassOf	intensionextension:InformationObject
ar-entity:Axiom	rdfs:subClassOf	coddata:OntologyAxiom
ar-entity:Content	rdfs:subClassOf	intensionextension:InformationObject

covers software code entities, these are linked to the `codtools:CodeEntity` class, which is associatable with pieces of software, tools, functionalities, etc. In line with the EvoOnt project goals, future developments can make it easier to access e.g. methods and functions in a semantic application at a much finer granularity.

Table 11.10: Alignments between Access Rights properties and codolight properties.

Access rights property	type of alignment	codolight property
ar-agents:subgroupOf	rdfs:subPropertyOf	partof:isPartOf
ar-agents:hasSubgroup	rdfs:subPropertyOf	partof:hasPart

Table 11.11: Alignments between Sweet Tools ontology and codolight.

Sweet Tools entity	type of alignment	codolight entity
sweet-tools:ToolLanguage	rdfs:subClassOf	codtools:ProgrammingLanguage
sweet-tools:Category	rdfs:subPropertyOf	classification:isClassifiedBy
sweet-tools:ToolCategory	rdfs:subClassOf	classification:Concept
sweet-tools:Item	rdfs:subClassOf	codkernel:DesignTool

Table 11.12: Alignments between Protégé Workflow classes and codolight classes.

Protégé workflow class	type of alignment	codolight class
workflow:Timestamp	rdfs:subClassOf	timeinterval:TimeInterval
workflow:Project	rdfs:subClassOf	codkernel:Project
workflow:Ontology_Component	rdfs:subClassOf	codkernel:OntologyElement
workflow:UIComponent	rdfs:subClassOf	codkernel:InterfaceObject
workflow:InitiationForm	rdfs:subClassOf	codkernel:InterfaceObject
workflow:Operation	rdfs:subClassOf	codkernel:DesignFunctionality
workflow:AnnotatableThing	rdfs:subClassOf	codkernel:KnowledgeResource
workflow:Server	rdfs:subClassOf	objectrole:Object
workflow:User	rdfs:subClassOf	codworkflows:AccountableAgent
workflow:GroupOperation	rdfs:subClassOf	codkernel:DesignFunctionality
workflow:Group	rdfs:subClassOf	collectionentity:Collection
workflow:Workflow	rdfs:subClassOf	codkernel:DesignWorkflow
workflow:Annotation	rdfs:subClassOf	coddata:Annotation
workflow:Activity	rdfs:subClassOf	taskrole:Task

Table 11.13: Alignments between Protégé Workflow properties and codolight properties.

Protégé workflow property	type of alignment	codolight property
workflow:activities	rdfs:subPropertyOf	partof:hasPart
workflow:performer	rdfs:subPropertyOf	participation:hasParticipant
workflow:member	rdfs:subPropertyOf	collectionentity:hasMember
workflow:group	rdfs:subPropertyOf	collectionentity:isMemberOf
workflow:next	rdfs:subPropertyOf	sequence:directlyPrecedes
workflow:annotates	rdfs:subPropertyOf	intensionextension:isAbout
workflow:associatedAnnotations	rdfs:subPropertyOf	intensionextension:isReferenceOf

Table 11.14: Alignments between Software Ontology Model (SOM) and codolight.

SOM entity	type of alignment	codolight entity
som:Entity	owl:equivalentClass	codtools:CodeEntity

Chapter 12

Conclusion and remarks

This deliverable has presented a deep revision and update of the C-ODO ontology design metamodel, called codolight. Codolight is dependent on explicit requirements and application tasks, it has been used for tool descriptions, aligned to external vocabularies, is lighter in complexity, and better associates the social and software layers of ontology design aspects.

We have provided a complete report, including: architectural considerations and the corolla/layering choices; commented OWL code and figures for each module of the codolight network; commented alignment axioms between codolight and several external vocabularies: OWL, OMV, DOAP, SOM, etc.

Codolight is actively used in some application tasks, including: browsing semantic data about ontology projects, smart searching and selecting of design components, creating custom design configuration interfaces, help collecting ontology requirements, providing a shared network of vocabularies.

Part of these functionalities are being implemented within NeOn in the Kali-ma tool (see [PPG+09]).

Bibliography

[BLC08] Tim Berners-Lee and Dan Connolly. Notation3 (n3): A readable rdf syntax. W3C Team Submission, W3C, January 2008.

[BPKL06] Christian Bizer, Emmanuel Pietriga, David Karger, and Ryan Lee. Fresnel: A Browser-Independent Presentation Vocabulary for RDF. In *5th International Semantic Web Conference, Athens, GA, USA, November 5-9*, 2006.

[CGL+06] Carola Catenacci, Aldo Gangemi, Jos Lehmann, Malvina Nissim, Valentina Presutti, and Gerardo Steve. D2.1.1: Design Rationales for collaborative development of networked ontologies Ð State of the art and the Collaborative Ontology Design Ontology. Technical report, NeOn, 2006.

[GLP+07] Aldo Gangemi, Jos Lehmann, Valentina Presutti, Malvina Nissim, and Carola Catenacci. CODO: an OWL Metamodel for Collaborative Ontology Design. In *Workshop on Social and Collaborative Construction of Structured Knowledge*, May 2007.

[Haa06] Peter Haase. D1.1.1: Networked ontology model. Technical report, NeOn, 2006.

[HHT09] Siegfried Handschuh, Tom Heath, and VinhTuan Thai. Visual interfaces to the social and the semantic web (VISSW 2009). In *IUI '09: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 499–500, New York, NY, USA, 2009. ACM.

[HP09] Peter Haase and Raul Palma. D1.1.5: Networked ontology model. Technical report, NeOn, 2009.

[HSH+05] Jens Hartmann, York Sure, Peter Haase, Raul Palma, and Mari del Carmen Suárez-Figueroa. OMV – Ontology Metadata Vocabulary. In Chris Welty, editor, *Ontology Patterns for the Semantic Web Workshop*, Galway, Ireland, 2005.

[PG08] Valentina Presutti and Aldo Gangemi. Content Ontology Design Patterns as Practical Building Blocks for Web Ontologies. In Qing Li, Stefano Spaccapietra, Eric Yu, and Antoni Olivé, editors, *ER*, volume 5231 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 2008.

[PGGPF07] Valentina Presutti, Aldo Gangemi, Asuncion Gomez-Perez, and Mari-Carmen Suarez Figueroa. Library of Design Patterns for Collaborative Development of Networked Ontologies. Deliverable D2.5.1, NeOn project, 2007.

[PPG+09] Wim Peters, Valentina Presutti, Aldo Gangemi, Dunja Mladenic, Raul Palma, Klaas Dellschaft, Holger Lewen, Alessandro Adamou, and Enrico Daga. D2.3.2 Practical Methods to Support Collaborative Ontology Design (v2). Technical report, NeOn, 2009.

[PST04] H. Pinto, S. Staab, and C. Tempich. DILIGENT: Towards a fine-grained methodology for DIstributed, Loosely-controlled and evolvInG Engineering of oNTologies. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004)*, Valencia, Spain, 2004.

[SCd+09] Marta Sabou, Guadalupe Aguadode Cea, Mathieu d'Aquin, Enrico Daga, Holger Lewen, Elena Montiel-Ponsoda, Valentina Presutti, and Mari Carmen Suarez-Figueroa. NeOn D2.2.3 Methods and Tools for the Evaluation and Selection of Knowledge Components. Technical report, NeOn, 2009.

[SNTM08] Abraham Sebastian, Natalya Fridman Noy, Tania Tudorache, and Mark A. Musen. A Generic Ontology for Collaborative Ontology-Development Workflows. In Aldo Gangemi and Jérome Euzenat, editors, *EKAW*, volume 5268 of *Lecture Notes in Computer Science*, pages 318–328. Springer, 2008.

[SSS+01] Albert Selvin, Simon Buckingham Shum, Maarten Sierhuis, Jeff Conklin, Beatrix Zimmermann, Charles Palus, Wilfred Drath, David Horth, John Domin, and Gangmin Li. Compendium: Making Meetings into Knowledge Events. In *Knowledge Technologies 2001*, 2001.