NeOn-project.org

NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 – “Semantic-based knowledge and content systems”

D 5.1.1 NeOn Modelling Components

Deliverable Co-ordinator: **Mari Carmen Suárez-Figueroa**

Deliverable Co-ordinating Institution: **UPM**

Other Authors: **Saartje Brockmans (UKARL), Aldo Gangemi (CNR), Asunción Gómez-Pérez (UPM), Jos Lehmann (CNR), Holger Lewen (UKARL), Valentina Presutti (CNR), and Marta Sabou (OU)**

Document Identifier:	NEON/2007/D5.1.1/v1.1	Date due:	February 28, 2007
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	March 30, 2007
Project start date:	March 1, 2006	Version:	v1.1
Project duration:	4 years	State:	Final
		Distribution:	Public

NeOn Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities, 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 28 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.upm.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: Richard Benjamins E-mail address: rbenjamins@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SI-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 – 655 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 Koblenz-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. Martino 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	**Asociación Española de Comercio Electrónico (AECE)** C/Alcalde Barnils, Avenida Diagonal 437 08036 Barcelona Spain Contact person: Jose Luis Zimmerman E-mail address: jlzimmerman@fecemd.org
Food and Agriculture Organization of the UN (FAO) Viale delle Terme di Caracalla 1 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

Work package participants

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

UPM

UKARL

OU

CNR

FAO

Change Log

Version	Date	Amended by	Changes
0.1	20-09-2006	Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez	Initial Draft
0.2	25-09-2006	Mari Carmen Suárez-Figueroa	Included subsection about processes for creating DL and OWL ontologies
0.3	5-10-2006	Mari Carmen Suárez-Figueroa, Holger Lewen, Marta Sabou	Included contributions from UKARL and OU (section about mappings) and contributions from UKARL (section about rules)
0.4	9-10-2006	Mari Carmen Suárez-Figueroa, Valentina Presutti	Included contributions from CNR (about W3C best practices)
0.5	02-11-2006	Mari Carmen Suárez-Figueroa, Marta Sabou, Jos Lehman	Included contributions from OU (about ontology selection and reuse) and contributions from CNR (about collaboratively designs, modular designs, and modelling with tools)
0.6	25-11-2006	Mari Carmen Suárez-Figueroa, Holger Lewen, Asunción Gómez-Pérez	Reviewed the deliverable and included comments from internal revisor (UKARL)
0.7	15-01-2007	Mari Carmen Suárez-Figueroa	First draft of the guidelines for modelling ontologies Reviewed best practices and patterns for modelling ontologies
0.80	21-02-2007	Mari Carmen Suárez-Figueroa, Aldo Gangemi, Valentina Presutti	Updated the section on components for modelling ontologies
0.81	23-02-2007	Mari Carmen Suárez-Figueroa, Aldo Gangemi, Valentina Presutti	Reviewed the section on components (best practices, patterns, etc.) for modelling ontologies
0.90	28-02-2007	Mari Carmen Suárez-Figueroa	Reviewed the deliverable
0.91	5-03-2007	Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez	Updated and reviewed the section on components (best practices, patterns, etc.) for modelling ontologies
0.92	5-03-2007	Mari Carmen Suárez-Figueroa, Valentina Presutti	Reviewed and updated the section about logical patterns
0.93	8-03-2007	Mari Carmen Suárez-Figueroa	Reviewed and updated the section about logical patterns

0.94	9-03-2007	Mari Carmen Suárez-Figueroa, Rosario Plaza	Reviewed the deliverable (from a lingüistic point of view)
0.95	12-03-2007	Mari Carmen Suárez-Figueroa	Included comments done by Q.A. reviewer (Peter Haase) during the first review
0.96	13-03-2007	Mari Carmen Suárez-Figueroa	Updated UML diagrams (following the comments done by Q.A. reviewer during the first review)
0.97	14-03-2007	Mari Carmen Suárez-Figueroa, Rosario Plaza	Reviewed the deliverable (from a lingüistic point of view) Included comments done by Q.A. reviewer (Peter Haase) during the first review
0.98	20-03-2007	Mari Carmen Suárez-Figueroa, Saartje Brockmans	Included new contributions from UKARL (section on rules)
0.99	21-03-2007	Mari Carmen Suárez-Figueroa	Included distinction between DL and Frames (in chapter 2) Updated UML diagrams (following the comments done by Q.A. reviewer during the first review)
1.0	21-03-2007	Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez	Updated introduction, executive summary, chapter 3 and conclusions (following the comments done by Q.A. reviewer during the first review and the decisions taken during WP5 phone call)
1.01	23-03-2007	Mari Carmen Suárez-Figueroa	Updated UML diagrams (following the comments done by Q.A. reviewer during the first review)
1.02	23-03-2007	Aldo Gangemi, Valentina Presutti	Updated chapter 3 and chapter 5 (about architectural patterns)
1.03	27-03-2007	Mari Carmen Suárez-Figueroa	Included UML diagrams of some content patterns. Included more logical patterns
1.04	28-03-2007	Mari Carmen Suárez-Figueroa, Rosario Plaza	Reviewed the deliverable (from a lingüistic point of view)
1.05	29-03-2007	Mari Carmen Suárez-Figueroa	Included comments done by Q.A. reviewer (Peter Haase) during the second review
1.06	29-03-2007	Mari Carmen Suárez-Figueroa	Updated UML diagrams (following the comments done by Q.A. reviewer during the second review)
1.1	2-04-2007	Mari Carmen Suárez-Figueroa	Updated the introduction and reviewed the whole deliverable

Executive Summary

As stated in the NeOn Description of the Work (DoW), the goal of the task *T5.1. Analysis of knowledge modelling components for networks of ontologies* and its corresponding deliverable *D5.1.1. NeOn Modelling Components* is "to analyse which of the main ontology modelling components permit ontology engineers to model networked ontologies collaboratively and how this could be done". From this statement, the deliverable has two major goals:

- Identifying the modelling components to be used for modelling networks of ontologies in a collaborative way.
- Creating guidelines for using the identified modelling components.

Since the NeOn modelling components (the OWL ontology metamodel, the rule metamodel, the mapping metamodel, etc.) are already described in the deliverable *D1.1.1. Networked ontology model: initial model* [31], we decided **to include in D5.1.1 the first version of an inventory of OWL-based design patterns** and to postpone the second version of the inventory of the OWL-based design patterns and patterns for the rest of modelling components for the deliverable D5.1.2 (at month 36).

In this deliverable we first present the State of the Art focused on processes for the creation of the modules already identified in D1.1.1. We also present some previous work on creating designs collaboratively.

Based on the State of the Art and on the reuse and/or adaptation of pre-existing knowledge sub-components (e.g. patterns, W3C and Knowledge Web best practices, etc.), this deliverable also introduces the first version of the inventory of *NeOn Modelling Components*, focusing on a subset of the NeOn networked ontology metamodel, which is the OWL ontology metamodel. Thus, the *NeOn Ontology Modelling Components* (namely, OWL-based design patterns) presented in this deliverable are divided into three different types: *logical patterns* (elements of the OWL module from the NeOn networked ontology metamodel [31], or compositions of those elements), *architectural patterns* (logical patterns or compositions of them that are used exclusively in the design of an ontology) and *content patterns* (instantiations of logical patterns or composition of them). Guidelines for using the modelling components will be included in the deliverable *D5.4.1. NeOn methodology for building contextualized networked ontologies*, where the NeOn methodology for building networks of ontologies will be presented and delivered in month 24.

Additionally, an annex collecting some definitions of the terminology used is provided.

Note on Sources and Original Contributions

The NeOn consortium is an inter-disciplinary team composed of many partners; therefore, to make deliverables self-contained and comprehensible to all partners, some deliverables thus necessarily include state-of-the-art surveys and associated critical assessment. Where there is no advantage in recreating such materials from first principles, the partners follow the standard scientific practice and occasionally make use of their own pre-existing intellectual property in such sections. In the interests of transparency, we identify below the main sources of such pre-existing materials in this deliverable:

- Parts of Section 2.1.1 contain material adapted from [6, 51, 23, 48, 46, 44].
- Parts of Section 2.1.2 contain material adapted from [45, 28].
- Parts of Section 2.2 contain material adapted from [16, 64].
- Parts of Section 2.3 are taken from the official SWRL proposal [33, 36].

Table of Contents

List of tables

List of figures

1. Introduction

1.1. Overview of WP5 Work in NeOn

The Semantic Web of the future will be characterized by using a very large number of ontologies; all of them developed with respect to a number of contextual factors, which may reflect the skills of the developers, their application needs, their cultural and social 'biases', and the tools they prefer to use. As the complexity of semantic applications increases, more and more knowledge will be embedded in applications, typically drawn from a wide variety of sources. This new generation of applications will thus reflect the fact that new ontologies are embedded in a network of already existing ontologies built by distributed teams. In this scenario it will become prohibitively expensive to adopt the current approach of building ontologies from scratch where the expectation is to produce a single, globally consistent semantic model, which serves the needs of application developers and fully integrates a number of pre-existing ontologies. In contrast with the current model, future applications will rely on *networks of contextualized ontologies*, which are usually locally, but not globally consistent. So, future Semantic Web applications will be based on networks of contextualized ontologies, which are in continuous evolution. Such networks could be ontologies that already exist or could be developed by reusing either other ontologies, or knowledge resources built by distributed teams.

With this new vision of the ontologies and the Semantic Web applications, it is important to provide strong (technical and methodological) support for collaborative and context-sensitive development of ontologies and applications.

The state of the art relating to methodologies for developing ontologies revealed that:

- ✓ Nowadays no methodology adequately supports the collaborative and context aspects of networks of ontologies, which are needed in the NeOn environment.
- ✓ METHONTOLOGY [28] and On-To-Knowledge [59] mainly include guidelines for building single ontologies from the ontology specification to the ontology implementation.
- ✓ METHONTOLOGY [28] and On-To-Knowledge [59] do not pay too much attention to the development of networks of ontologies carried out by geographically distributed teams and where contextual information is introduced by developers in different stages of the ontology development process.

Thus, the aforementioned methodological support for development *networks of contextualized ontologies* is the main goal of WP5, whose main outcomes are:

- ➢ The NeOn methodology to support the collaborative construction and dynamic evolution of networks of ontologies in distributed environments where contextual information is introduced by developers at different stages of the ontology development process. This work will be based on existing methodologies for ontology construction, in particular METHONTOLOGY [28], On-To-Knowledge [59] and DILIGENT [49]. The methodology will include specific methods from WP1-WP4, and the infrastructure supporting it will be developed in WP6.
- ➢ The NeOn methodology for the development of large scale Semantic Web applications.

In the construction of the NeOn methodology to support the collaborative construction and dynamic evolution of networks of ontologies in distributed environments, we have identified the following sub-goals:

- ➢ *Analysis of knowledge modelling components for networks of ontologies*. We have analyzed which are the main modelling components that permit teams (formed by ontology engineers, domain experts, etc.) to model networks of ontologies collaboratively and how this modelling can be carried out. A first version of the inventory of the OWL-based design patterns is delivered in D5.1.1; a more complete analysis will be presented in the subsequent deliverable to D5.1.1 (that is, in D5.1.2) and the guidelines in D5.4.1.

- *Analysis of protocols for exchanging and sharing*. We have analyzed existing protocols for exchanging and sharing information in order to support the development of NeOn protocols for exchanging and sharing ontologies and related metadata that support collaborative construction and dynamic evolution of networks of ontologies. This analysis is presented in D5.2.1; the development of protocols is part of the work to be performed in WP6.
- *Identification of the development process and lifecycle for networks of ontologies*.

  For the development process, we are now identifying and defining which activities are carried out when networks of ontologies are built in collaboration. As a result of this work, we will obtain the NeOn Glossary of Activities (which will be delivered in D5.3.1). At present, we are collaboratively building a draft of this glossary and trying to achieve a consensus between NeOn partners.

  For the lifecycle, we will identify when the activities included in the NeOn Glossary of Activities should be carried out and also the stages through which a network of ontologies moves during its life. We will describe the activities to be performed at each stage and how these are related (precedence, return, etc.). The results of this work will be also presented in D5.3.1.
- *Neon methodology for building contextualized networked ontologies*. We will identify what, how and by whom a given activity should be performed. For each activity, we will suggest a set of methods, techniques and tools to be used. In this subgoal a strong cooperation with WP1-4 is required. The results of this work will be presented in D5.4.1.

1.2. Overview of the Deliverable

As stated in the NeOn Description of the Work (DoW), the goal of the task *T5.1. Analysis of knowledge modelling components for networks of ontologies* and its corresponding deliverable *D5.1.1. NeOn Modelling Components* is "to analyse which of the main ontology modelling components permit ontology engineers to model networked ontologies collaboratively and how this could be done".

From this statement, it can be extracted that the deliverable has the following two major goals:

- Identifying the modelling components to be used for modelling networks of ontologies in a collaborative way.
- Creating guidelines for using the identified modelling components.

However, WP5 partners discovered an overlap between D5.1.1 and the task *T1.1. Formal networked ontology model* and its corresponding deliverable *D1.1.1. Networked ontology model: initial model*. The task T1.1 is defined as follows: "to develop models for representing and managing relations between multiple networked ontologies. The model will be the basis for the consistency and propagation models".

In summary, both aforementioned deliverables had similar goals, although the deliverable in WP5 has to include guidelines for using the NeOn modelling components. For this reason, we decided to focus on including collaborative aspects on the guidelines by means of reusing existing design patterns and well-accepted best practices, as a first approach to create the guidelines for modelling networks of ontologies.

Therefore, an adjustment was made in the D5.1.1 goals, which is as follows:

*"**The goal of D5.1.1 is to provide the first version of an inventory of OWL-based design patterns (here called NeOn Ontology Modelling Components) trying to maximize reusability of existing experiences.** The first version of the guidelines for modelling networks of ontologies is postponed for deliverable D5.4.1 (at month 24). Furthermore, the second version of the inventory of the NeOn Ontology Modelling Components and the inventory of the other modelling components*

(namely, NeOn Mapping Modelling Components, NeOn Rules Modelling Components, and NeOn Module Modelling Components) are postponed for deliverable D5.1.2 (at month 36)."

Then, this deliverable presents the first version of an inventory of OWL-based design patterns (here called *NeOn Ontology Modelling Components*) to be used by teams (formed by ontology engineers, domain experts, etc.) to model OWL ontologies. The design pattern inventory is divided into three different types (logical patterns, architectural patterns, and content patterns), and each component is described with a specific template.

Given the relationship between this deliverable and other deliverables (specially, D5.1.2 and D5.4.1), we took into account the following issues:

- The identification and pattern definitions should be based on the elements included in the NeOn networked ontology metamodel defined in D1.1.1 [31]. The NeOn networked ontology metamodel consists of several modules, as shown in Figure 1. The core module is the OWL ontology metamodel, which is extended with other modules such as the rule metamodel and the mapping metamodel. Thus, we decided to start with the OWL-based design patterns.

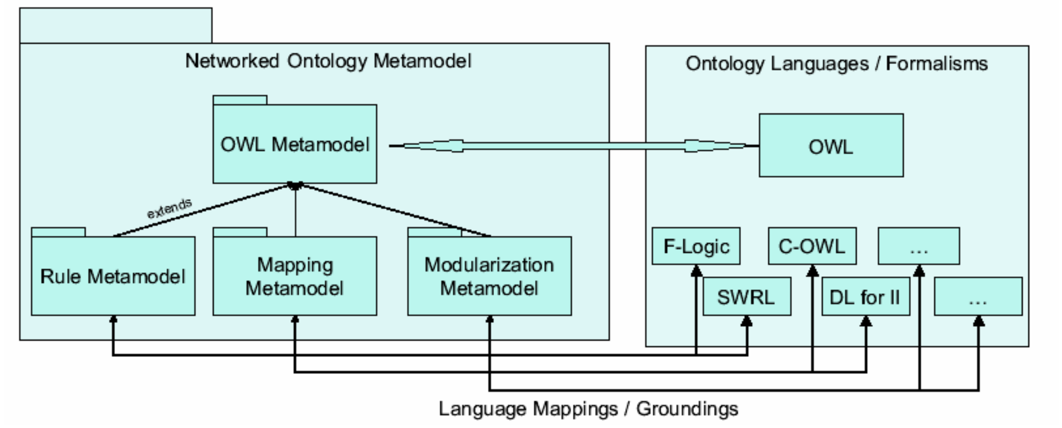


Figure 1. Modules of the Networked Ontology Metamodel [31]

The work here presented is focused on a subset of the NeOn networked ontology metamodel, the OWL ontology metamodel. The rule metamodel, the mapping metamodel and the modularization metamodel will be analyzed in the subsequent deliverable to D5.1.1 (which is D5.1.2). The approach followed is shown in Figure 2.

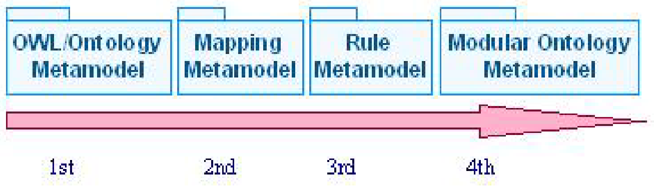


Figure 2. Approach to Analyze the Main Modelling Components

- The NeOn Modelling Components should be based on the reuse and/or adaptation of pre-existing knowledge sub-components. Different proposals to solve design or modelling problems (e.g. patterns, W3C and Knowledge Web best practices, etc.) are used in this work.
- Although guidelines for using the modelling components will be included in D5.1.4, in this deliverable we include a state of the art on the process followed for modelling ontologies, mappings, rules and work related to create modules. We also present some previous works on creating designs collaboratively because this collaboration issue is an important aspect of the NeOn project.

The deliverable is structured as follows:

Chapter 2 presents the State of the Art based on the analysis of previous works on modelling different components (such as Ontologies, Mappings, Rules, Modules, etc.).

Chapter 3 provides the rationale for building the inventory of *NeOn Ontology Modelling Components* (also considered as OWL-based design patterns). This first version of the inventory is divided into three different types: *logical patterns* (elements of the OWL module from the NeOn networked ontology metamodel [31], or compositions of those elements), *architectural patterns* (logical patterns or compositions of them that are used exclusively in the design of an ontology) and *content patterns* (instantiations of logical patterns or compositions of them). Patterns in these categories are described in detail in Chapters 4, 5, and 6, respectively.

Chapter 7 provides some conclusions on the work presented and future ideas related with the use of the inventory of patterns proposed for modelling ontologies. This section also includes the work to be included in D5.1.2 (next version of this deliverable) and in D5.4.1.

Finally, an annex collecting some definitions/clarifications of the terminology used is provided.

2. State of the Art of Modelling Ontology Components

The work provided in this deliverable is based on the NeOn networked ontology metamodel [31], which consists of several modules (the OWL ontology metamodel, the mapping metamodel, the rule metamodel, and the modularization metamodel).

Thus, in this chapter we present a review of the literature related to the creation of ontologies, mappings, rules, and modules. The chapter also includes previous works on creating designs collaboratively as the collaborative development is an important aspect in the NeOn project.

2.1. Analysis of Previous Processes for Creating Ontologies

One of the first decisions to make by any ontology developer when creating ontologies is which knowledge representation paradigm (frames, description logic, first (and second) order logic, semantic networks, etc.) to use to formalize the ontology to be created [28]. Not all the existing knowledge representation paradigms have the same expressiveness nor do they reason in the same way. It is appropriate to mention here that there are important connections and implications between the knowledge representation paradigms and the languages used for implementing the ontologies under a given knowledge representation paradigm. That is, an ontology formalized with description logic can be implemented in a language (for example, OWL) which has description logic as the underlying knowledge representation paradigm.

In this chapter we focus on the most commonly used paradigms, namely, description logic (DL) and frames. Although these two paradigms have many similarities: both represent concepts in the domain of discourse and relationships between them, they also have important differences in their semantics and the implications of their definitions [65].

The major differences between DL and Frames can be summarized as follows [65]:

- Unique name assumption (UNA). In Frames, if two objects have different names, they are assumed to be different. But, in DL, no such assumption is made.
- Closed world assumption (CWA) vs Open world assumption (OWA). In Frames (that assumes CWA) if something is absent, then it is supposed to be false. However, in DL (that assumes OWA) something is false only if it contradicts other information.
- Assertion vs Classification. In Frames, all ‘subclassOf’ relations must be asserted explicitly. That is, necessary conditions are described for instances of a class. However, in DL, ‘subclassOf’ relations can be inferred based on the class definition, by means of necessary and sufficient conditions to recognize members of a class.

In practice, the aforementioned differences lead to differences in the modelling style [65]. In the case of Frames, an ontology developer focuses on deciding which are the implications of belonging to a particular class. But, in the case of DL, an ontology developer thinks in terms of necessary and sufficient conditions to define a class.

Based on the previous differences and on the existing literature, we decided to divide this section into two different parts, one centered on processes for creating DL and OWL ontologies (Section 2.1.1) and the other, on processes for creating frame ontologies (Section 2.1.2), respectively.

2.1.1. Analysis of Previous Processes for Creating DL and OWL Ontologies

Description Logics (DLs) [4] are a family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally understandable way. The name *description logics* is based on two properties of the formalism: on the one hand, the important notions of the domain are described by concept descriptions. This means that expressions are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors provided by the particular DL. On the other hand, DLs differ from their predecessors, such as semantic networks and frames, in that they are equipped with a formal, logic-based semantics.

Description Logics [34] are the basis for ontology languages such as OIL, DAML+OIL and OWL. The decision to base these languages on DLs was motivated by the requirement that not only key inference problems (such as class satisfiability and subsumption) should be decidable, but also that “practical” decision procedures and “efficient” implemented systems should be available.

The **OWL Web Ontology Language**[1] is a language for defining and instantiating Web ontologies. *Ontology* is a term borrowed from philosophy that refers to the science of describing the kinds of entities in the world and how they are related. We can say that ontologies have two main components: names for important concepts (*elephant*, *herbivore*, etc.) in the domain (which we want to model); and background knowledge or constraints on that domain (*elephant weight at least 100 Kg.*). It is important to mention that modelling an ontology is difficult in complex domains. An OWL ontology may include descriptions of classes, properties and their instances. Given such an ontology, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the semantics. These entailments may be based on a single document or on multiple distributed documents that have been combined using defined OWL mechanisms.

The OWL Web Ontology Language [34] actually consists of three sub-languages of increasing expressive power: OWL Lite, OWL DL and OWL Full. Like OWL’s predecessor DAML+OIL, OWL Lite and OWL DL are, basically, very expressive description logics with an RDF syntax. OWL Full provides a more complete integration with RDF, but its formal properties are less well understood, and key inference problems would certainly be much harder to compute.

This section includes: a brief description of how to model with DLs (Section 2.1.1.1); a summary of Alan Rector’s work, focused on common problems, errors, and misconceptions on understanding OWL DL (Section 2.1.1.2); an analysis of the processes for selecting and reusing ontologies (Section 2.1.1.3); and finally an analysis of patterns for modelling ontologies (Section 2.1.1.4).

2.1.1.1. Modelling with Description Logics (DLs)

In [6] it is mentioned that most conceptual models, including DLs, subscribe to an *object-centered* view of the world. Thus, their ontology includes individual objects, which are associated with each other through (usually binary) relationships and grouped into classes.

The paper [6] presents some notions about elementary modelling with DLs. The most significant issues are the following:

- Most of the information about the state of the world is captured by the inter-relationships between individuals. Binary relationships are modeled directly in DLs using *roles* and *attributes*.
- In order to avoid inadvertent errors during modelling due to confusion between a role and its converse—or between a role and the kind of values filling it—one heuristic is to use a natural language name that is asymmetric and to adopt the convention that the relationship $R(a; b)$ should be read as “a R b”.
- It is always important to distinguish functional relationships, like lentTo (a book can be lent to at most one borrower at any time) from non-functional ones, like hasBorrowed.
- Individuals are grouped into classes. Classes usually abstract out common properties of their instances, e.g., every book in the library has a call number. Classes are modeled by

[1] http://www.w3.org/TR/owl-guide/

concepts in DLs, and usually the common properties are expressed as subsumption axioms about the concept.

- One of the fundamental properties of DLs is the support for the distinction between primitive concepts (which offer necessary conditions for membership) and defined concepts (which offer necessary and sufficient conditions for membership).

The authors [6] present a conceptual modelling methodology using DLs. The main steps of the methodology are the following:

1. Identify the individuals one can encounter in the universe of discourse (UofD).
2. Enumerate concepts that group these values.
3. Distinguish independent concepts from relationship-roles.
4. Develop a taxonomy of concepts. Revisit this later and consider issues such as disjointness and covering for subconcepts.
5. Identify any individuals (usually enumerated values) that are of interest in all states of the world in this UofD.
6. Search systematically for part-whole relationships between objects, creating roles for them.
7. Identify other 'properties' of objects, and then identify general relationships in which objects participate.
8. Determine local constraints involving roles such as cardinality limits and value restrictions. Elaborate any concepts introduced as value restrictions.
9. Determine more general constraints on relationships, such as those that can be modeled by subroles or same-as. (The latter often corresponds to "inheritance" across some relationship other than IS-A, and have been mentioned in several places earlier.)
10. Distinguish essential from incidental properties of concepts, as well as primitive from defined concepts.
11. Consider properties of concepts such as rigidity, identifiers, etc., and use the techniques of [30] to simplify and realign the taxonomy of primitive concepts.

2.1.1.2. Analysis of Alan Rector's Work

This section deals with Alan Rector's work related to the development of ontologies. In his work, he presents the most common problems, errors, and misconceptions on understanding OWL DL as well as tips on how to avoid such pitfalls in building OWL DL ontologies.

In [51] it is mentioned that for most people it is very difficult to understand the logical meaning and potential inferred statements of any DL formalism, including OWL DL. However, in the literature there is little guidance on how to use OWL DL or a related DL formalism to model or create ontologies.

The most common problems or difficulties in understanding OWL DL are, according to [51], the following ones:

1. Failure to make all information explicit, assuming that information implicit in names is "represented" and available to the classifier.
2. Erroneous use of universal restrictions rather than existential ones as the default. Most of the newcomers to OWL use (erroneously) universal (*allValuesFrom*) rather than existential (*someValuesFrom*) as the default qualifier.
3. Open world reasoning. Normally, newcomers to OWL have used systems with closed world reasoning (e.g. databases, logic programming, constraint languages in frame systems, etc.) with negation as failure (i.e., if something cannot be found, it is assumed to be absent). In contrast to those systems, OWL uses open world reasoning with negation as unsatisfiability (i.e. something is false only if it can be proved to contradict other information in the ontology).

4. The effect of range and domain constraints as axioms. OWL allows general expressions to be used in axioms. Like domain and range constraints, axioms are global and do not necessarily appear close the classes affected.

Apart from the aforementioned problems, the authors [51] present the following additional difficulties:

1. Trivial satisfiability of universal restrictions, that “only” (*allValuesFrom*) does not imply “some” (*someValuesFrom*).
2. The difference between defined and primitive classes and the mechanics of converting one to the other.
3. Errors in understanding common logical constructs.
4. Expecting classes to be disjoint by default. The paper mentions that one of the most common errors in building ontologies in OWL is to omit the disjointness axioms, when taxonomies are being modeled within the ontologies.
5. The difficulty of understanding subclass axioms used for implication.

A common question from newcomers to OWL, as explained in [51], is how they should decide which classes to declare as “defined” when building ontologies. To help newcomers with this problem, the authors suggest three basic heuristics:

- Pragmatic: Do you want things to be classified under the given class automatically?
- Do you want to commit to a definition now? You can always return to the item and later on change it from primitive to defined.
- Philosophical. Can you define the given class completely? There are many things which are “natural classes” (classes of people, classes of animals, etc.) and which are virtually impossible to define completely, at least outside a highly technical context. In this case, the best way is to create the class as primitive and merely described. Therefore, a useful heuristic is that if the definition is becoming long or controversial, consider leaving the class as primitive.

A brief summary of guidelines [51] for avoiding the most common problems and difficulties in building ontologies in OWL DL is shown in Table 1.

Table 1. Guidelines on How to Avoid Common Errors and Misconceptions in Modelling OWL-DL Ontologies

1. Always paraphrase a description or definition before encoding it in OWL.
2. Make all primitives disjoint (which requires that primitives form trees).
3. Use someValuesFrom as the default qualifier in restrictions.
4. Be careful to declare defined classes defined (the default is primitive). The classifier cannot classify instances as belonging to a primitive class (except in the presence of axioms/domain/range constraints).
5. Remember the open world assumption. Insert closure restrictions if that is what you mean.
6. Be careful with domain and range constraints. Check them carefully if classification does not work as expected.
7. Be careful about the use of “and” and “or” (*intersectionOf*, *unionOf*).
8. Always have an existential (*someValuesFor*) restriction corresponding to every universal (*allValuesFor*) restriction, either in the class or one of its superclasses (unless you specifically intend the class to be trivially satisfiable) to spot trivially satisfiable restrictions early,.
9. Run the classifier frequently; spot errors early.

2.1.1.3. Analysis of Processes for Selecting and Reusing Ontologies

The increased number of ontologies available online makes it possible to start the process of building an ontology by reusing (at least partially) existing ones. A pre-requisite of such reuse activities is the availability of tools that allow selecting potentially useful ontologies.

As described in [54, 55], there has been a number of efforts towards providing ontology selection mechanisms that operate on the totality of online available ontologies. However, this field is still very young, so it is characterized by the development of individual methods rather than establishing a generic selection/reuse methodology. Based on our analysis of the field, we conclude that a methodology for selecting and reusing ontologies on a large scale Web still needs to be developed.

2.1.1.4. Analysis of Patterns for Modelling Ontologies

The term "*pattern*" [23] appears in English in the 14th century and derives from Middle Latin "*patronus*" (meaning "*patron*", and, metonymically, "*exemplar*", that is something proposed for imitation). In the seventies, the architect and mathematician Christopher Alexander introduced the term "*design pattern*" for shared guidelines that help solve design problems [2].

In the field of ontology design patterns, we can make a distintion between logical and conceptual design patterns [23]. In the first case, the W3C Semantic Web Best Practices and Deployment Working Group (SWBPD)[2] states that best practices are necessary to provide some hand-on support for developers and users of the Semantic Web. This group defines best practices as 'a consensus-based guidance designed to facilitate Semantic Web deployment within RDF and OWL'; and it proposes patterns for solving design problems for OWL (OWL design patterns), independently of a particular conceptualization, addressing logical problems thus. In the second case, [23] proposes patterns for solving (in OWL or another logical language) design problems for the domain classes and properties that populate an ontology, addressing content problems.

In [48] some well known Semantic Web best practices (in particular those related to W3C activities) are analyzed to present them in a so-called cook-book style. This make it easier for teams to check whether the best practices are related to the modelling problems they concern and, if so, to apply them. The so-called cook-book style covers the following aspects of a concrete best practice:

- the problem(s);
- solutions: ingredients; required materials (ontology expressive power); and examples;
- tips (discussions on, e.g., pros and cons).

This section includes as examples two of the patterns proposed by the SWBPD, which are the case of modelling n-ary relations and that of representing classes as property values.

The case of modelling n-ary relations using OWL and RDF is studied in [46]. In order to address such case three main issues are identified and two modelling patterns are proposed.

- ❑ The first pattern is that of using a class to represent the n-ary relation (i.e., to reify the relation). The authors underline that this solution limits the use of many OWL constructs thus introducing maintenance problems (e.g., local range and cardinality restrictions).
- ❑ The second pattern is that of using an ordered list in order to represent the n–1 participants to the relation and connect such a list to the remaining one which is considered to have a special role in the relation.

OWL Full and RDFS do not put any restrictions on assigning a class to the value of a property, while OWL DL and OWL Lite properties can not have classes as their value. [44] elaborates this case and presents various alternative modelling approaches:

- ❑ To use classes as property values directly is the most intuitive approach if it is not required to be compatible with OWL DL.

[2] http://www.w3.org/2001/sw/BestPractices/

- To create a special instance of the class to be used as property values. This approach requires an implementation of this not so common feature, which has negative effects on the interoperability of applications and thus increases maintenance costs.
- To create a parallel hierarchy of instances as property values. This approach is pretty costly (since two parallel hierarchies have to be maintained) but cleaner in terms of modelling. Furthermore, a DL reasoner can infer transitive relations along the ad-hoc created hierarchy.
- To define a class by means of a class expression which contains a special restriction on the property of interest.
- To use the class as the value for an annotation property. With this approach is not possible to define restrictions on the property because it is defined as an annotation property, and DL reasoners will ignore it.

[23] states that experience in ontology engineering suggests recurrence of typical conceptual patterns emerging out of different ontology projects (although the projects themselves are aimed at different tasks and involve experts having heterogeneous backgrounds). These emerging patterns include e.g. *participation* (involving objects taking part in events) and the *role←→task* pattern (that allows talking about the temporary *roles* that objects can play, and about the *tasks* that events allow to execute, both in the framework of biological processes and in social activities such as e.g. a workflow).

In the paper [23], the notion of 'Content Ontology Design Patterns' ('CODePs') is introduced, and its difference with other sibling notions is discussed. Some examples of 'CODePs' are illustrated, and their usefulness in order to acquire, develop, and refine ontologies from either experts or documents is commented. 'CODePs', e.g., can be exploited as a tool to annotate 'focused' fragments of a reference ontology, i.e. the parts of an ontology containing the types and relations that underlay 'expert reasoning' in given fields or communities. They can be applied at different degrees of abstraction, and can be specialized or composed. 'CODePs' are expressible in OWL DL, and their high reusability and both formal and pragmatic nature make them suitable not only for isolated ontology engineering practices, but also (and especially) in distributed, collaborative environments like intranets, the Web or the Grid.

2.1.2. Analysis of Previous Processes for Creating Frame Ontologies

This section includes: a summary of the work carried out by D. McGuiness and N. Noy related to ontology development (Section 2.1.2.1); and a summary of the conceptual model proposed in METHONTOLOGY (Section 2.1.2.2).

2.1.2.1. Analysis of McGuiness' and Noy's Work

In [45] the authors define ontology as a formal explicit description of concepts in a domain of discourse. The properties of each concept describe various features and attributes of the concept (i.e., slots), and restrictions on slots. They also define knowledge base as a set of individual instances of classes.

The general principles of the development process of ontologies are described in [45]. First, the authors list a set of motivations why ontologies should be built:

- To share common understanding of the structure of information among people or software agents.
- To enable reuse of domain knowledge.
- To make domain assumptions explicit.
- To separate domain knowledge from the operational knowledge.
- To analyze domain knowledge.

Main considerations point out that there is no a single correct way or methodology for developing ontologies. It is always and necessarily an iterative process and the best approach differs depending on the application that one has in mind when performing this task.

One possible process for modelling an ontology can include the following summarized steps:

- Determine the scope of the domain to be described making use, for example, of competency questions.
- Reuse existing ontologies.
- Select terms by means of the terminology used by domain experts for describing the domain.
- Define the class hierarchy: this task can be performed by means of a top-down or bottom-up approach. It is also possible and common to use a combination of the two approaches (as described in [63]).
- Define the properties of classes, their value type and restrictions.
- Create instances.

For the step related to *defining classes and class hierarchies*, typical issues and possible ways to address them are shown in [45]. In this section we provide a summary of such guidelines:

- ✓ *Ensuring that the class hierarchy is correct*. The class hierarchy represents an “is-a” relation: a class A is a subclass of B if every instance of A is also an instance of B. A subclass of a class represents a concept that is a “kind of” the concept that the superclass represents.

  A common modelling mistake is to include both a singular and a plural version of the same concept in the hierarchy making the former a subclass of the latter. The best way to avoid such an error is always to use either singular or plural in naming classes.
- ✓ *Transitivity of the hierarchical relations*. Remember that a subclass relationship is transitive: if B is a subclass of A and C is a subclass of B, then C is a subclass of A.
- ✓ *Classes and their names*. It is important to distinguish between a class and its name: classes represent concepts in the domain and not the words that denote these concepts.

  The name of a class may change if we choose a different terminology, but the term itself represents the objective reality in the world.

  In more practical terms, the following rule should always be followed: synonyms for the same concept do not represent different classes. Synonyms are just different names for a concept or a term.
- ✓ *Avoiding class cycles*. We should avoid cycles in the class hierarchy. We say that there is a cycle in a hierarchy when some class A has a subclass B and at the same time B is a superclass of A.
- ✓ *Analyzing siblings in a class hierarchy*. Siblings in the hierarchy are classes that are direct subclasses of the same class. All the siblings in the hierarchy (except for the ones at the root) must be at the same level of generality.
- ✓ *How many is too many and how few are too few?* There are no hard rules for the number of direct subclasses that a class should have. However, many well-structured ontologies have between two and a dozen direct subclasses. Therefore, we have the following two guidelines:
  - If a class has only one direct subclass there may be a modelling problem or the ontology is not complete.
  - If there are more than a dozen subclasses for a given class then additional intermediate categories may be necessary.

  However, if no natural classes exist to group concepts in the long list of siblings, there is no need to create artificial classes (just leave the classes the way they are). After all, the ontology is a reflection of the real world, and if no categorization exists in the real world, then the ontology should reflect that.

- ✓ *When to introduce a new class (or not)*. One of the hardest decisions to make during modelling is when to introduce a new class or when to represent a distinction through different property values.

  There are several rules of thumb that help decide when to introduce new classes in a hierarchy.

  Subclasses of a class usually:

  - have additional properties that the superclass does not have, or
  - restrictions different from those of the superclass, or
  - participate in different relationships than the superclasses.

  In practical terms, each subclass should either have new slots added to it, or have new slot values defined, or override some facets for the inherited slots.

  However, sometimes it may be useful to create new classes even if they do not introduce any new properties. Classes in terminological hierarchies do not have to introduce new properties.

  When defining a class hierarchy, our goal is to strike a balance between creating new classes useful for class organization and creating too many classes.

- ✓ *A new class or a property value?* When modelling a domain, we often need to decide whether to model a specific distinction (such as white, red, or rosé wine) as a property value or as a set of classes again depends on the scope of the domain and the task at hand.

  If the concepts with different slot values become restrictions for different slots in other classes, then we should create a new class for the distinction. Otherwise, we represent the distinction in a slot value.

  If a distinction is important in the domain and we think of the objects with different values for the distinction as different kinds of objects, then we should create a new class for the distinction.

  A class to which an individual instance belongs should not change often.

- ✓ *An instance or a class?* Deciding whether a particular concept is a class in an ontology or an individual instance depends on what the potential applications of the ontology are. Deciding where classes end and individual instances begin starts with deciding what is the lowest level of granularity in the representation. The level of granularity is in turn determined by a potential application of the ontology. In other words, what are the most specific items that are going to be represented in the knowledge base?

  Individual instances are the most specific concepts represented in a knowledge base.

  Another rule can “move” some individual instances into the set of classes: if concepts form a natural hierarchy, then we should represent them as classes.

- ✓ *Limiting the scope*. As a final note on defining a class hierarchy, the following set of rules is always helpful in deciding when an ontology definition is complete: the ontology should not contain all the possible information about the domain: you do not need to specialize (or generalize) more than you need for your application (at most one extra level each way).

  Similarly, the ontology should not contain all the possible properties of and distinctions among classes in the hierarchy.

Other important issue mentioned in [45] is the need of *defining naming conventions for classes and slots and adhere to it*. Defining naming conventions in an ontology and then strictly adhering to these conventions not only makes the ontology easier to understand but also helps avoid some common modelling mistakes. Related to define naming conventions we need to take into account the following issues:

- The consistent use of capitalization for names is important, because this can greatly improve the readability of an ontology.
- The choice of use singular or plural names for calling the classes should be consistent throughout the whole ontology.
- The possibility of using prefix (e.g. has-) and suffix (e.g. –of) conventions in the names to distinguish between classes and slots.
- Strings such as “class”, “property”, “slot”, and so on should not be added to names.
- It is usually a good idea to avoid abbreviations in concept names.
- Names of direct subclasses of a class should either all include or not include the name of the superclass.

A brief summary of the guidelines provided in this section is shown in Table 2.

Table 2. Guidelines on How to Model Frame Ontologies

- To ensure that the class hierarchy is correct. A common modelling mistake is to include both a singular and a plural version of the same concept in the hierarchy making the former a subclass of the latter. The best way to avoid such an error is always to use either singular or plural in naming classes.
- To remember that hierarchical relations are transitive.
- To remember that synonyms for the same concept do not represent different classes.
- To avoid class cycles in the class hierarchy.
- To check that siblings in the class hierarchy are at the same level of generality.
- To remember that: if a class has only one direct subclass, there may be a modelling problem.
- To remember that in practical terms, each subclass should either have new knowledge added to it (new slot, new slot values, etc.).
- To remember that if concepts form a natural hierarchy, then they should be represented as classes, and not as instances.

2.1.2.2. METHONTOLOGY

This methodology was developed within the Ontology Engineering Group[3] at Universidad Politécnica de Madrid. METHONTOLOGY [21, 28] enables the construction of ontologies at the knowledge level.

This methodology proposes a set of steps for conceptual modelling during the ontology conceptualization activity. This activity deserves a special attention because it determines the rest of the ontology building. Its objective is to organize and structure the knowledge acquired during the knowledge acquisition activity, using external representations that are independent of the knowledge representation paradigms and implementation languages in which the ontology will be formalized and implemented.

Once the conceptual model is built, the methodology proposes to transform the conceptual model into a formalized model, which will be implemented in an ontology implementation language. That is, along this process the ontologist is moving gradually from the knowledge level to the implementation level, slowly increasing the degree of formality of the knowledge model so that it can be understood by a machine.

The ontology conceptualization activity in METHONTOLOGY organizes and converts an informally perceived view of a domain into a semi-formal specification using a set of intermediate

[3] http://www.oeg-upm.net/

representations (IRs) based on tabular and graph notations that can be understood by domain experts and ontology developers. METHONTOLOGY proposes to conceptualize the ontology using a set of tabular and graphical IRs that extend those used in the conceptualization phase of the IDEAL methodology for knowledge-based systems development [29]. These IRs bridge the gap between the people's perception of a domain and the languages used to implement ontologies.

When dealing with ontologies, ontologists should not be anarchic in the use of modelling components in the ontology conceptualization activity. They should not define, for instance, a formal axiom if the terms used to define the axiom are not precisely defined on the ontology. METHONTOLOGY includes in the ontology conceptualization activity a set of steps/tasks for structuring knowledge, as shown in Figure 3. The figure emphasizes the ontology components (concepts, attributes, relations, constants, formal axioms, rules, and instances) attached to each task/step, and illustrates the order proposed to create such components during the ontology conceptualization activity. This modelling process is not sequential as in a waterfall life cycle model, though some order must be followed to ensure the consistency and completeness of the knowledge represented. If new vocabulary is introduced, the ontologist can return to any previous task/step.

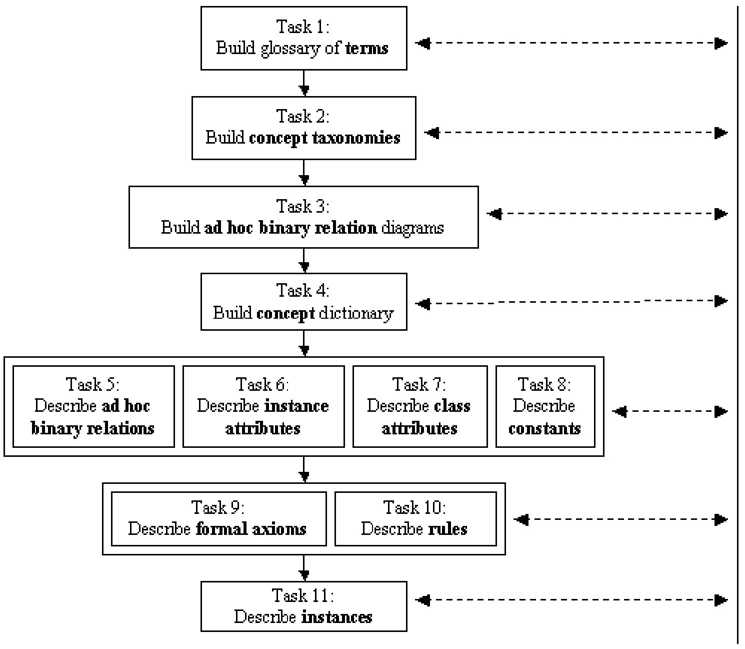

Task 1:
Build glossary of **terms**
Task 2:
Build **concept taxonomies**
Task 3:
Build **ad hoc binary relation** diagrams
Task 4:
Build **concept** dictionary
Task 5:
Describe **ad hoc binary relations**
Task 6:
Describe **instance attributes**
Task 7:
Describe **class attributes**
Task 8:
Describe **constants**
Task 9:
Describe **formal axioms**
Task 10:
Describe **rules**
Task 11:
Describe **instances**

Figure 3. Tasks of the Ontology Conceptualization Activity according to METHONTOLOGY [28]

METHONTOLOGY authors' experience in building ontologies has revealed that ontologists should carry out the following tasks/steps (shown in Figure 3):

Task 1: To build the glossary of terms that identifies the set of terms to be included on the ontology, their natural language definition, and their synonyms and acronyms.

Task 2: To build concept taxonomies to classify concepts. The output of this task could be one or more taxonomies where concepts are classified.

Task 3: To build ad hoc binary relation diagrams to identify ad hoc relationships between concepts of the ontology and with concepts of other ontologies.

Task 4: To build the concept dictionary, which mainly includes the concept instances[4] for each concept, their instance and class attributes, and their ad hoc relations.

Once the concept dictionary is built, the ontologist should define in detail each of the ad hoc binary relations, instance attributes and class attributes identified on the concept dictionary, as well as the main constants of that domain.

Task 5: To describe in detail each ad hoc binary relation that appears on the ad hoc binary relation diagram and on the concept dictionary. The result of this task is the ad hoc binary relation table.

Task 6: To describe in detail each instance attribute that appears on the concept dictionary. The result of this task is the table where instance attributes are described.

Task 7: To describe in detail each class attribute that appears on the concept dictionary. The result of this task is the table where class attributes are described.

Task 8: To describe in detail each constant and to produce a constant table. Constants specify information related to the domain of knowledge, they always take the same value, and are normally used in formulas.

Once that concepts, taxonomies, attributes and relations have been defined, the ontologist should describe formal axioms (task 9) and rules (task 10) that are used for constraint checking and for inferring values for attributes. And only optionally should the ontologists introduce information about instances (task 11).

2.2. Analysis of Previous Processes for Creating Mappings

Ontology mapping is crucial when integrating different heterogeneous ontologies. Since some publications have already dealt with the state of the art of mapping and aligning, including an overview of existing tools in deliverables of other EU-projects [17, 19], this section focuses on methodologies for creating mappings. However, a clear methodology for creating mappings has not been published yet. The approach that seems closest to a mapping methodology is the mapping process defined by Ehrig [16] in his dissertation.

We will use his process [16] as a basis and will extend it with a couple of novel additions. In Section 2.2.1 we describe the general method proposed by Ehrig based on the material presented in his dissertation [16]. We include our additions within the framework of this method.

In Section 2.2.2 we explain the van Hage's candidate alignment selection method, that can be used in the Ehrig's general mapping process.

In Section 2.2.3 we describe two different methodological approaches for similarity computation between ontology candidates.

Finally, we exemplify the main steps of the process on a well-known mapping tool, the PROMPT suite (Section 2.2.4).

2.2.1. Ehrig's General Mapping Process

Ehrig defines a general mapping process, which can be considered as a methodology for creating mappings. Figure 4 illustrates the input, the output, and the six main steps of this general process.

[4] Although instances can be created when the ontology is used (after its construction) the ontologist can decide whether to model relevant instances or not. This field is optional.

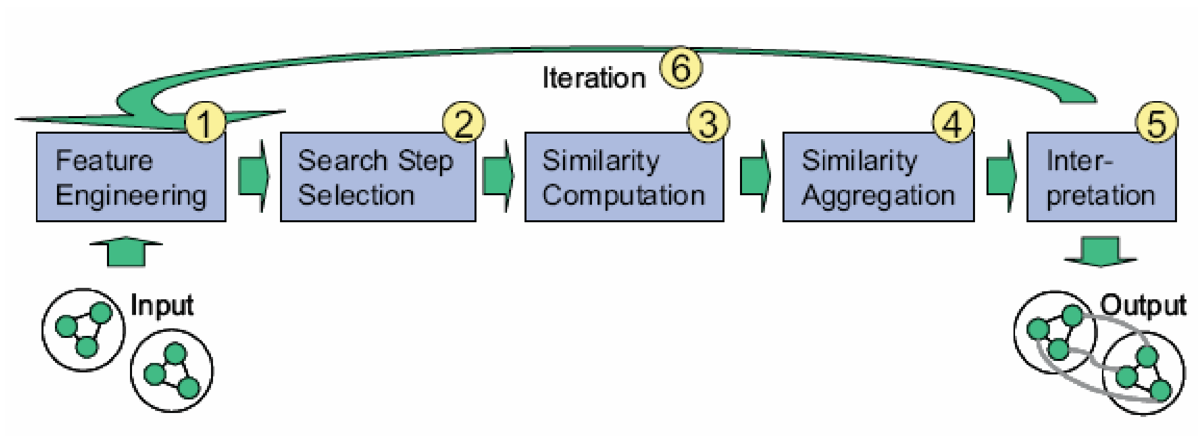


Figure 4. Alignment Process [16]

Input: Input for the process are two or more ontologies, which need to be aligned with one another. Additionally, it is often possible to enter a priori known (manual) alignments. A set of emerging mapping techniques also consider a variety of external sources as background knowledge for the ontology mapping (see an overview of such techniques in [53]). These can help to improve the search for alignments.

1. **Feature engineering**: Small excerpts of the overall ontology definition are selected to describe a specific entity. These excerpts are not arbitrary constructs, but have a specific meaning within the ontology—they represent a certain semantics. In a later step these features are then used for comparison. For instance, the alignment process may only rely on a subset of OWL primitives, possibly only the taxonomy, or even just the linguistic descriptions of entities (e.g., the label "*car*" to describe the concept *o1:car*).

2. **Search Step Selection**: The derivation of ontology alignments takes place in a search space of candidate alignments. This step may choose to compute the similarity of certain candidate concepts pairs $\{(e, f)|e \in O1, f \in O2\}$ and to ignore others (e.g., only compare *o1:car* with *o2:automobile* and not with *o2:hasMotor*). This step is very important because comparing every entity in one ontology with every entity in the other ontology is not always feasible, especially in cases when two large ontologies are compared. We are aware that van Hage has proposed a method to reduce the search space of candidate alignments [64]. We describe this method in Section 2.2.2.

3. **Similarity Computation**: For a given description of two entities from the candidate alignments this step indicates a similarity (e.g., $sim_{label}(o1{:}car,o2{:}automobile) = sim_{syntactic}(\text{“car”}, \text{“automobile”}) = 0$). In some related work this step does not only return a similarity, but actually a value of either 0 or 1; these are then called individual matchers and are based on a certain feature. Note also that a small amount of techniques tries to return semantic relations as mappings between entities instead of numeric similarity measures. An interesting methodological distinction can be made here between methods that only rely on the information provided within two ontologies to compute alignments and methods that exploit external sources to find out mappings. We elaborate on this distinction in Section 2.2.3.

4. **Similarity Aggregation**: In general, there may be several similarity values for a candidate pair of entities, e.g., one for the similarity of their labels and one for the similarity of their relationship to other entities. These different similarity values for one candidate pair have to be aggregated into a single aggregated similarity value (e.g., $(sim_{label}(o1{:}car,o2{:}automobile) + sim_{subconcepts}(o1{:}car,o2{:}automobile) + sim_{instances}(o1{:}car,o2{:}automobile))/3=0.5$). In the case of methods that work on a semantic rather than on a numeric value, the aggregation of the obtained similarities has to take possible logical contradictions into account. One of the current research topics is finding better ways to combine different values derived from various approaches, e.g. linguistic combination [35].

5. **Interpretation**: The interpretation finally uses individual or aggregated similarity values to derive alignments between entities. The similarities need to be interpreted (note that this step is not

needed if semantic mappings are derived in steps 3 and 4). Common mechanisms are to use thresholds [14, 47] or to combine structural and similarity criteria [15]. In the end, a proposition of alignment (or not) for the selected entity pairs is returned (e.g., align(o1:car) ='⊥', that is, no valid alignment was found).

6. **Iteration**: The similarity of one entity pair influences the similarity of neighboring entity pairs, for example, if the instances are equal, this affects the similarity of the concepts and vice versa. Therefore, the similarity is propagated through the ontologies by following the links in the ontology. Several algorithms perform an iteration over the whole process in order to bootstrap the amount of structural knowledge. In each iteration, the similarities of a candidate alignment are recalculated based on the similarities of neighboring entity pairs. Eventually, it may lead to a new similarity (e.g., *sim(o1:car,o2:automobile) = 0.85*), subsequently resulting in a new alignment (e.g., *align(o1:car) = o2:automobile*). Iteration terminates when no new alignments are proposed. Note that in a subsequent iteration, one or several of steps 1 through 5 may be skipped, because all features might already be available in the appropriate format or because similarity computation might only be required in the first round.

Output: The output is a representation of alignments, e.g., an alignment table indicating the relation $align_{O1,O2}$ and possibly with additional confidence values.

2.2.2. van Hage's Candidate Alignment Selection Method

As mentioned in Section 2.2.1, it is important to optimize the second step of the mapping process so that the search space of potentially relevant alignment candidates is reduced. van Hage *et al.* propose a methodology for this purpose, as follows (text adapted from [64]):

1. *Find a small set of high precision mapping relation as starting points, preferably distributed evenly over the ontologies*. This could be done with tools such as PROMPT.

2. *Manually remove all the incorrect relations*.

3. *For each correct relation select the concepts surrounding the subject and object concepts*. This can be accomplished in the following two steps:

   (a) Travel up the subclass hierarchy from the starting point. Go as far as possible provided that it is still clear what is subsumed by the examined concept, without having to examine the subtrees of the sibling concepts.

   (b) Select all subclasses of the two top concepts.

The effect of this method is that it reduces the search space by eliminating crossreferences between concepts in unrelated parts of the ontologies. Indeed, the high precision mappings obtained from the first two steps are likely to establish an initial link only between potentially relevant modules of the compared ontologies.

2.2.3. Methodologies for Similarity Computation

From a methodological perspective, similarities between ontology candidates can be computed by (1) only relying on the information provided by the two compared ontologies, and (2) exploiting domain knowledge that is external to the ontologies that are mapped.

In this section we overview the advantages and disadvantages of these two different approaches. Since the use of background knowledge is a rather new phenomenon and has not been much debated in the literature, we also provide a brief overview of existing approaches. A more detailed version of this discussion is available in [53].

2.2.3.1. Relying Only on the Mapped Ontologies

Most of the current mapping approaches rely entirely on the characteristics of the compared ontologies to establish mappings. As a result, they typically suffer from a number of problems. First of all, most approaches do not provide a formal semantics to the mapping structures they produce[5]. Therefore, it is difficult for reasoners to make use of these structures, e.g., to answer queries across ontologies [38]. Semantic Web tools, such as PowerAqua, which wish to reason on the results of mapping techniques require that the discovered mappings be expressed as semantic relations between the entities of the ontologies. An analysis of the state of the art of mapping systems presented in [19] explains to some extent the lack of approaches that can provide semantic mappings. The major factor seems to be that most systems combine a range of non-semantic techniques, such as terminological approaches (exploiting string similarity between labels), structural approaches (relying on the structure of the mapped ontologies), and extensional approaches (mapping concepts on the basis of shared instances). Only few systems rely on semantic techniques (also called model based approaches in [19]), thus exploiting the semantics both of the mapped ontologies, and of the mapping language, to infer mappings from the available knowledge. As a result, during the last Ontology Alignment Contest (OAC) [18] only one algorithm (CtxMatch [7]) was able to produce partial semantic mappings in the form of subconcept relations. The other techniques produce confidence based mappings that are derived by aggregating the output of terminological and structural algorithms. Unfortunately, this kind of low semantic (quantitative) relations are difficult to interpret and to exploit in reasoning procedures. Ideally, semantic techniques should produce meaningful relations between the mapped entities, on which further reasoning can be applied. They should focus on qualitatively good mappings that can be justified and explained through the knowledge and inferences used to deduce them.

A more important limitation is that current approaches to ontology mapping heavily rely on string-based and structure-based similarity measures [19] to identify which concepts to map. While these techniques can produce good results, there are also numerous examples in which they fail to find mappings which ought to be discovered. As already observed by [1], traditional methods fail when there is little lexical overlap between the labels of the ontology entities, or when the ontologies have weak or dissimilar structures. This observation has been verified to some extent in the last OAC [18]. In the first task of this contest where a base ontology was mapped to its systematically modified versions, the performance of most methods decreased significantly in the test cases where important changes have been performed to the labels and structures of the ontologies (tests 250 - 266). In fact, traditional techniques are based on the hypothesis of an equivalence between some forms of syntactic correspondences and semantic relations. While it is true that, in many cases, string and structural similarities can imply meaningful mappings, this hypothesis is far from being always verified. For instance, the relation between the concepts *Beef* and *Food* may not be discovered on the basis of syntactical considerations, but becomes obvious when considering the meaning of these concepts (their semantics). By ignoring such semantics, syntactic techniques fail to identify several important mappings.

2.2.3.2. Exploiting Background Knowledge

Section 2.2.3.1 suggests that the meaning of the mapped concepts should be considered to discover meaningful and syntactically unidentifiable mappings. Unfortunately, while meaning on the Semantic Web is expressed using ontologies, in the case of ontology mapping, the constituents of a mapping can only be given meaning in the context of their own distinct ontology, which cannot cover both the source and target elements, as well as the relation linking them. In other words, a semantic mapping between two ontologies could only be interpreted in a larger domain than the ones of these ontologies.

Therefore, in order to achieve semantic mapping, the integration of external knowledge is required as a way to cover both input ontologies and to fill the semantic gap between them. So far the following types of background knowledge have been used in mapping:

[5] A notable exception is the CtxMatch/S-Match algorithms.

1. **WordNet** is one of the most often used sources of background knowledge. For example, CTxMatch [7] (and its follow-up, SMatch [27]) translates ontology labels into logical formulae between their constituents, and maps these constituents to corresponding senses in WordNet. A SAT solver is then used to derive semantic mappings between the different concepts. When using WordNet, it is important to be aware that it is a lexical resource (rather than a truly semantic resource), relating terms by using terminological relations like synonymy or hypernymy. Therefore, it can be seen as a source of linguistic knowledge, useful in relating labels during the terminological step of a matching procedure.

2. **Reference Domain Ontologies**. Another approach is to rely on a reference domain ontology as a semantic bridge between two ontologies. In [1], the authors have experimentally proven that state of the art matchers fail to satisfactorily match two weakly structured vocabularies of medical terms. As a solution, they propose to use the DICE ontology as a source of background knowledge. Terms from the two vocabularies are first mapped to so called “anchor terms” in DICE and then their mapping is deduced based on the semantic relation of the anchor terms. As such, the obtained mappings can describe a larger variety of semantic mappings between terms, not just equivalence. Similarly, [60] presents a case study in the medical domain where mappings between two ontologies are inferred from manually established mappings with a third ontology, and by using the reasoning mechanisms permitted by the C-OWL language.

The advantage of these approaches is that they use richly axiomatized ontologies as background knowledge and therefore guarantee the semantic nature of the mappings. However, a weakness is that the appropriate reference ontology needs to be manually selected prior to mapping. As already pointed out, in many scenarios this approach is unfeasible, as we might not know in advance which terms from which ontologies we may want to map. Even in the cases where a reference ontology can be manually selected prior to performing the mapping, there is no guarantee that such an ontology actually exists.

3. **Online textual resources** can provide an important source of background knowledge. van Hage *et al.* [64] rely on the combination of two “linguistic ontology mapping techniques” that exploit online available textual sources to resolve mappings between two thesauri in the food domain. On the one hand, they use Google to determine subclass relationships between pairs of concepts using the Hearst pattern based technique introduced by the PANKOW system [12]. On the other hand, they exploit the regularities of an online cooking dictionary to learn hypernym relations between concepts of the source and target ontologies.

The strength of this approach is that it reduces the high cost of establishing adequate background knowledge. Indeed, the background knowledge sources are dynamically discovered and used [64]. There is no need for a manual and domain dependent ontology selection task prior to mapping. The drawback is that the right knowledge has to be extracted first. However, knowledge extraction techniques generally lead to considerable noise and so, reduce the quality of the mapping (e.g., Mayonnaise v Cold). Therefore, without human validation, online texts cannot be considered as reliable semantic resources.

4. **Online available ontologies**. The above described techniques show that the use of background knowledge overcomes the major limitations of syntactic approaches: it allows obtaining semantic relations even between dissimilar ontologies. However, existing approaches either 1) rely on an a priori selected reference ontology or, if they acquire knowledge dynamically, 2) suffer from the noise introduced by knowledge extraction techniques. As a result, they are not suitable for use by novel Semantic Web tools, such as PowerAqua, which require both that the returned mappings be semantically sound and that the relevant background knowledge be dynamically selected, at run-time.

The technique presented in [53] is based on the hypothesis that the growing amount of online available semantic data which makes up the Semantic Web can be used as a source of background knowledge in ontology mapping in a way that overcomes these limitations. Indeed, this large-scale, heterogeneous semantic data collection provides formally specified knowledge thus guaranteeing the semantic quality of the derived mappings. Moreover, the size and heterogeneity of the collection makes it possible to dynamically select and combine the appropriate knowledge and to avoid the manual selection of a single, large ontology.

2.2.4. PROMPT-Suite: An Explanatory Example

The features most of the current mapping tools provide can be mapped to the mapping process introduced in Section 2.2.1. Such mapping process will be shown using PROMPT and Anchor-PROMPT as an example:

The PROMPT-Suite [47] consists of different approaches tackling different questions around ontology alignment, mainly for ontology merging.

1. **Feature Engineering**: The original PROMPT only uses labels. The labels could be taken, e.g., from an RDF(S) ontology. Anchor-PROMPT uses several different relations of an ontology as shown in Table 3.

2. **Search Step Selection**: PROMPT relies on a complete comparison. Each pair of entities from the ontologies is checked for similarity.

3. **Similarity Computation**: On the one hand, PROMPT determines alignment based on whether entities have similar labels. It checks for identical labels as shown in Table 3. On the other hand, Anchor-PROMPT traverses paths between anchor points. The anchor points are entity pairs already identified as being equal, e.g., based on their identical labels. Along these paths, new alignment candidates are suggested. Paths are traversed along hierarchies as well as along other relations (see Table 4).

4. **Similarity Aggregation**: As PROMPT uses only one similarity measure, aggregation is not necessary. Anchor-PROMPT however needs to aggregate, which unfortunately is not explained in detail, but points to an average calculation.

5. **Interpretation**: PROMPT presents the entity pairs that have identical labels to the user. Anchor-PROMPT applies a threshold before doing so. For these pairs, chances are high that they are actually the same. The user manually selects the ones he deems to be correct, which are then merged. PROMPT and Anchor-PROMPT are therefore semi-automatic tools for ontology alignment.

Table 3. Features and Similarity Measures in PROMPT/Label

Comparing	No.	Feature	Similarity Measure
Entities	1	(label, X_1)	equality (X_1, X_2)

Table 4. Features and Similarity Measures in Anchor-PROMPT

Comparing	No.	Feature	Similarity Measure
Concepts	1	(label, X_1)	equality (X_1, X_2)
	4	(direct relations, Y_1)	path (Y_1, Y_2)
	6	all (superconcepts, Y_1)	path (Y_1, Y_2)
	7	all (subconcepts, Y_1)	path (Y_1, Y_2)
Other Entities	1	(label, X_1)	equality (X_1, X_2)

6. **Iteration**: The similarity computation does not rely on any previously computed entity alignments. One round is therefore sufficient in PROMPT. In Anchor-PROMPT, iteration is done to allow manual refinement. After the user has acknowledged the proposition, the system recalculates the corresponding similarities and presents new merging suggestions.

2.3. Analysis of Previous Processes for Creating Rules

This section covers related work to create rules and presents two rule languages important for NeOn: SWRL and F-Logic. A detailed survey on combining rules and ontologies can be found in [3].

The exact definition of a *rule* may vary depending on the kind of rule formalism or language used. The general idea of a rule is to have a way to express that a certain situation/action leads to another situation/action. An example may be: *if it rains (antecedent), the street is wet (consequent)*. Speaking in a more formal way: If the antecedent is true (the body of a rule), then the consequent (the head of a rule) must also be true.

While there is no official standard Ontology Rule language yet, there are several proposals building on different logic paradigms. Currently, the Rule Interchange Format (RIF)[6] Working Group is developing an interlingual rule format based on W3C standards.

In Section 2.3.1, we provide pointers to different works to facilitate the process of creating a rule, either graphically or with a rule table. Sections 2.3.2 and 2.3.3 give an overview of current prominent rule language proposals for ontologies and thus for the Semantic Web. They are based on different paradigms and have different strengths and weaknesses.

2.3.1. Process of Creating a Rule

There are no reported methodologies or processes for creating rules. This might be so because the main problems which users run into when creating rules are very likely due to difficulties with the formalization of the rule as opposed to the content. The complicated syntax might be difficult to understand or write for the not so experienced user. So let us assume a person knows that the street is wet after it has rained, how does he or she formalize this in a rule language?

One way of facilitating the creation of rules when the content is already clear is by allowing users to create rules graphically. Saartje Brockmans *et al.* [8] introduce a MOF compliant profile for SWRL rules that allows modelling rules visually using UML tools.

In METHONTOLOGY [28] rule definition is mentioned as one of the tasks of ontology engineering. Its approach is to use a rule table to define rules once the rules needed for the ontology are identified. The information that should be included in the table entry is: name, natural language description, the expression formally describing the rule, the concepts, the attributes and the relations to which the rule refers, and the variables used in the expression.

Using the template *if <conditions> then <consequent>*, rule expressions can be specified. The left-hand side of the rule should consist of conjunctions of atoms, whereas the right-hand side should be a single atom. An example can be found in [28].

2.3.2. SWRL

Although OWL is very expressive, it is restricted to obtain decidability. This suppose that it cannot express arbitrary axioms: the only axioms it can express are of a certain tree-structure. In contrast, decidable rule-based formalism such as function-free Horn rules do not share this restriction, but lack some of the expressive power of OWL: they are restricted to universal quantification and lack negation in their basic form. To overcome the limitations of both approaches, several rule extensions for OWL have been heavily discussed [66]. At the end of 2005, the W3C chartered a working group for the definition of a Rule Interchange Format [67]. One of the most prominent proposals for an extension of OWL with rules is the Semantic Web Rule Language (SWRL) [33]. SWRL proposes allowing the use of Horn-like rules together with OWL axioms.

The SWRL specifications, submitted to W3C in May 2004, include a high-level abstract syntax for Horn-like rules extending the OWL abstract syntax. An extension of the model-theoretic semantics from OWL provides the formal meaning for rules written in this abstract syntax. Moreover, besides

[6] http://www.w3.org/2005/rules/

the abstract syntax, SWRL allows an XML syntax based on the OWL XML presentation syntax, as well as an RDF concrete syntax based on the OWL RDF/XMI exchange syntax.

SWRL has a high expressive power but like that a high computational complexity. Moreover, SWRL becomes undecidable as rules can be used to simulate role value maps [58]. To balance the expressive power against the execution speed and termination of the computation, suitable subsets of the language can allow efficient implementations. This section introduces the available constructs in SWRL, the XML syntax, and the model-theoretic semantics. The original SWRL specifications are built on OWL 1.0. The minor changes in syntax and semantics to adapt to OWL 1.1 are incorporated in this overview.

2.3.2.1. Language Constructs

We present the SWRL grammar using standard BNF notation, and a similar syntax to the OWL 1.1 functional-style syntax, called the SWRL abstract syntax. Since SWRL extends OWL, this grammar is an extension of the OWL grammar.

The usual conventions for the BNF notation hold: nonterminal symbols are written in bold, e.g. **owlClassURI**; terminal symbols are written between single quotes, e.g. 'ObjectPropertyRange'; zero or more instances of a symbol are denoted with curly braces, e.g. {**description**}; alternatives are denoted with vertical bars, e.g. **fact** | **declaration**; zero or one instances of a symbol are written with square brackets, e.g. [**description**].

The SWRL extension of OWL defines a rule as an axiom:

axiom := **rule**

A rule in SWRL contains an antecedent, also referred to as *body*, and a consequent, also referred to as *head*. Since a rule is defined as an axiom and any OWL axiom can be annotated, a SWRL rule can contain annotations, too. Moreover, it can be given a URI as identification, assuring compatibility with OWL.

Both antecedent and consequent contain a number of atoms, possibly zero, where multiple atoms are treated as a conjunction in SWRL. Consequently, a rule actually says that *if* all atoms in the antecedent hold, *then* the consequent holds. An empty antecedent is treated as trivially true, i.e. satisfied by every interpretation, whereas an empty consequent is treated as false, i.e. not satisfied by any interpretation.

rule := 'Implies' '(' [**URI**] {**annotation**} **antecedent consequent** ')'

antecedent := 'Antecedent' '(' {**atom**} ')'

consequent := 'Consequent' '(' {**atom**} ')'

An atom in SWRL rules can have the following forms:

- An **OWL class description**, defined on a variable or an OWL individual. The atom holds if the value of the variable or the individual belongs to the class description.
- An **OWL data range** specification using a variable or a data value. The atom holds if the variable or the individual belongs to the data range.
- An **OWL property**. In case of an object property, the subject and object of the property are an individual or a variable. If the specified property is a datatype property, the atom takes an individual or a variable as the subject, and a variable or data value as the object. The atom holds if the object of the property is related to the subject by the specified property.
- A **sameAs** construct, defined on two objects, which are an individual or a variable. This construct is actually just some syntactic sugar and could be represented with other existing constructs. However, since it is very useful in practice, the SWRL specifications define it as one of the basic constructs. The atom holds if the two terms are the same.
- A **differentFrom** construct, defined on two objects, which are an individual or a variable. Just as well as *sameAs*, also *differentFrom* is syntactic sugar but very practical. The atom holds if

the two terms are different.

- A **built-in** construct with a built-in ID and a set of variables and data values. Built-ins are classified into seven modules, such as built-ins for lists or built-ins for comparisons. Implementations could select the modules to be supported. The atom holds if the relation defined through the built-in ID holds for the specified terms.

An atom in SWRL rules can be defined as follows:

atom	:=	**description** '(' **i-object** ')' |
		dataRange '(' **d-object** ')' |
		objectPropertyURI '(' **i-object i-object** ')' |
		dataPropertyURI '(' **i-object d-object** ')' |
		'sameAs' '(' **i-object i-object** ')' |
		'differentFrom' '(' **i-object i-object** ')' |
		'builtIn' '(' **builtinID** {**d-object**} ')'
builtinID	:=	**URI**
i-object	:=	**i-variable** | **IndividualURI**
d-object	:=	**d-variable** | **constant**
i-variable	**:=**	'I-variable' '(' **URI** ')'
d-variable	:=	'D-variable' '(' **URI** ')'

At last, when it comes to the variables occurring in the atoms of the rules, their scope is always limited to the rule itself. To cope with undecidability, Boris Motik *et al.* proposed the so-called DL-Safe Rules [43], which are a decidable subset of SWRL. The basic idea is to require that each variable in a rule occurs in a non-DL-atom in the rule body. Through this restriction, rules are only applicable to individuals explicitly introduced in the ABox, which causes DL-Safe Rules to be decidable. DL-Safe Rules can be interpreted using a reasoner such as KAON2[7], developed by Boris Motik. Examples and a very detailed description of the KAON2 infrastructure together with the theoretical implications can be found in [42].

2.3.2.2. Syntactic Representation and Semantics

Next to the abstract syntax just presented, SWRL also provides an XML exchange syntax defined in the XML schema language and an RDF concrete syntax. An extension of the OWL 1.1 model-theoretic semantics provides a formal meaning for SWRL ontologies by defining extensions of OWL 1.1 interpretations. These extensions, called bindings, also map variables to elements of the domain. Both the different syntaxes and the model-theoretic semantics of SWRL are described in detail in [33].

2.3.3. F-Logic

F-Logic (Frame-Logic), for which [36] can be referred to for a full account, is a deductive, object-oriented and frame-based logic. Originally, the language was developed for deductive and object-oriented databases at the Department of Computer Science of the State University New York in 1995. Later on, however, it has been applied for implementing ontologies. Among other modelling primitives, F-Logic provides constructs for defining declarative rules, which infer new information from the available information. Furthermore, queries that directly use parts of the ontology, can be asked. The semantics of F-Logic is well-defined as it will be highlighted at the end of this section. From a syntactic point of view, F-Logic is a superset of first-order logic. In this section, we describe first how F-Logic expressions are correctly built and which constructs are available for building F-

[7] http://kaon2.semanticweb.org/

Logic expressions. Then, at the end of the section, we address F-Logic's model-theoretic semantics. In this part, we use the first and current version of F-Logic and not the new one, which is being developed but still in a preliminary state at the moment of writing.

2.3.3.1. Language Constructs and Syntax

We now introduce the syntax and the object-model of F-Logic. Before introducing the different constructs, we shortly define the F-Logic alphabet. The F-Logic alphabet consists of a set of *predicate symbols P*, a set of *function symbols F*, and a set of *variables V*, where $F \cup V$ forms the set of **id-terms** which identify objects, methods and classes.

Rules:

An F-Logic program contains a set of rules and facts. **Rules** infer new information from available facts and consist of a head, an implication sign ('←'), and a body. The rule **head** is a conjunction of P-atoms or F-Molecules, both of which will be explained further, all connected through 'AND'. In the **body**, P-molecules and F-molecules can be connected by different predicate logic connectives: implies ('→'), implied by ('←'), equivalent ('↔'), and ('AND'), or ('OR'), and not ('NOT'). When appearing in the head, variables are introduced in front using a forall-quantifier ('∀'). In the body, they can appear anywhere using logical quantifiers exist ('∃') or forall ('∀'). The so-called safety-condition requires that a variable that appears in the head, must also appear in a positive P- or F-atom in the body to avoid infinite computations.

The example rule below states that an employee of one of the members of a laboratory union, gets technical support from the institution union.

$\forall X,Y\ X : \text{Person}[\text{getsTechnicalSupport} \rightarrow Z] \leftarrow$

$X : \text{Person}[\text{isEmployed} \rightarrow Y\,] \wedge Y : \text{Laboratory}[\text{isMember} \rightarrow Z].$

F-Atoms and F-Molecules:

Let c, c_1 , c_2 , m, o, r, r_1 , r_2 , ..., r_n and t be id-terms of the F-Logic alphabet. Then we can define an **F-atom** as an expression with one of the following forms:

• **instanceOf assertion**: $o : c$ denotes that object o is an instance of class c.

• **subclassOf assertion**: $c_1 :: c_2$ denotes that class c_1 is a subclass of class c_2 .

• **single-valued method signature**: $c[m => t]$ is a signature-atom specifying that the application of the single-valued (or functional) method m on an object of class c has as result an object of type t, where functional means that at most one object exists as value for the application of the method on an object.

• **multi-valued method signature**: $c[m =>> t]$ is a multi-valued signature-atom denoting that the application of the method m on an instance of class c has several possible objects of type t as result. Note that every multi-valued signature atom can be represented by a set of single-valued signature atoms.

• **single-valued method application**: $o[m \rightarrow r]$ expresses that the application of the method m on the object o has the object r as value. This atom type is also called data-atom.

• **multi-valued method application**: $o[m \text{ ->> } r_1 , r_2 , ..., r_n]$ expresses that r_1 , r_2 , ..., r_n are the result of the application of the method m on object o.

All method signatures and method applications can additionally have parameters. **F-molecules** group F-atoms together conjunctively, for example:

$X : \text{Researcher}[\text{authorOf} \rightarrow Y;\ \text{cooperatesWith} \rightarrow Z]$

is equivalent to

$X : \text{Researcher} \wedge X[\text{authorOf} \rightarrow Y\,] \wedge X[\text{cooperatesWith} \rightarrow Z]$

Note that atoms can also be nested in each other. For example, the value of a method application can be defined as an atom itself. Additionally, F-Logic provides some **built-in** features, like several comparison predicates, the basic arithmetic operators, and so forth.

P-Atoms and P-Molecules:

For compatibility with languages like Datalog, F-Logic allows **P-atoms**, which consist of a predicate symbol p ($\in P$) and a list of id-terms as parameters: $p(F_1 , ..., F_n)$. F-atoms and F-molecules can be nested into P-atoms (but this is not possible the other way around), which is then called a **P-molecule**.

2.3.3.2. Semantics

The F-Logic semantics is based on the fixpoint semantic of Datalog programs [62]. The evaluation starts with an empty object base, and rules and facts are evaluated iteratively (note that queries are not part of an F-Logic program). When the rule body is valid in the actual object base with certain variable bindings, these bindings are propagated into the rule head. Likewise, new information from rule heads or deduced due to closure properties is inserted into the object base. Until no new information can be obtained anymore, rules are continuously evaluated.

As with Datalog, the evaluation of a negation-free F-Logic program reaches a fixpoint which coincides with the unique minimal model of that program, which is defined as the smallest set of P- and F-atoms so that all closure properties and all facts and rules of the program are satisfied. As soon as a fixpoint is reached, the semantic of an F-Logic program is computed.

For a complete presentation of the F-Logic semantics, please refer to [36].

2.4. Analysis of Related Work to Create Modular Designs

Modularity is another key-factor of ontology design. In [24] the authors employ modularity as a quality-measure for ontologies, and to do so it elaborates on the material presented in [20]. According to these treatments, the modularity of an ontology is based on its adjustment to an existing repository of reusable components, which enhances the quality of the ontology both at design-time and at reuse-time. On the other hand, modularity depends on the appropriateness of the methodology used for design and requires a specification of reusable components. For example, it requires (one or more) libraries of ontologies, with indications of their provenance, specificity, application history, etc. Modular designs are not independent: they require a preliminary assessment based on topic and task. In the case of reusable generic components, such assessment should be performed at a high enough level of generality. On the other hand, modularity depends on topic assessment because we need to know what theories are needed in a certain ontology project. It also depends on task assessment because we need to know how much of a reusable theory is needed. This dependence causes a form of circularity: a reusable component has to be assessed against a task, but it is supposed to provide a ready-made solution to task assessment. There is no trivial solution to this circularity, and a good practice is to isolate the fragment as much as possible, and to import it. This approach is applied more effectively if a reusable component spots its content design patterns.

2.5. Analysis of Related Work to Create Designs Collaboratively

The NeOn deliverable D2.1.1 [11] presents a comprehensive overview of the State of the Art of research on collaboration. This notion is treated from three main perspectives: requirements in collaborative activities, tools for collaboration support, and matching requirements and tools.

This section includes a summary of some of the material of D2.1.1 that has relevance to D5.1.1.

- Requirements in Collaborative Activities

Kollock presented [37] a brief survey of the main studies (at the time when the World Wide Web was emerging) on principles that seem to underlie successful cooperative communities. The aim of the paper was to understand if such principles and best practices (or some of them) are applicable in building successful cooperative online communities. The paper provided a list of design principles identified by various authors. Such principles are still basic good practices for (online) communities creation and management: it should be arranged that individuals meet each other, that they are able to recognize each other, they must have information about how the other behaves, etc.

- Tools for Collaboration Support

[13] describes an application suite named Ontology Builder and Ontology Server (OBOS) developed for supporting the creation and maintenance of ontologies used in e-commerce and B2B applications.

[57] presents the application of a Compendium approach to Hypertext-Augmented Collaborative Modelling (HACM). Within the Advanced Knowledge Technology (AKT) consortium the AKTive Portal was designed to be a next generation portal infrastructure supporting the capture, indexing, dissemination and querying of information.

[56] studies Claimspotter, an open architecture based on the ScholOnto ontology. Claimspotter supports the semiformal (collaborative) annotation of scholarly documents.

[40] introduces the theoretical principles behind the ClaiMaker tool, a system designed to represent discourse in a semiotic way within the scholarly domain. More generally, the paper discusses the representational requirements for collaborative systems that support sensemaking and argumentation over contested topics. Sensemaking is intended to express and contest explicit, possibly competing views of the world. Supporting sensemaking, therefore, means supporting a way of annotating different interpretations of the same object or issue. This is what ClaiMaker does, with a theoretical backbone consisting of semiotic (in a saussurian fashion) and coherence relations, as in Mann and Thompson’s Rhetorical Structure Theory (RST).

[61] presents the integration of two existing tools (i.e., Compendium and I-X) for the Co-OPR project, the simulation of a personnel recovery mission. The experiment presented deals with decision-making support for a team collaborating on the same mission. In particular, Compendium has been used to support the collaboration between members of the team who were geographically distributed; I-X has been used as a tool to support a team whose members were physically in the same place. The paper underlines the effectiveness and usability of both tools when used together and provides a very pragmatic evaluation. It is focused mainly on the usability and utility of the provided functionalities.

- Matching requirements and tools

Tools that support collaboration are obviously developed on the basis of user requirements. Some valuable insights come from comparing such requirements and available tools so as to identify not only the technical gaps, but to rather determine which gaps can be bridged by advancing technology and which are instead unavoidable (i.e. which requirements are unsupportable).

What Ackerman [1] terms “the social-technical gap” is “the divide between what we know we must support socially and what we can support technically”, and is likely to be the highest challenge of Computer-Supported Cooperative Work (CSCW). What is relevant to NeOn is not only the description of social requirements in collective work, but also the description of the kind of support, that is technically difficult to achieve or, to put in different terms, the definition of an upperbound for the development of tools that support collaborative activities.

[49] claims that one of the main failures of human-computer interaction (HCI) is the treatment of turn-taking. They present experiments showing that HCI systems are not equipped for dealing with natural turn-taking issues, such as pauses, overlaps, and similar behavior. Although this applies to human-machine interaction, in the spoken dialogue domain there might be similar problems in collaborative activities between humans conducted over the Web, especially if the dialogue is carried out in a synchronous manner.

In the specifics of collaboration towards ontology development, [39] is a source of interesting points with respect to requirements and tool support. According to Lu, since modern ontologies are characterized by their huge size and high complexity, ontology engineering is to be considered an inherently collaborative activity, involving the effort of many domain experts and software developers which are often not co-located. This is especially prominent when not only the design stage, but also the whole ontology life cycle are considered. Throughout this work, 'collaboration' seems to be defined as a reiterated process, the output of which, at each of the involved stages, is the obtaining of a 'convergence of views'.

3. Inventory of NeOn Ontology Modelling Components

Within WP5 we have to analyze which are the main modelling components (we call them "*NeOn Modelling Components*") that permit teams (formed by ontology engineers, domain experts, etc) to model networks of ontologies collaboratively, and how this could be done according to some guidelines. We informally define a *modelling component* as any information resource that could be used in the activity of modelling.

As mentioned in Section 1.2, the work on modelling components (namely, design patterns) has an strong relation with the work on the NeOn networked ontology metamodel [31], carried out in WP1, which consists of several modules (OWL ontology metamodel, rule metamodel, mapping metamodel, etc.).

In D5.1.1 we provide the first version of the inventory of the main modelling components to model OWL ontologies (they are called "*NeOn Ontology Modelling Components*" or "*OWL-based design patterns*"), which are based on the elements of the OWL ontology metamodel, i.e., modelling components that enable teams to model OWL ontologies collaboratively (that is, a particular concept or a class, a particular relation or an object property, a concrete pattern which solves a particular problem, best practices, etc.).

The *OWL-based design patterns* presented in this deliverable are organized into three different types: logical patterns (described in Chapter 4), architectural patterns (described in Chapter 5) and content patterns (described in Chapter 6). These patterns are related to the NeOn networked ontology metamodel [31], described in WP1.

In general, the main difference between the networked ontology metamodel perspective and the collaborative design perspective resides in the social nature of the second one. Semantically speaking, the extensional content is the same, but the social perspective of collaborative design additionally provides use cases and an historical depth on how the pattern has been used or evaluated in the past in realistic settings. In other words, design puts logical solutions in the context of actual modelers that use logic to specify conceptualizations in an environment, according to workflows, available resources, available knowledge, requirements, argumentation, etc. An example of how design impacts on logic is that the first drastically reduces the combinatorial space of logical solutions in a certain (social) context.

Here, we present the definition of each type of pattern (logical, architectural, and content) and their relation to the NeOn networked ontology metamodel.

Logical ontology design pattern (LP): semantically speaking, equivalent either to the elements of the OWL module from the NeOn networked ontology metamodel [31], or to compositions of those elements. Speaking of design, an LP is a content-independent structure, i.e. an untyped structure expressed only with a logical vocabulary. E.g., in the case of OWL, it can only be instantiated by elements in the owl namespace. An LP can be applied more than once in the same ontology to solve similar modelling problems. For example, one can instantiate the LP 'subClassOf relation' for each user-defined subclass axiom needed. An LP affects only a specific and delimited part of an ontology, i.e. it does not affect the overall shape of an ontology.

Examples of LPs listed here include the definition of a class, the definition of a class as subclass of another class, and the definition of n-ary relations. In this context, the D1.1.1 metamodel [31] defines the vocabulary for the most primitive OWL logical design patterns. The analogous of LPs in software engineering are programming design patterns, as those described in [22].

Architectural ontology design pattern (AP): equivalent to LPs (or compositions of them) that are used exclusively in the design of an ontology. An AP is then also a content-independent structure. In other words, an AP is supposed to characterize the overall structure of an ontology. In simple terms, an AP dictates 'how an ontology should look like'. A basic example of AP is *taxonomy,* which is only composed of the LPs 'Class' (primitive or defined) and 'subClassOf relation', so that an ontology to which it applies can only be designed by applying those LPs. The analogous of APs in software engineering are software architecture patterns such as those described in [10].

Content design pattern (CP): is an instance of an LP (or compositions of LPs). A CP is a typed structure expressed with a domain specific (non logical) vocabulary. A CP represents and solves a domain modelling problem and affects the (limited) part of the ontology dealing with that domain modelling problem [23]. An example of CP is 'Plans', consisting in an ontology that provides a vocabulary and axioms to characterize plans as constituted of roles, tasks, parameters, and goals. The typical use of a CP is its specialization within a particular domain for a particular task. For example, 'Plans' can be specialized as a CP 'MedicalGuidelines' for e.g. representing nurse-oriented guidelines. 'MedicalGuidelines' on its turn can be instantiated into a specific medical guideline, e.g. 'CuteHospitalNurseGuidelineToOrthopedicDepartmentAssistance'.

Patterns of the above categories are described with a unified template, inspired by previous work on Software Engineering Patterns [22], with the following items/slots:

- *General Information*, which includes name, identifier and ontology modelling component type. These slots are mandatory.

- *Use Case*, which includes the problem to be addressed. The content for this slot is slightly different in each type.

- *Ontology Design Pattern*, which includes the proposed solution in different formats. Here the content could change depending on the type of ontology modelling component.

- *Relations to other ontology model components*, which refers to possible relationships (use, specialize, etc.) with other components. This slot is optional.

- *Comments*, which refers to remarks for clarifying the use of the component. This slot is also optional.

Because of some slots in the aforementioned template are dependent on the type of ontology modelling component, we have decided to provide specific explanation on some slots for each type, following the general structure defined in the unified template.

In D5.1.2 we will provide the second version of the inventory of the *NeOn Ontology Modelling Components* (reusing and extending the results to be presented in D2.5.1 within WP2) and the inventory for the rest of modelling components (namely, NeOn Mapping Modelling Components, NeOn Rules Modelling Components, and NeOn Module Modelling Components). The aforementioned approach is shown in Figure 5. In D5.4.1 we will provide the guidelines for modelling networks of ontologies based on the first version of the inventory presented in D5.1.1.

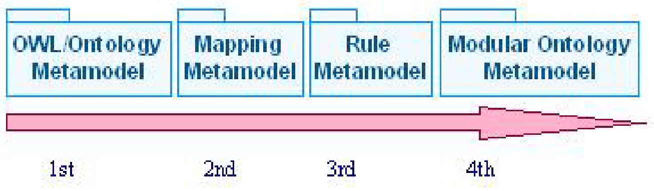


Figure 5. Approach to Analyze the Main Modelling Components

4. NeOn Ontology Modelling Components: Logical Patterns

In this chapter we present the particular template for describing logical patterns in the inventory; we also include samples of logical patterns using the template.

4.1. Template for Logical Patterns

For dealing with ontology modelling components, which are here considered as logical patterns (that is, elements of the OWL module from the NeOn networked ontology metamodel [31], or compositions of those elements), we propose the template shown in Table 5.

Table 5. Template for Logical Patterns

Slot	Value
General Information	
Name	Name of the component
Identifier	An acronym composed of: component type + component + number
Type of Component	Logical Pattern (LP)
Use Case	
General	Description in natural language of the general problem addressed by the modelling component.
Examples	Description in natural language of some examples for the general problem.
Ontology Design Pattern	
Informal	
General	Description in natural language of the general solution provided by the modelling component, refering to the NeOn OWL Ontology Metamodel defined in D1.1.1 [31].
Examples	Description in natural language of the solution applied to the examples.
Graphical	
(UML) Diagram for the General Solution	Graphical representation of the general solution provided, taking into account the UML Profile proposed in [9].
(UML) Diagram for Examples	Graphical representation of the solution provided, using examples and taking into account the UML

	Profile proposed in [9].
Formalization	
General	Formalization of the pattern in terms of the NeOn OWL Ontology Metamodel [31].
Examples	Formalization of the examples (using abstract syntax for OWL code).
Relationships	
Relations to other modelling components	Description of any relation to other modelling componens (use, specialize, etc.).
Comments	
Comments	Remarks for clarifying the use of the modelling component.

4.2. Inventory of Logical Patterns

To date, we have identified the following *NeOn Ontology Modelling Components* considered as Logical Patterns: primitive class, defined class, 'subclassof' (subsumption) relation between classes, multi-inheritance between classes (using 'subClassOf'), equivalence relation between classes, object property, 'subpropertyof' relation between object properties, datatype property, 'subpropertyof' relation between datatype properties, existential restriction, universal restriction, intersection of classes, union of classes, property domain, property range, functional property, inverse property, transitive property, symmetric property, individual, cardinality restriction, closure axiom, disjoint classes, covering axiom, defining n-ary relations, representing classes as property values, qualified cardinality restrictions (QCRs), representing specified values in OWL: "value partitions" and "value sets", and XML Schema Datatypes in RDF and OWL.

However, in this document, the inventory of *NeOn Ontology Modelling Components* that are Logical Patterns only includes a sample of the aforementioned patterns; the rest of the patterns and probably some new ones will be included in D5.1.2. (the subsequent deliverable to D5.1.1).

The current inventory of *NeOn Ontology Modelling Components* considered as Logical Patterns includes as a sample the following ones: primitive class, defined class, 'subclassof' relation between classes, multi-inheritance between classes (using 'subClassOf'), equivalence relation between classes, object property, 'subpropertyof' relation between object properties, datatype property, existential restriction, universal restriction, union of classes, individual, disjoint classes, covering axiom, defining n-ary relations, and representing specified values in OWL.

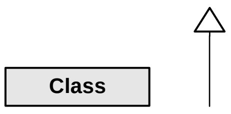
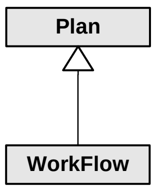
4.2.1. Logical Pattern for Modelling a Primitive Class

A class that only has necessary conditions is known as a *Primitive Class* [32]. Necessary conditions can be read as: *if something is a member of this class then it has necessarily to satisfy these conditions*. Classes that only have necessary conditions are also known as *'partial' classes*.

Determining whether a concept should be *primitive* or *defined* is a key aspect of modelling ontologies using DL, as the knowledge representation paradigm. The basic idea is that a *primitive concept* is appropriate when no complete definition exists or when only part of a completely known definition is relevant. *Defined concepts* are appropriate when the complete definition is known and relevant, or when one wants the system to determine membership in a class.

The LP for modelling a primitive class is shown in Table 6.

Table 6. Logical Pattern for Modelling a Primitive Class

Slot	**Value**
General Information	
Name	Primitive Class
Identifier	LP-PC-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that elements, belonging to a certain group or set, must satisfy a set of conditions (that is, necessary conditions).
Examples	Suppose that someone wants to express that all 'workflows' are also 'plans'. 'Plans' are represented by the class 'Plan'.
Ontology Design Pattern	
Informal	
General	Instantiate the class `Class` and to assert on it the necessary conditions.
Examples	Create the class 'WorkFlow' and assert (among necessary conditions) that is 'subclassOf' the class 'Plan'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	
Formalization	
General	`Class(Class partial OntologyElement)`
Examples	`Class(WorkFlow partial Plan)`
Relationships	
Relations to other modelling components	Possible use of this LP in others *NeOn Ontology Modelling Components* (LPs, APs, and CPs).

Comments	
Comments	Important to mention [32]: a primitive class does not allow modelling the following: *if something fulfils these (necessary) conditions then it must be a member of this class*. That is, if class A is described as having necessary conditions, then we can say that *if an individual is a member of class A it must satisfy the conditions*. We cannot say that any individual that satisfies these conditions is a member of class A. The way to represent a *'partial' class* is to state that such class is exactly a subclass of the conjunction of a collection of superclasses and restrictions[8] (that they would be the necessary conditions).

4.2.2. Logical Pattern for Modelling a Defined Class

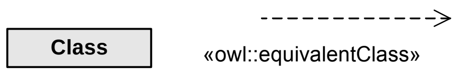
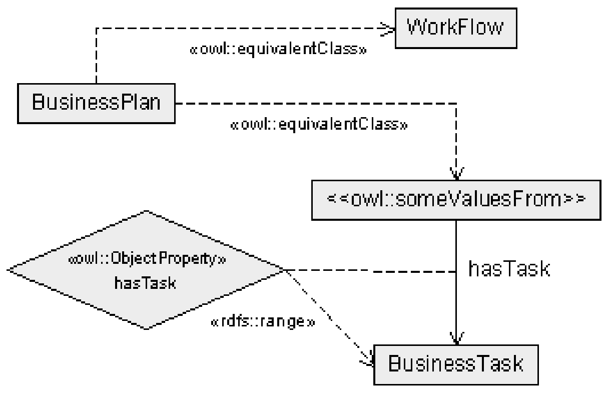
A class that has at least one set of necessary and sufficient conditions is known as a *Defined Class* [32]. Classes that have at least one set of necessary and sufficient conditions are also known as *'complete' classes*. They have a definition, and any individual that satisfies the definition belongs to the class.

Table 7 shows the LP for modelling a defined class.

Table 7. Logical Pattern for Modelling a Defined Class

Slot	Value
General Information	
Name	Defined Class
Identifier	LP-DC-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that elements which satisfy a given set of conditions belong to a certain group or set. In other words, to express that if any individual satisfies such conditions, then it must be a member of the group or set. In this case, necessary and sufficient conditions.
Examples	Suppose that someone wants to express that a 'workflow' which includes one or more 'business tasks' is a 'business plan'. Moreover, a 'business plan' is a 'workflow' that contains at least one 'business task'.
Ontology Design Pattern	
Informal	
General	Instantiate the class `Class` and to assert on it the necessary and sufficient conditions.

[8] http://www.w3.org/TR/2004/REC-owl-semantics-20040210/syntax.html

Examples	Create the classes 'BusinessPlan', 'WorkFlow', and 'BusinessTask' and assert (among necessary and sufficient conditions) that 'BusinessPlan' is 'subclassOf' 'WorkFlow' and has 'BusinessTask'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	 WorkFlow «owl::equivalentClass» BusinessPlan «owl::equivalentClass» <<owl::someValuesFrom>> «owl::ObjectProperty» hasTask hasTask «rdfs::range» BusinessTask
Formalization	
General	`Class(Class partial OntologyElement)`
Examples	`Class(BusinessPlan complete restriction(hasTask someValuesFrom(BusinessTask)) WorkFlow)`
Relationships	
Relations to other modelling components	Possible use of this LP in others *NeOn Ontology Modelling Components* (LPs, APs, and CPs).
Comments	
Comments	If class A is now defined using necessary and sufficient conditions, we can say that if an individual is a member of the class A, it must satisfy the conditions and we can now say that if any (random) individual satisfies these conditions, then it must be a member of class A. The conditions are not only necessary for membership of A but also sufficient to determine that something satisfying these conditions is a member of A [32]. The way to represent a '*complete*' *class* is to state that such class is exactly equivalent to the conjunction of a collection of superclasses and restrictions[9] (that they would be the necessary and sufficient conditions).

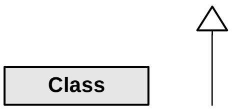
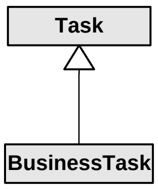
[9] http://www.w3.org/TR/2004/REC-owl-semantics-20040210/syntax.html

4.2.3. Logical Pattern for Modelling a SubClassOf Relation

OWL classes are interpreted as sets that contain individuals, and they may be organised into a superclass-subclass hierarchy, also known as a taxonomy. Subclasses specialise their superclasses. One of the key features of OWL-DL is that these superclass-subclass relationships (also called subsumption relationships) can be computed automatically by a reasoner [32].

The LP for modelling a 'subClassOf' relation is shown in Table 8.

Table 8. Logical Pattern for Modelling a SubClassOf Relation

Slot	Value
General Information	
Name	subClassOf Relation
Identifier	LP-SC-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that elements, belonging to a certain group or set, also belong to a more general set.
Examples	Suppose that someone wants to model that any 'business task' is a 'task'.
Ontology Design Pattern	
Informal	
General	Instantiate the class `Class` and the object property `subClassOf`.
Examples	Create the classes 'BusinessTask' and 'Task' and assert that 'BusinessTask' is 'subclassOf' 'Task'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	
Formalization	

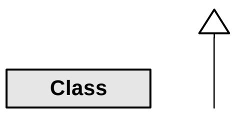
General	`Class(Class partial OntologyElement)`
	`Class(Property partial OntologyElement)`
	`Class(ObjectProperty partial Property)`
	`ObjectProperty(subClassOf domain(Class)`

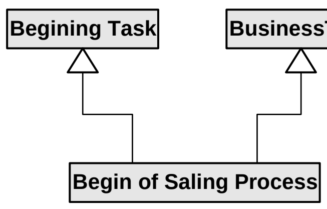
	`range(Class))`
Examples	`Class(BusinessTask partial Task)`
Relationships	
Relations to other modelling components	Possible use of this LP in APs and CPs. Use of this LP in LP-MI-01.

4.2.4. Logical Pattern for Modelling Multiple Inheritance between Classes

Multiple inheritance in an ontology occurs when a class can be subclass of several classes. The LP for modelling this situation is shown in Table 9.

Table 9. Logical Pattern for Modelling Multiple Inheritance

Slot	**Value**
General Information	
Name	Multiple Inheritance
Identifier	LP-MI-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that elements, belonging to a certain group or set, also belong to several more general sets.
Examples	Suppose that someone wants to model that any ‘begin of saling process’ is a ‘beginning task’ and also a ‘business task’.
Ontology Design Pattern	
Informal	
General	Instantiate the class `Class` and the object property `subClassOf`.
Examples	Create the classes ‘Begin of Saling Process’, ‘Beginning Task’, and ‘Business Task’, and assert that ‘Begin of Saling Proces’ is ‘subclassOf’ ‘Beginning Task’ and ’subclassOf’ ‘BusinessTask’.
Graphical	
(UML) Diagram for the General Solution	

(UML) Diagram for Examples	
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Property partial OntologyElement)`
	
	`Class(ObjectProperty partial Property)`
	
	`ObjectProperty(subClassOf domain(Class) range(Class))`

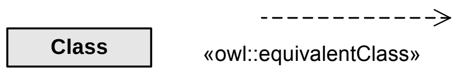
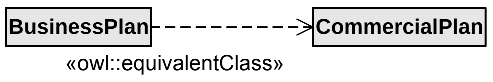
Examples	`Class(Begin of Saling Process partial Begining Task BusinessTask)`
Relationships	
Relations to other modelling components	Possible use of this LP in APs and CPs.

4.2.5. Logical Pattern for Modelling an Equivalence Relation between Classes

Table 10 shows the LP for modelling an equivalence relation between classes.

Table 10. Logical Pattern for Modelling an Equivalence Relation between Classes

Slot	Value
General Information	
Name	Equivalence Relation
Identifier	LP-EQ-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that two groups have precisely the same set of elements.
Examples	Suppose that someone wants to express that ‘business plans’ are the same as ‘commercial plans’.
Ontology Design Pattern	
Informal	
General	Instantiate the class `Class` and the object property `equivalentClass`.
Examples	Create the classes ‘BusinessPlan’ and ‘CommercialPlan’, and assert that ‘BusinessPlan’ is

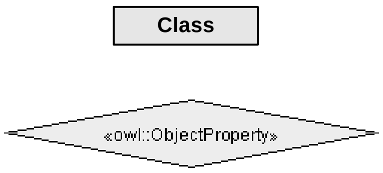
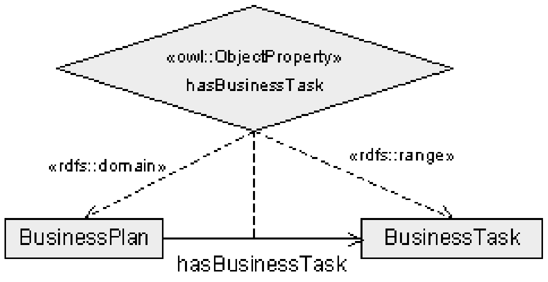
	'equivalent' to 'CommercialPlan'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	
Formalization	
General	`Class(Class partial OntologyElement)` `ObjectProperty(equivalentClass domain(Class) range(Class))`
Examples	`Class(BusinessPlan complete CommercialPlan)`
Relationships	
Relations to other modelling components	Possible use of this LP in others *NeOn Ontology Modelling Components* (LPs, APs, and CPs).

4.2.6. Logical Pattern for Modelling an Object Property

OWL properties represent relationships between two individuals [32]. Individuals, also known as instances, represent objects in the domain in which we are interested.
There are two main types of properties, object properties and datatype properties. In this section, we focus on object properties that link an individual to an individual. Table 11 shows the LP for modelling an object property.

Table 11. Logical Pattern for Modelling an Object Property

Slot	**Value**
General Information	
Name	Object Property
Identifier	LP-OP-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that elements, belonging to a certain group or set, have a relationship or link with elements, belonging to a certain group or set.
Examples	Suppose that someone wants to express that 'business plans' have 'business tasks'.
Ontology Design Pattern	
Informal	

General	Instantiate the classes `Class` and `ObjectProperty`.
Examples	Create the classes 'BusinessPlan' and 'BusinessTask', and assert that 'BusinessPlan' is link to 'BusinessTask' by means of an object property 'hasBusinessTask'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	 «owl::ObjectProperty» hasBusinessTask «rdfs::domain» «rdfs::range» BusinessPlan BusinessTask hasBusinessTask
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Property partial OntologyElement)`
	
	`Class(ObjectProperty partial Property)`

Examples	`ObjectProperty(hasBusinessTask domain(BusinessPlan) range(BusinessTask))`
Relationships	
Relations to other modelling components	Possible use of this LP in others *NeOn Ontology Modelling Components* (LPs, APs, and CPs). Use of this LP in LP-SP-01.

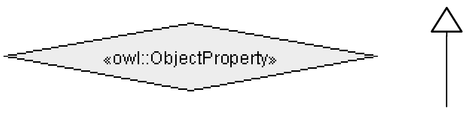
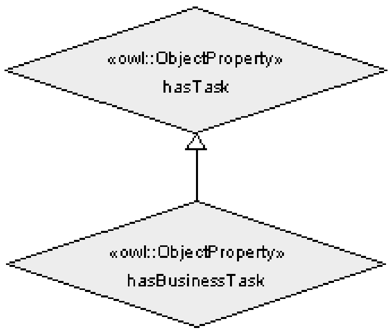
4.2.7. Logical Pattern for Modelling a SubPropertyOf Relation

OWL properties may have subproperties that specialise their superproperties (in the same way that subclasses specialise their superclasses) [32]. With this feature it is possible to form hierarchies of properties (object properties and datatype properties). However, it is not possible to create taxonomies of properties that mix object properties and datatype properties.

In this section, Table 12 shows the LP for modelling a 'subPropertyOf' relation between object properties.

Table 12. Logical Pattern for Modelling a SubPropertyOf Relation

Slot	**Value**
General Information	

Name	subPropertyOf Relation
Identifier	LP-SP-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that a relationship between two individuals specialises a more general relationship between such individuals.
Examples	Suppose that someone wants to model that the object property 'hasBusinessTask' specialices the more general object property 'hasTask'.
Ontology Design Pattern	
Informal	
General	Instantiate the class `ObjectProperty` and the object property `subPropertyOf`.
Examples	Create the object properties 'hasBusinessTask' and 'hasTask', and assert that 'hasBusinessTask' is 'subPropertyOf' 'hasTask'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	 «owl::ObjectProperty» hasTask «owl::ObjectProperty» hasBusinessTask
Formalization	
General	`Class(Property partial OntologyElement)` `Class(ObjectProperty partial Property)` `ObjectProperty(subPropertyOf domain(Property) range(Property))`
Examples	`ObjectProperty(hasBusinessTask super(hasTask))`

Relationships	
Relations to other modelling components	Possible use of this LP in APs and CPs.

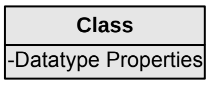
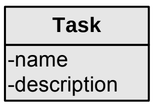
4.2.8. Logical Pattern for Modelling a Datatype Property

As mentioned before (in Section 4.2.6), OWL properties represent relationships between two individuals [32]. There are two main types of properties, object properties and datatype properties. In this section, we focus on datatype properties that link an individual to a XML Schema Datatype value[10] or an rdf literal[11].

Note that it is also possible to create subproperties of datatype properties, as in Section 4.2.7.

Table 13 shows the LP for modelling a datatype property.

Table 13. Logical Pattern for Modelling a Datatype Property

Slot	Value
General Information	
Name	Datatype Property
Identifier	LP-DP-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that elements, belonging to a certain group or set, have a relationship or link with elements, belonging to a certain group or set (literals, values, etc.).
Examples	Suppose that someone wants to express that 'tasks' have 'name' and 'description'.
Ontology Design Pattern	
Informal	
General	Instantiate the classes `Class` and `DatatypeProperty`.
Examples	Create the class 'Task', and assert that 'Task' has 'name' and 'description'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	

[10] http://www.w3.org/TR/xmlschema-2/

[11] http://www.w3.org/TR/rdf-primer/

Formalization	
General	`Class(Class partial OntologyElement)` `Class(Property partial OntologyElement)` `Class(DatatypProperty partial Property)`
Examples	`DatatypeProperty(name domain(Task) range(http://www.w3.org/2001/XMLSchema#string))` `DatatypeProperty(description domain(Task) range(http://www.w3.org/2001/XMLSchema#string))`
Relationships	
Relations to other modelling components	Possible use of this LP in the following LPs: LP-PC-01 and LP-DC-01. Possible use of this LP in APs and CPs.

4.2.9. Logical Pattern for Modelling an Existential Restriction

Existential restrictions [32], also known as 'someValuesFrom' restrictions or 'some' restrictions, are denoted using ∃. These restrictions describe the set of individuals that have at least one specific kind of relationship to individuals that are members of a specific class (Figure 6).

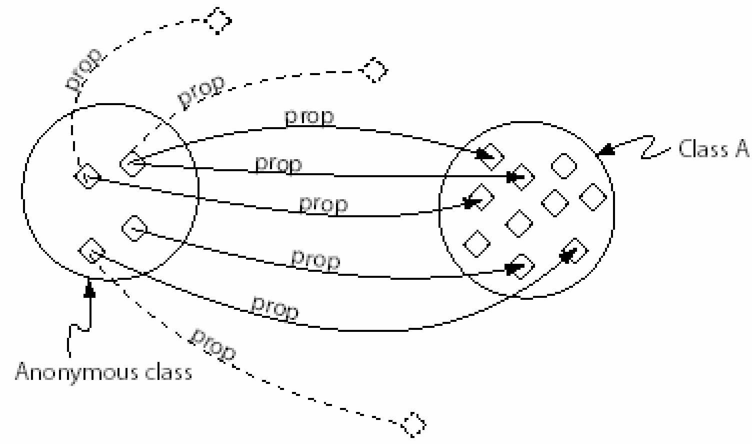
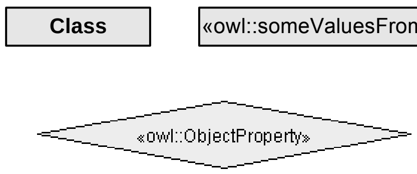
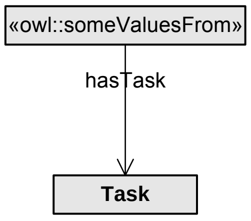


Figure 6. An Existential Restriction (∃ prop ClassA) [32]

The LP for modelling an existential restriction is shown in Table 14.

Table 14. Logical Pattern for Modelling an Existential Restriction

Slot	**Value**
General Information	
Name	Existential Restriction
Identifier	LP-ER-01
Type of Component	Logical Pattern (LP)
Use Case	

General	Express the set of elements that take place in at least one relationship with an other element, belonging to a certain group or set.
Examples	Suppose that someone wants to express the set of individuals that have at least one relationship with 'tasks'. In other words, someone wants to express things that 'have tasks'.
Ontology Design Pattern	
Informal	
General	Instantiate the classes `Class` , `ObjectProperty`, and `ExistentialRestriction`.
Examples	Create the class 'Task' and the relationship 'hasTask' with 'Task', and assert an existential restriction.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Property partial OntologyElement)`
	
	`Class(ObjectProperty partial Property)`
	
	`Class(ExistentialRestriction partial QualifiedNumberRestriction)`

Examples	`restriction(hasTask someValuesFrom(Task))`
Relationships	
Relations to other modelling components	Possible use of this LP in the following LPs: LP-PC-01, LP-DC-01, LP-NR-01, and LP-NR-02. Possible use of this LP in APs and CPs.
Comments	
Comments	The fact that we are using an existential restriction to describe the group of individuals having at least one relationship R with an individual that is a member of a class C does not mean that these individuals only have a relationship R with an individual

	that is a member of the class C (there could be other R relationships that just haven't been explicity specified).

4.2.10. Logical Pattern for Modelling a Universal Restriction

Universal restrictions are also known as 'allValuesFrom' restrictions, or 'all' restrictions since they constrain the filler for a given property to a specific class [32]. Universal restrictions (denoted by ∀) describe the set of individuals that, for a given property, only have relationships to other individuals that are members of a specific class (Figure 7). A feature of universal restrictions is that, for the given property, the set of individuals that the restriction describes will also contain the individuals that do not have any relationship along this property to any other individuals.

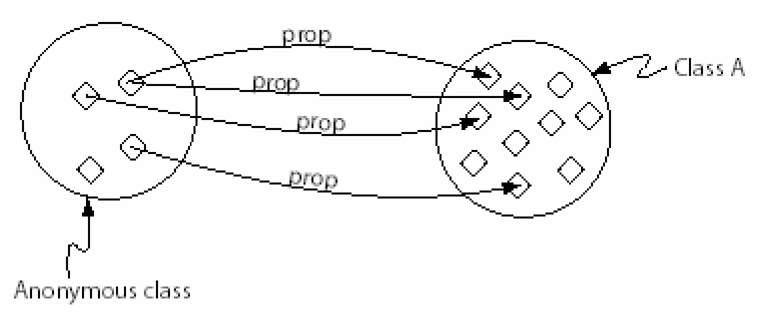
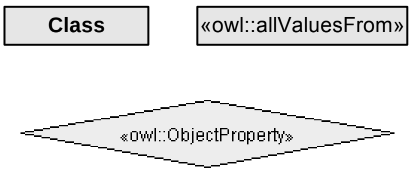
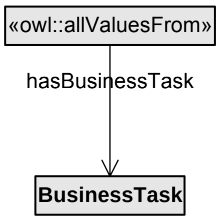


Figure 7. Universal Restriction (∀ prop ClassA) [32]

The LP for modelling a universal restriction is shown in Table 15.

Table 15. Logical Pattern for Modelling a Universal Restriction

Slot	Value
General Information	
Name	Universal Restriction
Identifier	LP-UR-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express the set of elements that only has relationships to elements belonging to a certaing group or set.
Examples	Suppose that someone wants to express the set of individuals that only have 'hasBusinessTask' relationships to 'business task'.
Ontology Design Pattern	
Informal	
General	Instantiate the classes `Class`, `ObjectProperty`, and `UniversalRestriction`.
Examples	Create the class 'BusinessTask' and the relationship 'hasBusinessTask', and assert an universal restriction.
Graphical	

(UML) Diagram for the General Solution	
(UML) Diagram for Examples	
Formalization	

General	`Class(Class partial OntologyElement)`
	`Class(Property partial OntologyElement)`
	`Class(ObjectProperty partial Property)`
	`Class(UniversalRestriction partial Restriction)`

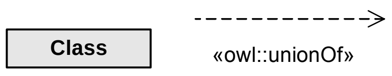
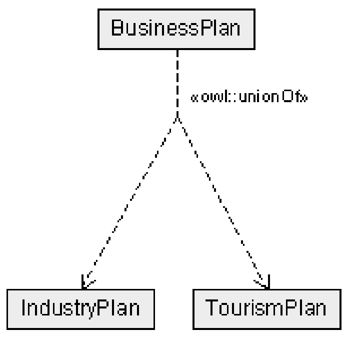
Examples	`restriction(hasBusinessTask allValuesFrom(BusinessTask))`
Relationships	
Relations to other modelling components	Possible use of this LP in the following LPs: LP-PC-01, LP-DC-01, LP-NR-01, and LP-NR-02. Possible use of this LP in APs and CPs.
Comments	
Comments	An important point to note is that universal restrictions do not 'guarentee' the existence of a relationship for a given property. They merely state that if such a relationship for the given property exists, then it must be with an individual that is a member of a specified class.

4.2.11. Logical Pattern for Modelling a UnionOf Relation

The LP for modelling a 'unionOf' relation is shown in Table 16.

Table 16. Logical Pattern for Modelling a UnionOf Relation

Slot	**Value**
General Information	
Name	unionOf Relation
Identifier	LP-UO-01
Type of Component	Logical Pattern (LP)
Use Case	

General	Express that one group includes the union of the elements from one or more groups.
Examples	Suppose that someone wants to express that the set of individuals of ‘business plan’ is the union of the individuals of ‘industry plan’ and of the individuals of ‘tourism plan’.
Ontology Design Pattern	
Informal	
General	Instantiate the classes `Class` and `Union`.
Examples	Create the classes ‘BusinessPlan’, ‘IndustryPlan’, and ‘TourismPlan’, and assert that ‘BusinessPlan’ is the ‘unionOf’ ‘IndustryPlan’ and ‘TourismPlan’.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	 BusinessPlan «owl::unionOf» IndustryPlan TourismPlan
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Property partial OntologyElement)`
	
	`Class(ObjectProperty partial Property)`
	
	`Class(Union partial BooleanCombination)`

Examples	`Class(BusinessPlan partial owl:Thing unionOf(IndustryPlan TourismPlan))`
Relationships	
Relations to other modelling components	Possible use of this LP in LPs, APs, and CPs. Use of this LP in LP-SV-02.

4.2.12. Logical Pattern for Modelling an Individual

Individuals represent objects in the domain of discourse. Individuals are also known as instances. Individuals can be referred to as being ‘instances of classes’ [32].

OWL does not use the Unique Name Assumption (UNA). This means that two different names could actually refer to the same individual [32].

The LP for modelling an individual is shown in Table 17.

Table 17. Logical Pattern for Modelling an Individual

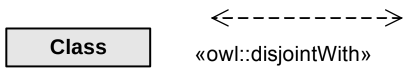
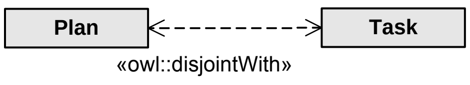
Slot	Value
General Information	
Name	Individual
Identifier	LP-In-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express elements, belonging to a certain group or set.
Examples	Suppose that someone wants to express a concrete 'plan' for achieving a particular goal.
Ontology Design Pattern	
Informal	
General	Instantiate the class `Individual`.
Examples	Create the individual 'planA' as an instance of the class 'Plan'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	
Formalization	
General	`Class(Individual partial OntologyElement Annotation)`
Examples	`Individual(planA type(Plan))`
Relationships	
Relations to other modelling components	Possible use of this LP in others *NeOn Ontology Modelling Components* (LPs, APs, and CPs).
Comments	
Comments	Remember that in OWL, it must be explicitly stated that individuals are the same as each other, or different to each other; otherwise they might be the same as each other, or they might be different to each other.

4.2.13. Logical Pattern for Modelling Disjoint Classes

OWL classes are assumed to 'overlap'. Therefore, we cannot assume that an individual is not a member of a particular class simply because it has not been asserted to be a member of that class. In order to ensure that an individual that has been asserted to be a member of a class cannot be a member of another class, we must make such classes disjoint. This means that it is not possible for an individual to be a member of a combination of such classes [32].

The LP for modelling disjoint classes is shown in Table 18.

Table 18. Logical Pattern for Modelling Disjoint Classes

Slot	Value
General Information	
Name	Disjoint Classes
Identifier	LP-Di-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that an element, belonging to a certain group or set, cannot belong to another group or set. In other words, express that two different sets are disjoint.
Examples	Suppose that someone wants to express that 'plans' are disjoint with 'tasks'.
Ontology Design Pattern	
Informal	
General	Instantiate the class `Class` and the object property `disjointWith`.
Examples	Create the classes 'Plan' and 'Task', and assert that 'Plan' is 'disjointWith' 'Task'.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Property partial OntologyElement)`
	
	`Class(ObjectProperty partial Property)`
	
	`ObjectProperty(disjointWith domain(Class) range(Class))`

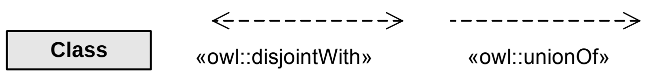
Examples	`DisjointClasses(Plan Task)`
Relationships	
Relations to other modelling components	Possible use of this LP in the following LPs: LP-NR-01,LP-NR-02 and LP-SV-02. Possible use of this LP in APs and CPs.

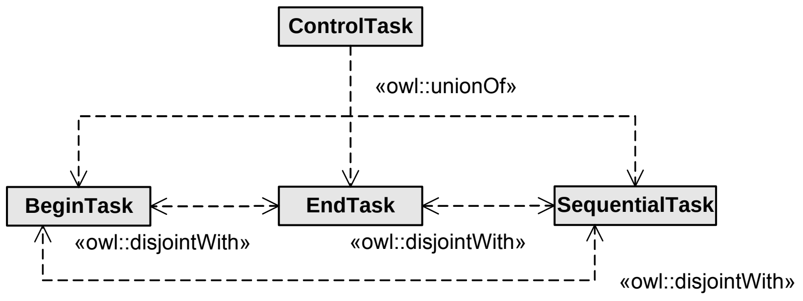
4.2.14. Logical Pattern for Modelling Exhaustive Classes

An exhaustive class is defined as the union of a set of mutually disjoint subclasses.

The LP for modelling exhaustive classes is shown in Table 19.

Table 19. Logical Pattern for Modelling Exhaustive Classes

Slot	Value
General Information	
Name	Exhaustive Classes
Identifier	LP-EC-01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that general group or set is the union several more specific groups or set, that are mutually disjoint.
Examples	Suppose that someone wants to express that a ‘control task’ can be only a ‘begin task’, an ‘end task’, or a ‘sequential task’.
Ontology Design Pattern	
Informal	
General	Instantiate the classes `Class` and `Union`, and the object property `disjointWith`.
Examples	Create the classes ‘ControlTask’, ‘BeginTask’, ‘EndTask’, and ‘SequentialTask’, and assert that ‘BeginTask’, ‘EndTask’, and ‘SequentialTask’ are ‘disjoint’, and ‘ControlTask’ is the ‘union’ of ‘BeginTask’, ‘EndTask’, and ‘SequentialTask’.
Graphical	
(UML) Diagram for the General Solution	

(UML) Diagram for Examples	 ControlTask «owl::unionOf» BeginTask EndTask SequentialTask «owl::disjointWith» «owl::disjointWith» «owl::disjointWith»
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Property partial OntologyElement)`
	
	`Class(ObjectProperty partial Property)`
	
	`ObjectProperty(disjointWith domain(Class) range(Class))`
	
	`Class(Union partial BooleanCombination)`

Examples	`DisjointClasses(EndTask SequentialTask BeginTask)`
	
	`Class(ControlTask partial owl:Thing unionOf(BeginTask EndTask SequentialTask))`

Relationships	
Relations to other modelling components	Possible use of this LP in the following LPs: LP-NR-01 and LP-NR-02. Possible use of this LP in APs and CPs.

4.2.15. Logical Pattern for Modelling N-ary Relation

In Semantic Web languages such as RDF and OWL, a property is a *binary* relation. This binary relation is used to link two individuals or an individual and a value. In some cases, however, the natural and convenient way to represent certain situations is to use relations and to link an individual to more than just one individual or value. These relations are called *n-ary relations* [46].

The W3C Semantic Web Best Practices and Deployment Working Group (SWBPD)[12] proposes two different solutions (namely, patterns) for solving the aforementioned problem: introducing a new class for the relation (Section 4.2.15.1) and using a list for arguments in the relation (Section 4.2.15.2). These two solutions are summarized in Section 2.1.1.4.

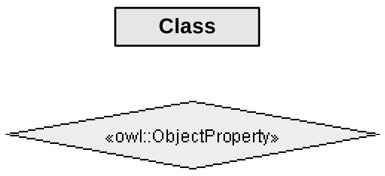
4.2.15.1. Introducing a new class for the relation

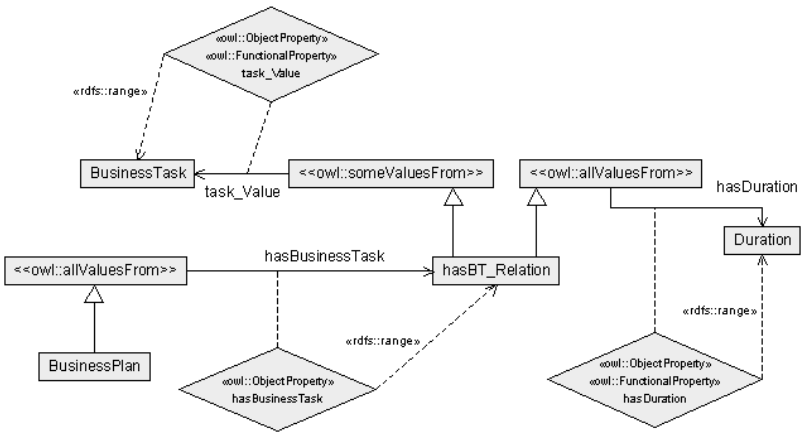
In this case, the solution proposed [46, 48] resides in creating a new class and *n* new object properties to represent an n-ary relation. An instance of the relation linking the *n* individuals is then an instance of this class.

[12] http://www.w3.org/2001/sw/BestPractices/

The LP for modelling n-ary relations by means of introducing a new class for the n-ary relation is shown in Table 20.

Table 20. Logical Pattern for Modelling N-ary Relation: Introducing a New Class for the Relation

Slot	**Value**
General Information	
Name	N-ary Relation: New Class
Identifier	LP-NR -01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that: ▪ A binary relationship really needs a further argument. ▪ Two binary relationships always go together and should be represented as one n-ary relation. ▪ A relationship is really amongst several things.
Examples	Suppose that someone wants to express that 'business plans' have 'business tasks' with a concrete 'duration'.
Ontology Design Pattern	
Informal	
General	Create a new class and *n* new object properties. Therefore, instantiate the classes `Class` and `ObjectProperty`.
Examples	Create the classes 'BusinessPlan', 'BusinessTask', 'hasBT_Relation', and 'Duration'. In the definition of the class 'BusinessPlan', specify an object property 'hasBusinessTask' with the range restriction going to 'hasBT_Relation' class. Define 'task_Value' and 'hasDuration' as functional object properties. In the definition of the class 'hasBT_Relation', specify two object properties 'task_Value' and 'hasDuration' with the range restriction going to the classes 'BusinessTask' and 'Duration', respectively.
Graphical	
(UML) Diagram for the General Solution	

(UML) Diagram for Examples	 «owl::ObjectProperty» «owl::FunctionalProperty» task_Value «rdfs::range» BusinessTask task_Value <<owl::someValuesFrom>> <<owl::allValuesFrom>> hasDuration Duration <<owl::allValuesFrom>> hasBusinessTask hasBT_Relation «rdfs::range» «rdfs::range» BusinessPlan «owl::ObjectProperty» hasBusinessTask «owl::ObjectProperty» «owl::FunctionalProperty» hasDuration
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Property partial OntologyElement)`
	
	`Class(ObjectProperty partial Property)`

Examples	`Class(BusinessPlan partial restriction(hasBusinessTask allValuesFrom(hBT_Relation)) owl:Thing)`
	
	`Class(hBT_Relation partial owl:Thing restriction(task_Value someValuesFrom(BusinessTask)) restriction(hasDuration allValuesFrom(Duration)))`
	
	`Class(BusinessTask partial owl:Thing)`
	
	`Class(Duration partial owl:Thing)`
	
	`ObjectProperty(task_Value Functional domain(owl:Thing))`
	
	`ObjectProperty(hasDuration Functional domain(owl:Thing))`

Relationships	
Relations to other modelling components	Possible use of this LP in APs and CPs.

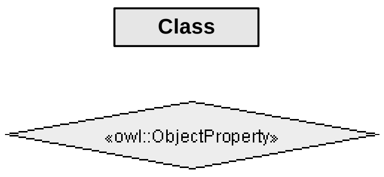
4.2.15.2. Using lists for arguments in the relation

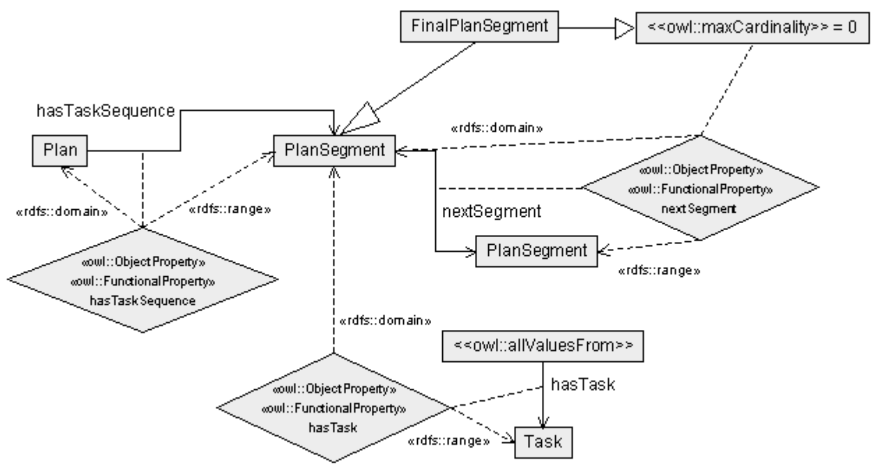
The solution proposed [46] is used when the order of the arguments of the n-ary relation is important. Thus, the idea is to represent several individuals participating in the relation as a collection or an ordered list.

In cases where all but one participant in a relation do not have a specific role and essentially form an ordered list, it is natural to connect these arguments into a sequence according to some relation and to relate the odd participant to this sequence (or the first element of the sequence) [46].

The LP for modelling n-ary relations by means of using lists for arguments in the relation is shown in Table 21.

Table 21. Logical Pattern for Modelling N-ary Relation: Using Lists for Arguments in the Relation

Slot	Value
General Information	
Name	N-ary Relation: Using Lists
Identifier	LP-NR -02
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that the nature of the relation is such that one or more of the arguments is fundamentally a sequence rather than a single individual.
Examples	Suppose that someone wants to express that a 'plan' is composed by 'tasks', with a concrete order. *Eg*.: P1 has T1, then T2, then T3, and finally T4.
Ontology Design Pattern	
Informal	
General	Create an *ordering relation* and two classes. Therefore, instantiate the classes `Class` and `ObjectProperty`.
Examples	Create the classes 'Plan', 'Task', 'PlanSegment', and 'FinalPlanSegment'. In the definition of the class 'PlanSegment', specify an object property 'hasTask' with the range going to the class 'Task'. In the definition of the class 'FinalPlanSegment', specify that such class is 'subclassOf' the class 'PlanSegment' and a maximum cardinality restriction equal to zero in the object property 'nextSegment'. Define 'hasTaskSequence' as a functional object property with domain the class 'Plan' and range the class 'PlanSegment'. Define 'nextSegment' as a functional object property with domain and range the class ' PlanSegment'. Define 'hasTask' as a functional object property with domain the class 'PlanSegment'.
Graphical	
(UML) Diagram for the General Solution	

(UML) Diagram for Examples	 FinalPlanSegment <<owl::maxCardinality>> = 0 hasTaskSequence Plan PlanSegment «rdfs::domain» «rdfs::range» «owl::ObjectProperty» «owl::FunctionalProperty» hasTaskSequence nextSegment nextSegment PlanSegment <<owl::allValuesFrom>> hasTask hasTask Task
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Property partial OntologyElement)`
	
	`Class(ObjectProperty partial Property)`

Examples	`Class(Plan partial owl:Thing)`
	
	`ObjectProperty(hasTaskSequence Functional domain(Plan) range(PlanSegment))`
	
	`Class(PlanSegment partial owl:Thing restriction(hasTask allValuesFrom(Task)))`
	
	`Class(FinalPlanSegment partial PlanSegment restriction(nextSegment maxCardinality(0)))`
	
	`ObjectProperty(nextSegment Functional domain(PlanSegment) range(PlanSegment))`
	
	`ObjectProperty(hasTask Functional domain(PlanSegment))`
	
	`Class(Task partial owl:Thing)`

Relationships	
Relations to other modelling components	Possible use of this LP in APs and CPs.

4.2.16. Logical Pattern for Modelling Specified Values

It is a common requirement for modelling ontologies to be able to represent notions such as a "small man", a "high ranking officer" or a "health person", that is, modelling various descriptive "features" (also known variously as "qualities", "attributes" or "modifiers"). In almost all such cases it is necessary to specify the constraints on the values for the "feature" (e.g. that size may be "small", "medium" or "large")[13].

[13] http://www.w3.org/TR/swbp-specified-values/

In OWL such descriptive features are modelled as properties whose range specifies the constraints on the values that the property can take on.

There are at least three different ways to represent such features and their specified values[14].

- As individuals whose enumeration makes up the parent class representing the feature.
- As disjoint classes which exhaustively partition the parent class representing the feature.
- As datatypes. Data types will be used more usually when there is a literal, numeric or derived data types rather than when there is an enumerated list of values.

The W3C Semantic Web Best Practices and Deployment Working Group (SWBPD)[15] proposes two different solutions (namely, patterns) for solving the aforementioned problem: representing values as sets of individuals (Section 4.2.16.1) and representing values as subclasses partitioning a "feature" (Section 4.2.16.2).

4.2.16.1. Values as sets of individuals

In this case, the solution proposed [48] resides in creating a class, representing the "feature" and in creating an enumeration of individuals (different from each other), representing the values for the "feature".

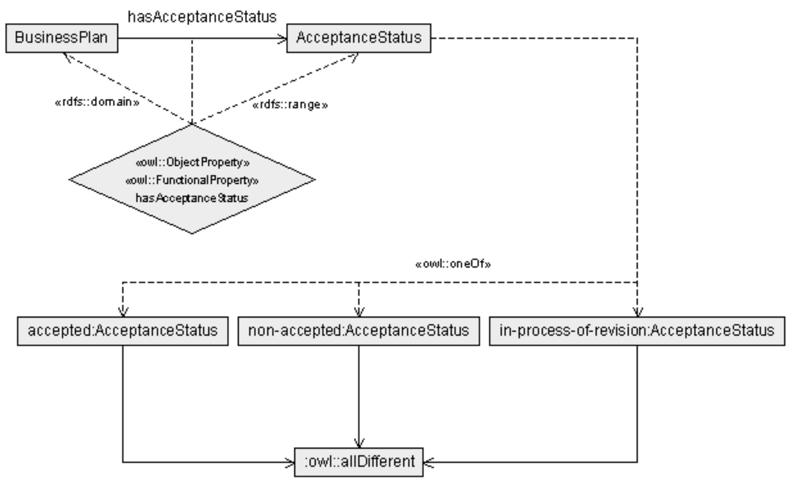
The LP for modelling specified values by means of representing values as sets of individuals is shown in Table 22.

Table 22. Logical Pattern for Modelling Specified Values: Values as Sets of Individuals

Slot	Value
General Information	
Name	Specified Values: Set of Individuals
Identifier	LP-SV -01
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that: a class has descriptive "features".
Examples	Suppose that someone wants to express that 'business plans' have a concrete status relating to their acceptance. That is, a 'business plan' can be 'accepted', 'non-accepted', and 'in process of revision'.
Ontology Design Pattern	
Informal	
General	Create a class, representing the "feature" and an enumeration of *n* individuals (which are different between them), representing the values for the "feature". Therefore, instantiate the classes `Class`, `Individual` and `AllDifferent`, and the object property `oneOf`.

[14] http://www.w3.org/TR/swbp-specified-values/

[15] http://www.w3.org/2001/sw/BestPractices/

Examples	Create the classes ‘BusinessPlan’ and ‘AcceptanceStatus’; and a functional object property ‘hasAcceptanceStatus’ between ‘BusinessPlan’ and ‘AcceptanceStatus’. Define the class ‘AcceptanceStatus’ as one of the following individuals: ‘accepted’, ‘non-accepted’, and ‘in-process-of-revision’. Define such individuals as ‘allDifferent’.
Graphical	
(UML) Diagram for the General Solution	**Class** **Individual** «owl::oneOf» «owl::allDifferent»
(UML) Diagram for Examples	 hasAcceptanceStatus BusinessPlan AcceptanceStatus «rdfs::domain» «rdfs::range» «owl::ObjectProperty» «owl::FunctionalProperty» hasAcceptanceStatus «owl::oneOf» accepted:AcceptanceStatus non-accepted:AcceptanceStatus in-process-of-revision:AcceptanceStatus :owl::allDifferent
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Individual partial OntologyElement Annotation)`
	
	`ObjectProperty(oneOf domain(EnumeratedDataRange) range(DataValue) range(Individual))`
	
	`Class(AllDifferent partial owl:Thing)`

Examples	`Class(BusinessPlan partial owl:Thing)`
	
	`Class(AcceptanceStatus partial owl:Thing oneOf(non-accepted in-proces-of-revision accepted))`
	
	`ObjectProperty(hasAcceptanceStatus Functional domain(BusinessPlan) range(AcceptanceStatus))`
	
	`Individual(accepted type(AcceptanceStatus))`
	
	`Individual(non-accepted type(AcceptanceStatus))`
	
	`Individual(in-process-of-revision type(AcceptanceStatus))`

	`AllDifferent (accepted non-accepted in-process-of-revision)`
Relationships	
Relations to other modelling components	Possible use of this LP in APs and CPs.
Comments	
Comments	OWL supports only equality or difference between individuals. It does not allow individuals to have partial overlaps. It is not possible, as it is for classes, to say that one individual is equivalent to the the union (disjunction) of two other individuals[16]. There is no way to represent alternative partitionings of the same feature space. Because individuals cannot overlap, if the class is defined as equivalent to enumeration of one list of distinct values, it cannot also be equivalent to a different list of distinct values.

4.2.16.2. Values as subclasses partitioning a 'feature'

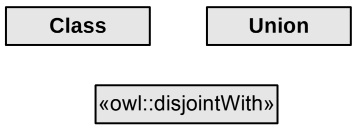
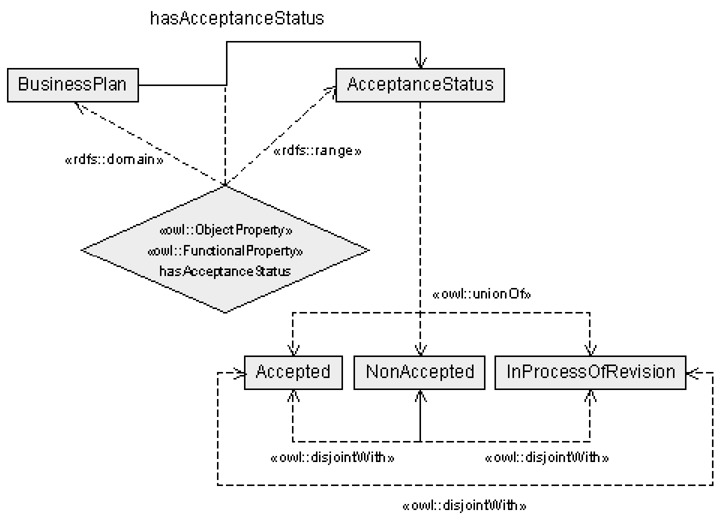
In this case, the solution proposed [48] resides in creating a class, representing the "feature", as the union of *n* classes (mutually disjoint), representing the values for the "feature".

The LP for modelling specified values by means of representing values as subclasses partitioning a "feature" is shown in Table 23.

Table 23. Logical Pattern for Modelling Specified Values: Values as Subclasses

Slot	Value
General Information	
Name	Specified Values: Subclasses
Identifier	LP-SV -02
Type of Component	Logical Pattern (LP)
Use Case	
General	Express that: a class has descriptive "features".
Examples	Suppose that someone wants to express that 'business plans' have a concrete status relating to their acceptance. That is, a 'business plan' can be 'accepted', 'non-accepted', and 'in process of revision'.
Ontology Design Pattern	
Informal	
General	Create a class, representing the "feature", as the union of *n* mutually disjoint classes, representing the values for the "feature".

[16] http://www.w3.org/TR/swbp-specified-values/

	Therefore, instantiate the classes `Class` and `Union`, and the object property `disjointWith`.
Examples	Create the classes ‘BusinessPlan’, ‘AcceptanceStatus’, ‘Accepted’, ‘NonAccepted’, and ‘InProcessOfRevision’; and a functional object property ‘hasAcceptanceStatus’ between ‘BusinessPlan’ and ‘AcceptanceStatus’. In the definion of ‘AcceptanceStatus’, specify that ‘AcceptanceStatus’ is the union of the classes ‘Accepted’, ‘NonAccepted’, and ‘InProcessOfRevision’. Define ‘Accepted’ disjoint with ‘NonAccepted’ and with ‘InProcessOfRevision’. Define ‘NonAccepted’ disjoint with ‘InProcessOfRevision’.
Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for Examples	 hasAcceptanceStatus BusinessPlan AcceptanceStatus «rdfs::domain» «rdfs::range» «owl::ObjectProperty» «owl::FunctionalProperty» hasAcceptanceStatus «owl::unionOf» Accepted NonAccepted InProcessOfRevision «owl::disjointWith» «owl::disjointWith» «owl::disjointWith»
Formalization	

General	`Class(Class partial OntologyElement)`
	
	`Class(Union partial BooleanCombination)`
	
	`ObjectProperty(disjointWith domain(Class) range(Class))`

Examples	`Class(BusinessPlan partial owl:Thing)`
	
	`Class(AcceptanceStatus partial owl:Thing unionOf(InProcessOfRevision NonAccepted Accepted))`
	
	`ObjectProperty(hasAcceptanceStatus Functional domain(BusinessPlan) range(AcceptanceStatus))`

	`Class(Accepted partial owl:Thing)` `DisjointClasses(Accepted NonAccepted InProcessOfRevision)`
	
	 `Class(NonAccepted partial owl:Thing)` `DisjointClasses(NonAccepted Accepted InProcessOfRevision)`
	
	 `Class(InProcessOfRevision partial owl:Thing)` `DisjointClasses(InProcessOfRevision Accepted NonAccepted)`

Relationships	
Relations to other modelling components	Possible use of this LP in APs and CPs.
Comments	
Comments	There can be several alternative partitionings of the same feature space[17]. The use of classes for values seems unintuitive to many people who come from the database and frame communities where value sets are usually enumerated lists of symbols.

[17] http://www.w3.org/TR/swbp-specified-values/

5. NeOn Ontology Modelling Components: Architectural Patterns

In this chapter we present a particular template for describing architectural patterns in the inventory; we also include a sample of architectural patterns using such template.

5.1. Template for Architectural Patterns

For ontology modelling components that we consider as architectural patterns (that is, structures expressed by means of a combination of LPs, that characterize the whole ontology), we propose the template shown in Table 24.

Table 24. Template for Architectural Patterns

Slot	Value
General Information	
Name	Name of the component
Identifier	An acronym composed of: component type + component + number
Type of Component	Architectural Pattern (AP)
Use Case	
General	Description in natural language of the general problem addressed by the modelling component.
Examples	Description in natural language of some examples for the general problem.
Ontology Design Pattern	
Informal	
General	Description in natural language of the general solution provided by the modelling component, refering to the NeOn OWL Ontology Metamodel defined in D1.1.1 [31].
Examples	Description in natural language of the instantiated solution for the examples.
Graphical	
(UML) Diagram for the General Solution	Graphical representation of the general solution provided, taking into account the UML Profile proposed in [9].
(UML) Diagram for Examples	Graphical representation of the solution provided, using examples and taking into account the UML

	Profile proposed in [9]. This could be optional.
Relationships	
Relations to other modelling components	Description of any relation to other modelling componens (use, specialize, etc.).
Comments	
Comments	Remarks for clarifying the use of the modelling component.

5.2. Inventory of Architectural Patterns

To date, we have identified the following *NeOn Ontology Modelling Components* considered as Architectural Patterns: tree structure, binary tree structure, graph structure, taxonomy structure, lightweight ontology and modular architecture.

However, in this document we only include a sample of the aforementioned patterns; we will include the rest of the partners and probably new ones in the subsequent deliverable to D5.1.1 (that is, D5.1.2) and in D2.5.1 (within WP2).

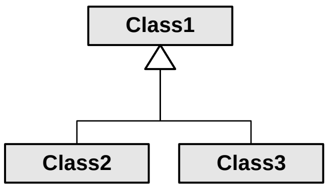
The current inventory of *NeOn Ontology Modelling Components* considered as Architectural Patterns includes as a sample the following ones: taxonomy, lightweight ontology and modular architecture.

5.2.1. Taxonomy

This *NeOn Ontology Modelling Component*, shown in Table 25, consists of organizing the ontology as a hierarchical structure of classes only related by subsumption relations.

Table 25. Architectural Pattern for Modelling a Taxonomy

Slot	Value
General Information	
Name	Taxonomy
Identifier	AP-TX-01
Type of Component	Architectural Pattern (AP)
Use Case	
General	Perform categorization/classification of information at different extents of granularity.
Examples	Suppose that someone wants to design an ontology of perfumes in order to categorize perfume product types based on their fragrance.
Ontology Design Pattern	
Informal	

General	Define a class for each element representing a type. Relate each class (*child*) to another (its *parent*) through the `subClassOf` relation. Instantiate the class `Class` and the object property `subClassOf`.
Examples	Create the classes 'Perfume' (that represents perfume types), and assert that 'Floral', 'Oriental', 'Woody', 'Fresh', and 'Fougère' are 'subclasses' of 'Perfume'.
Graphical	
(UML) Diagram for the General Solution	
Relationships	
Relations to other modelling components	Relations to the following LPs: LP-PC and LP-SC.
Comments	
Comments	A taxonomy can or cannot contain classes that have more than one parent class in different branch of the ontology (multiple inheritance). A taxonomy with multiple inheritance is useful when the classification has to be done based on different criteria. In this case an object can be instance of more than one class depending to which perspective it is observed from. If the requirement is to have one and only one (branch) type for each element, then multiple inheritance needs to be disallowed during the ontology design session.

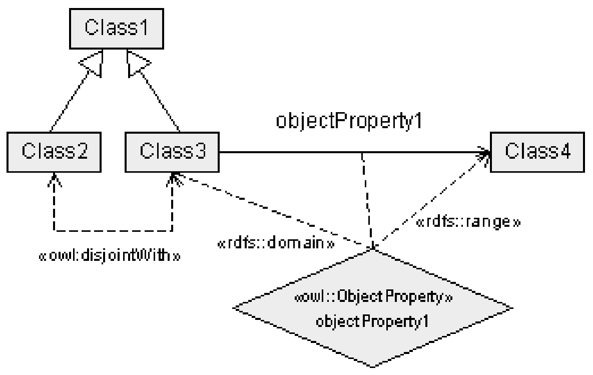
5.2.2. Lightweight ontology

This *NeOn Ontology Modelling Component*, shown in Table 26, adds the following features to the taxonomy structure (Section 5.2.1):

- A class can be related to other classes through the *disjointWith* relation.
- Object and datatype properties can be defined and used to relate classes.
- A specific domain and range can be associated with defined object and datatype properties.

Table 26. Architectural Pattern for Modelling a Lightweight Ontology

Slot	**Value**
General Information	
Name	Lightweight Ontology
Identifier	AP-LW-01

Type of Component	Architectural Pattern (AP)
Use Case	
General	Express disjointness between classes of a taxonomy and relationships between objects of various types.
Examples	Suppose that someone wants to design an ontology of perfumes in order to categorize perfume product types based on their fragrance, assigning them a price, and associating them with a producer. Fragance types are disjoint.
Ontology Design Pattern	
Informal	
General	Define a taxonomy of classes (see section 5.2.1) and use the `disjointWith` relation between classes in order to express class disjointness. Define `Property` elements (either `DatatypeProperty` or `ObjectProperty`) with specific `range` and `domain` if needed.
Examples	Create the Fragrance Taxonomy. Declare each class to be disjoint with the others (use `disjointWith`). Create the class 'Producer' that represents perfume producers. Define a `DatatypeProperty` for expressing prices e.g., *price*, which *domain* is the class *Fragrance,* and *range* is a `DataRange` [31] (e.g., `xsd:integer`). Define an `ObjectProperty` (e.g., `producedBy`) for expressing the relation between a perfume (i.e., whose `domain` is the class `Perfume`) and its producer (i.e., whose `range` is the class `Producer`). Instantiate the classes `Class`, `ObjectProperty`, and `DatatypeProperty`, and the object properties `subclassOf` and `disjointWith`.
Graphical	
(UML) Diagram for the General Solution	 Class1 Class2 Class3 Class4 objectProperty1 «owl:disjointWith» «rdfs::domain» «rdfs::range» «owl::ObjectProperty» objectProperty1
Relationships	
Relations to other modelling components	Relations to the following patterns: LP-OP, LP-DP, LP-DC, and AP-TX.

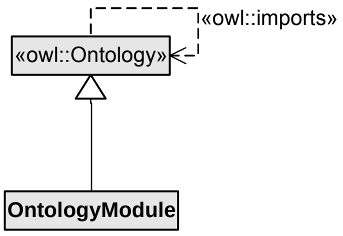
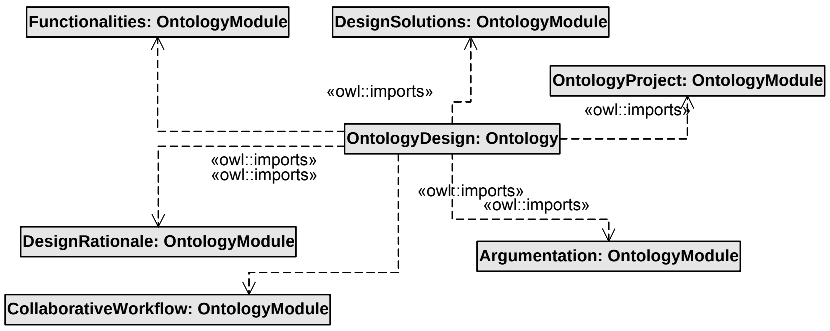
5.2.3. Modular architecture

This *NeOn Ontology Modelling Component*, shown in Table 27, consists in structuring an ontology as a configuration of components, each having its own identity based on some design criteria. Each module has the role of *part* in a *part-whole* relation to the ontology that has the *whole* role. Each module ontology has at least one element related to at least another module. Typically, when an ontology is committed to a huge domain of knowledge, which has to address many complex tasks, a good practice is to decompose the domain into smaller subdomains which address simpler tasks. Each subdomain can be then encoded in an ontology module, so as to provide the whole ontology with a modular architecture. Although this pattern complies to the NeOn metamodel for modular ontologies defined in [31], its scope is less general and its aim is different. This architectural pattern deals with design problems and solutions. Typically, the whole ontology imports the modules. Mapping links, external links, and other relations which do not imply a certain extent of dependency between the whole ontology and its parts, do not allow to build a 'Modular architecture' in the sense of this AP.

Table 27. Architectural Pattern for Modelling a Modular Architecture

Slot	Value
General Information	
Name	Modular Architecture
Identifier	AP-MD-01
Type of Component	Architectural Pattern (AP)
Use Case	
General	Design an ontology for a huge and complex domain of knowledge.
Examples	Suppose that someone wants to design an ontology which domain is 'ontology design'[18].
Ontology Design Pattern	
Informal	
General	Decompose the domain into a number of components i.e., modules, based on desing criteria. Assign a different namespace to each module, and possibly store each module in a different file. Design the cross-relations between modules.
Examples	Decompose the 'ontology design' domain in the following six subdomains: ontology project, collaborative workflow, argumentation, design rationales, design solutions, and functionalities. Develop an ontology (i.e., module) for each subdomain and assign its own namespace to it. We name the whole ontology `OntologyDesign`, while we name the modules as follows: `OntologyProject`, `CollaborativeWorkflow`, `Argumentation`, `DesignRationales`, `DesignSolutions`, and `Functionalities`. Make `OntologyDesign` import the six ontology modules. Each ontology module defines object properties, which represent cross-relations between the modules. These cross-relations represent the fact that an ontology project is developed by means of a set of functionalities that support a collaborative workflow, which allow designers to argument their design solutions through the expression of design rationales.

[18] This example is inspired by the C-ODO ontology [11] that has been developed in the context of WP2.

Graphical	
(UML) Diagram for the General Solution	
(UML) Diagram for the Example Solution	 Functionalities: OntologyModule DesignSolutions: OntologyModule OntologyProject: OntologyModule «owl::imports» «owl::imports» OntologyDesign: Ontology «owl::imports» «owl::imports» «owl::imports» «owl::imports» DesignRationale: OntologyModule Argumentation: OntologyModule CollaborativeWorkflow: OntologyModule
Relationships	
Relations to other modelling components	Possible relation to all the LPs, CPs, and APs.
Comments	
Comments	With regard to the NeOn Networked Ontology Model [31], an ontology designed with the Modular architecture AP is a 'Network of ontologies', while the vice versa is not always true. Moreover, a module of a Modular architecture is a 'Networked ontology'.

6. NeOn Ontology Modelling Components: Content Patterns

In this chapter we present a particular template for describing content patterns in the inventory; we also include a sample of content patterns using such template.

6.1. Template for Content Patterns

For ontology modelling components that we consider as content patterns (that is, instantiations of logical patterns or a composition of them), we propose the template shown in Table 28.

Table 28. Template for Content Patterns

Slot	**Value**
General Information	
Name	Name of the component
Identifier	An acronym composed of: component type + component + number
Type of Component	Content Pattern (CP)
Use Case	
General	Description in natural language of the general problem addressed by the modelling component.
Examples	Description in natural language of some examples for the general problem.
Ontology Design Pattern	
Informal	
General	Description in natural language of the general solution provided by the modelling component, refering to the NeOn OWL Ontology Metamodel defined in D1.1.1 [31]. In this case we focus on a generic domain.
Examples	Description in natural language of the solution provided using examples. In this case we focus on a specific domain. This could be optional.
Graphical	
(UML) Diagram for the General Solution	Graphical representation of the general solution provided, taking into account the UML Profile proposed in [9].

(UML) Diagram for Examples	Graphical representation of the solution provided, using examples and taking into account the UML Profile proposed in [9]. This could be optional.
Formalization	
General	Formalization of the pattern in terms of the most general classes and properties in OWL abstract syntax.
Examples	Formalization of specialized solution for the examples (using abstract syntax for OWL code). This could be optional.
Relationships	
Relations to other modelling components	Description of any relation to other modelling componens (use, specialize, etc.).
Comments	
Comments	Remarks for clarifying the use of the modelling component.

6.2. Inventory of Content Patterns

To date, we have identified the following *NeOn Ontology Modelling Components* considered as Content Patterns: participation pattern, description-situation pattern, role-task pattern, role-entity pattern, collection-entity pattern, collective-plan pattern, plan-execution pattern, simple part-whole relations pattern, and design-artifact pattern.

However, in this document, we only include a sample of the aforementioned patterns; we will include the rest of the patterns and probably new ones in the subsequent deliverable to D5.1.1 (that is, D5.1.2).

The current inventory of *NeOn Ontology Modelling Components* considered as Architectural Patterns includes as a sample the following: participation pattern, description-situation pattern, role-task pattern, plan-execution pattern, and simple part-whole relations pattern.

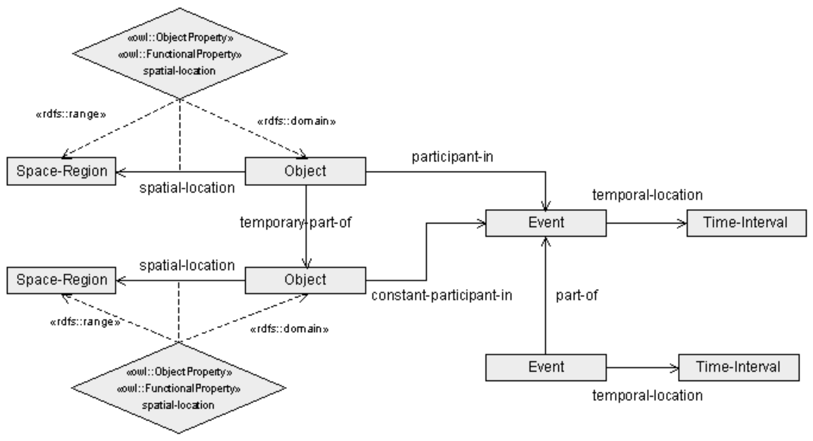
6.2.1. Participation Pattern

This *NeOn Ontology Modelling Component*, shown in Table 29, represents participation at spatio-temporal location [23], and it has been extracted from the DOLCE foundational ontology [41], developed within the WonderWeb Project (IST-2001-33052)[19].

Table 29. Content Pattern for Modelling Participation

Slot	**Value**
General Information	
Name	Participation

[19] http://wonderweb.semanticweb.org

Identifier	CP-PA-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express that objects take part in events.
Examples	Suppose that someones wants to represent people which take part in an international conference.
Ontology Design Pattern	
Informal	
General	The pattern consists in a 'participant-in' relation between *objects* and *events*, and assumes a time indexing for it. Time indexing is provided by the temporal location of the event at a *time interval*, while the respective spatial location at a *space region* is provided by the participating object. The pattern should instantiate the classes `Class` and `ObjectProperty`.
Graphical	
(UML) Diagram for the General Solution	 «owl::ObjectProperty» «owl::FunctionalProperty» spatial-location «rdfs::range» «rdfs::domain» Space-Region Object spatial-location participant-in temporal-location temporary-part-of Event Time-Interval spatial-location Space-Region Object constant-participant-in part-of «rdfs::range» «rdfs::domain» «owl::ObjectProperty» «owl::FunctionalProperty» spatial-location Event Time-Interval temporal-location *Note*: Diagram based on slides from the EKAW 2006 Tutorial about 'Ontology Design Patterns) [20].
Formalization	

General	`Class(Space-Region partial owl:Thing)`
	
	`ObjectProperty(spatial-location Functional domain(Object) range(Space-Region))`
	
	`Class(Object partial owl:Thing)`
	
	`ObjectProperty(temporary-part-of domain(Object) range(Object))`
	
	`ObjectProperty(participant-in domain(Object)`

[20] http://www.cs.vu.nl/~guus/public/ekaw-tutorial/content-patterns.pdf

	range(Event))
	
	ObjectProperty(constant-participant-in domain(Object) range(Event))
	
	Class(Event partial owl:Thing)
	
	ObjectProperty(part-of domain(Event) range(Event))
	
	ObjectProperty(temporal-location domain(Event) range(Time-Interval))
	
	Class(Time-Interval partial owl:Thing)

Relationships	
Relations to other modelling components	Relations to the following LPs: LP-DC / LP-PC and LP-OP.

6.2.2. Description-Situation Pattern

This *NeOn Ontology Modelling Component*, shown in Table 30, supports a first-order manipulation of *knowledge* (or *descriptive*) *objects* (such as plans, diagnoses, norms, institutions, etc. – i.e., *theories*) and *situations* (such as cases, facts, settings, etc. – i.e, *models* or *states of affairs*), and also allows a characterisation of the elements of descriptions and situations [25].

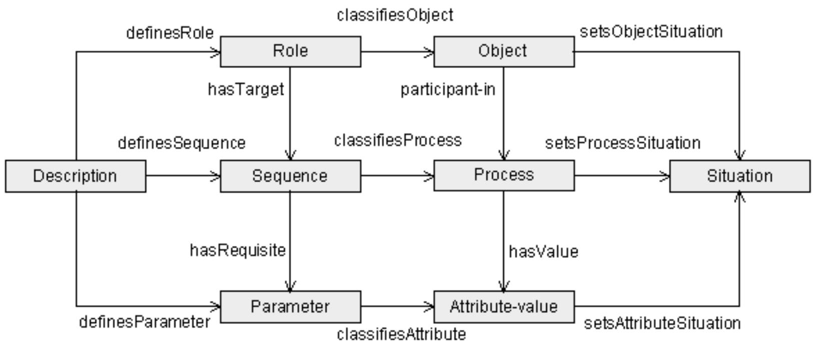
This pattern is based on an extension of DOLCE, called D&S (Descriptions and Situations) [26], partly developed within the Metokis Project (IST-2002-2.3.1.7)[21]. D&S provides a vocabulary and an axiomatization to type-reified classes and relations ("concepts" and "descriptions"), and to token-reified tuples ("situations").

A satisfied relation holds between situations and descriptions, implying that at least some components in a description must classify at least some entity in the situation setting [5].

Table 30. Content Pattern for Modelling Descriptions and Situations

Slot	Value
General Information	
Name	Description-Situation
Identifier	CP-DS-01
Type of Component	Content Pattern (CP)
Use Case	
General	Represent the (possible, actual, obliged, desired, etc.) correspondence between situations and descriptions.
Examples	Suppose that someone wants to represent a description of a workflow, and the relations to a concrete situation for such workflow, that is the concrete work being done.
Ontology Design Pattern	
Informal	

[21] http://metokis.salzburgresearch.at

General	The pattern consists of the classes 'Description', 'Role', 'Sequence', 'Parameter', 'Object', 'Process', 'Attribute-value', and 'Situation'. And the following relations: 'definesRole' between 'Description' and 'Role'; 'definesSequence' between 'Description' and 'Sequence'; 'definesParameter' between 'Description' and 'Parameter'; 'hasTarget' between 'Role' and 'Sequence'; 'hasRequisite' between 'Sequence' and 'Parameter'; 'classifiesObject' between 'Role' and 'Object'; 'classifiesProcess' between 'Sequence' and 'Process'; 'classifiesAttribute' between 'Parameter' and 'Attribute-value'; 'participant-in' between 'Object' and 'Process'; 'hasValue' between 'Process' and 'Attribute-value'; 'setsObjectSituation' between 'Object' and 'Situation'; 'setsProcessSituation' between 'Process' and 'Situation'; 'setsAttributeSituaton' between 'Attribute-value' and 'Situation'. The pattern should instantiate the classes `Class` and `ObjectProperty`.
Graphical	
(UML) Diagram for the General Solution	 classifiesObject definesRole Role Object setsObjectSituation hasTarget participant-in definesSequence classifiesProcess setsProcessSituation Description Sequence Process Situation hasRequisite hasValue Parameter Attribute-value definesParameter classifiesAttribute setsAttributeSituation *Note*: Diagram based on slides from the EKAW 2006 Tutorial about 'Ontology Design Patterns) [22].
Formalization	

General	`Class(Description partial owl:Thing)`
	
	`Class(Role partial owl:Thing)`
	
	`Class(Sequence partial owl:Thing)`
	
	`Class(Paramenter partial owl:Thing)`
	
	`Class(Object partial owl:Thing)`
	
	`Class(Process partial owl:Thing)`
	
	`Class(Attribute-value partial owl:Thing)`
	
	`Class(Situation partial owl:Thing)`
	
	`ObjectProperty(definesRole domain(Description) range(Role))`

[22] http://www.cs.vu.nl/~guus/public/ekaw-tutorial/content-patterns.pdf

	`ObjectProperty(definesSequence domain(Description) range(Sequence))`
	
	 `ObjectProperty(definesParameter domain(Description) range(Parameter))`
	
	 `ObjectProperty(hasTarget domain(Role) range(Sequence))`
	
	 `ObjectProperty(hasRequisite domain(Sequence) range(Parameter))`
	
	 `ObjectProperty(classifiesObject domain(Role) range(Object))`
	
	 `ObjectProperty(classifiesProcess domain(Sequence) range(Process))`
	
	 `ObjectProperty(classifiesAttribute domain(Parameter) range(Attribute-value))`
	
	 `ObjectProperty(participant-in domain(Object) range(Process))`
	
	 `ObjectProperty(hasValue domain(Process) range(Attribute-value))`
	
	 `ObjectProperty(setsObjectSituation domain(Object) range(Situation))`
	
	 `ObjectProperty(setsProcessSituation domain(Process) range(Situation))`
	
	 `ObjectProperty(setsAttributeSituation domain(Attribute-value) range(Situation))`

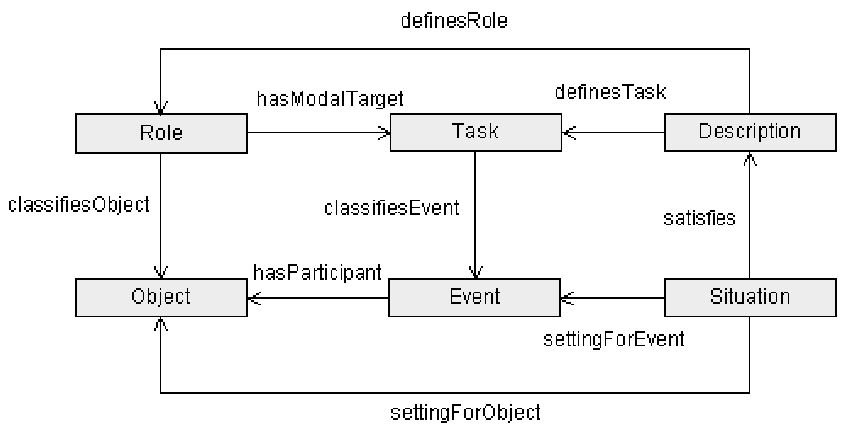
Relationships	
Relations to other modelling components	Relations to the following LPs: LP-DC / LP-PC and LP-OP.

6.2.3. Role-Task Pattern

This *NeOn Ontology Modelling Component*, shown in Table 31, is based on Description-Situation pattern (Section 6.2.2) and allows the expression, in OWL-DL, of the temporary *roles* that objects can play, and of the *tasks* that events/actions allow to execute [23]. The reified relation specifying roles and tasks is a *description*, the reified tuple that satisfies the relation for certain individual objects and events is called *situation*. Roles can have assigned tasks as *modal targets*.

Table 31. Content Pattern for Modelling Roles and Tasks

Slot	Value
General Information	
Name	Role-Task
Identifier	CP-RT-01

Type of Component	Content Pattern (CP)
Use Case	
General	Express that objects can play different roles, and that objects can execute different task (depending on the played role).
Examples	Suppose that someone wants to express the assigments of tasks to role-players in a project plan.
Ontology Design Pattern	
Informal	
General	The pattern consist of the classes 'Role', 'Object', 'Task', 'Event', 'Description', and 'Situation'. And the following relations: 'classifiesObject' between 'Role' and ''Object'; 'hasModalTarget' between 'Role' and 'Task'; 'definesTask' between 'Description' and 'Task'; 'definesRole' between 'Description' and 'Role'; 'classifiesEvent' between 'Task' and 'Event'; 'hasParticipant' between 'Event' and 'Object'; 'settingForEvent' between 'Situation' and 'Event'; 'settingForObject' between 'Situation' and 'Object'; and 'satisfies' between 'Situation' and 'Description'. The pattern should instantiate the classes `Class` and `ObjectProperty`.
Graphical	
(UML) Diagram for the General Solution	 definesRole hasModalTarget definesTask Role Task Description classifiesObject classifiesEvent satisfies hasParticipant Object Event Situation settingForEvent settingForObject *Note*: Diagram based on [23].
Formalization	

General	`Class(Role partial owl:Thing)`
	`Class(Object partial owl:Thing)`
	`Class(Task partial owl:Thing)`
	`Class(Event partial owl:Thing)`
	`Class(Description partial owl:Thing)`
	`Class(Situation partial owl:Thing)`

	`ObjectProperty(classifiesObject domain(Role) range(Object))` `ObjectProperty(classifiesEvent domain(Task) range(Event))` `ObjectProperty(satisfies domain(Situation) range(Description))` `ObjectProperty(hasModalTarget domain(Role) range(Task))` `ObjectProperty(definesTask domain(Description) range(Task))` `ObjectProperty(definesRole domain(Description) range(Role))` `ObjectProperty(hasParticipant domain(Event) range(Object))` `ObjectProperty(settingForEvent domain(Situation) range(Event))` `ObjectProperty(settingForObject domain(Situation) range(Object))`
Relationships	
Relations to other modelling components	Relations to the following LPs: LP-DC / LP-PC and LP-OP.

6.2.4. Plan-Execution pattern

This *NeOn Ontology Modelling Component*, shown in Table 32, is based on the DOLCE D&S Plan Ontology (DDPO) [5] and provides the logical and ontological foundation of so-called task taxonomies for knowledge content. In DDPO, *plans* are formal *descriptions* that represent "action schemas", i.e. sequences of actions that lead from a given situation to a new one, and the related entities involved.

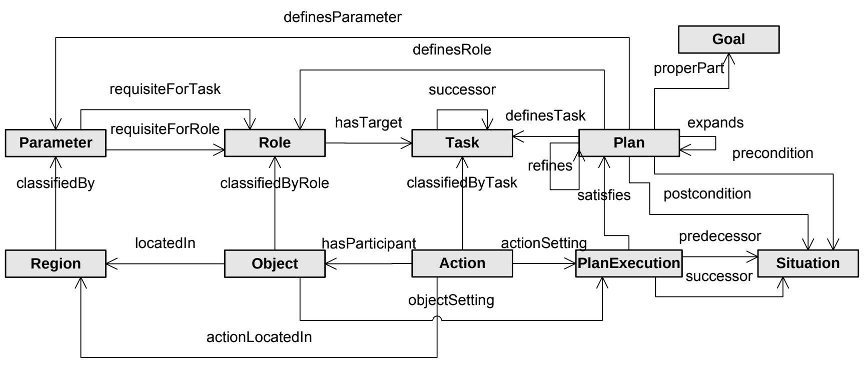
According to DDPO, typical components of a plan are *tasks* that provide instructions to execute (classify) *actions*. A plan defines or uses at least one task and one *role* (which must classify an *object*, and at least one role must classify an agent) and has at least one *goal* as a proper part (that goal is usually desired by the creator or beneficiary of a plan).

In addition, plans may include *parameters* that are classified by *attributes* (called "*regions*") of actions or *objects*. Parameters are related to roles or tasks by a requisite for relation, expressing the kind of requisites that the entities which are classified by the said roles or tasks should have in a given plan.

Notice that plans, as *descriptions*, are different from *plan executions*, which are *situations* (Section 6.2.2). A plan execution is a situation that (proactively) satisfies a plan description.

Plans may also have situations as pre- or post-conditions. A situation is a pre-condition for a plan if it should preliminarily satisfy some description before executions of that plan are performed. A situation is a postcondtion of a plan if it should satisfy some description after plan executions of that plan are performed. It often holds that the goal situation is a postcondition of plans, but this is not mandatory. Of course, every plan execution has predecessor and successor situations, but only some of them are pre- or post-conditions for the plan that the plan execution is supposed to satisfy.

Table 32. Content Pattern for Modelling Plans and Executions

Slot	**Value**
General Information	
Name	Plan-Execution
Identifier	CP-PE-01
Type of Component	Content Pattern (CP)
Use Case	
General	Represent the correspondence between a description of a plan (tasks, goals, roles, etc.) and an execution of such plan (actions, objects, etc.).
Examples	Suppose that someone wants to represent a software development plan (which describes the plan for developing and improving a public website during different periods) and the concrete executions of such plan in the past 10 years.
Ontology Design Pattern	
Informal	
General	The pattern partially consist of the classes ‘Plan’, ‘Goal’, ‘Situation’, ‘PlanExecution’, and ‘Task’. And the following relations: ‘definesTask’ between ‘Plan’ and ‘Task’; ‘properPart’ between ‘Plan’ and ‘Goal’; ‘satisfies’ between ‘PlanExecution’ and ‘Plan’; and ‘precondition’ between ‘Plan’ and ‘Situation’. The rest of classes and relations are shown in the next slot (in a graphical way). The pattern should instantiate the classes `Class` and `ObjectProperty`.
Graphical	
(UML) Diagram for the General Solution	 definesParameter definesRole Goal properPart requisiteForTask successor definesTask hasTarget expands requisiteForRole Parameter Role Task Plan precondition refines classifiedBy classifiedByRole classifiedByTask satisfies postcondition predecessor locatedIn hasParticipant actionSetting Region Object Action PlanExecution Situation successor objectSetting actionLocatedIn *Note*: Diagram based on slides from the EKAW 2006 Tutorial about ‘Ontology Design Patterns) [23] and on [23].
Formalization	
General	`Class(Plan partial owl:Thing)`

[23] http://www.cs.vu.nl/~guus/public/ekaw-tutorial/content-patterns.pdf

	`Class(Goal partial owl:Thing)`
	
	`Class(Situation partial owl:Thing)`
	
	`ObjectProperty(definesTask domain(Plan) range(Task))`
	
	`ObjectProperty(properPart domain(Plan) range(Goal))`
	
	`ObjectProperty(satisfies domain(PlanExecution) range(Plan))`

Relationships	
Relations to other modelling components	Relations to the following LPs: LP-DC / LP-PC and LP-OP.
Comments	
Comments	The intended use of this pattern is to specify plans at an abstract level and independently from existing resources.

6.2.5. Simple Part-Whole Relation Pattern

Representing part-whole relations is a very common issue for those developing ontologies for the Semantic Web. Part-whole relations are one of the basic structuring primitives of the universe, and many applications require representations of them (catalogues of parts, fault diagnosis, anatomy, geography, etc.). OWL does not provide any built-in primitives for part-whole relations (as it does for the subclass relation), but contains sufficient expressive power to capture most, but not all, common cases[24].

An important and common requirement for the basic relation from a part to its whole that is transitive, i.e. if A is part of B, and B is part of C, then A is part of C. OWL provides a general construct for declaring properties to be transitive. If we define a property, say `partOf`, to be transitive, then any reasoner conformant with OWL will draw the conclusions that both A and B are parts of C.

Sometimes it is useful to use the property hierarchy to define a subproperty of `partOf` that is not transitive and links each subpart just to the next level.

OWL supports inverse relations, so we can define an inverse of `partOf`, say `hasPart`. For any two individuals `I1` and `I2`, if "`I1 partOf I2`" then "`I2 hasPart I1`". However, care must be taken when using inverses in restrictions on classes. To say that "All As are parts of some B" does not imply that "All Bs have some As as parts", i.e. the restriction

(Class A partial restriction(partOf someValuesFrom(B))

does not imply

(Class B partial restriction(partOf someValuesFrom(A))

Therefore, if we want to say both that "all As are parts of some B" and "all Bs have part some A", we have to assert each statement separately. Such pairs of statements are sometimes called "reciprocals".

Unfortunately, all current OWL reasoners scale very badly for large part-whole hierarchies connected by *both* `hasPart` and `partOf`. Therefore, if reasoners are to be used, it is usually necessary to choose to use either `partOf` or `hasPart` but not both. Often it is preferable to use `partOf` because the most common queries and class definitions are for the parts of things,.

[24] http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/index.html

The W3C Semantic Web Best Practices and Deployment Working Group (SWBPD)[25] analysed how to treat with simple part-whole relations in OWL ontologies. In this section we present two different patterns, based on the work done by SWBPD, for using part-whole relations: modelling a part-whole relation (Section 6.2.5.1) and representing a part-whole class hierarchy (Section 6.2.5.2).

6.2.5.1. Modelling a part-whole relation

In this case, the solution proposed[26] resides in creating an importable ontology, which represent the family of part-whole relations.

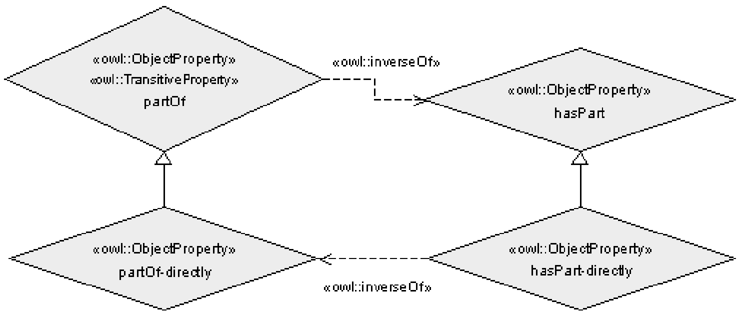
The LP for modelling a part-whole relation is shown in Table 33

Table 33. Content Pattern for Modelling a Part-Whole Relation

Slot	**Value**
General Information	
Name	Part-Whole Relation
Identifier	CP-PW-01
Type of Component	Content Pattern (CP)
Use Case	
General	Express 'part-whole' relations in general.
Examples	Suppose that someone wants to express that a 'Research Plan' is part of a 'Research Project'.
Ontology Design Pattern	
Informal	
General	Create the object properties 'part of', 'part of directly', 'has part', and 'has part directly'. Define the object properties 'part of' and 'has part' as inverses. Define the object properties 'part of directly' and 'has part directly' as inverses. Define the object property 'part of' transitive. Therefore, instantiate the class `ObjectProperty` and the object properties `inverseOf` and `transitive`.
Graphical	

[25] http://www.w3.org/2001/sw/BestPractices/

[26] http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/index.html

(UML) Diagram for the General Solution	 «owl::ObjectProperty» «owl::TransitiveProperty» partOf «owl::inverseOf» «owl::ObjectProperty» hasPart «owl::ObjectProperty» partOf-directly «owl::ObjectProperty» hasPart-directly «owl::inverseOf»
Formalization	

General	`ObjectProperty(partOf inverseOf(hasPart) Transitive domain(owl:Thing))`
	
	`ObjectProperty(hasPart inverseOf(partOf) domain(owl:Thing))`
	
	`ObjectProperty(hasPart-directly super(hasPart)inverseOf(partOf-directly))`
	
	`ObjectProperty(partOf-directly super(partOf) inverseOf(hasPart-directly))`

Relationships	
Relations to other modelling components	Relations to the following LPs: LP-OP and LP-SP.

6.2.5.2. Representing a part-whole class hierarchy

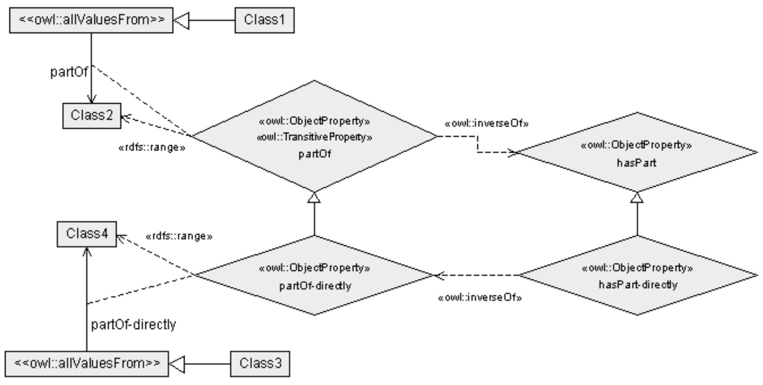
In this case, the solution proposed[27] resides in using the 'partOf' relation (Section 6.2.5.1) to describe composition at the level of a whole class of objects (e.g., all cars have engines).

The LP for modelling a part-whole class hierarchy is shown in Table 34.

Table 34. Content Pattern for Modelling a Part-Whole Class Hierarchy

Slot	**Value**
General Information	
Name	Part-Whole Relation: For class hierarchies
Identifier	CP-PW-02
Type of Component	Content Pattern (CP)
Use Case	
General	Express that several components form part of an aggregate whole.
Examples	Suppose that someone wants to express that a 'Research Plan' is part of a 'Research Project'. And a 'Research Plan' is composed by a 'Theoretical Plan' and an 'Experimental Plan'.

[27] http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/index.html

Ontology Design Pattern	
Informal	
General	Use the 'part-whole' relations (defined in CP-PW-01). Choose between using 'partOf' or 'hasPart'. Express part-whole relations amongst classes using 'someValuesFrom' with 'partOf' and 'partOf-directly'.
Graphical	
(UML) Diagram for the General Solution	 <<owl::allValuesFrom>> Class1 partOf Class2 «rdfs::range» «owl::ObjectProperty» «owl::TransitiveProperty» partOf «owl::inverseOf» «owl::ObjectProperty» hasPart Class4 «rdfs::range» «owl::ObjectProperty» partOf-directly «owl::inverseOf» «owl::ObjectProperty» hasPart-directly partOf-directly <<owl::allValuesFrom>> Class3
Formalization	

General	`ObjectProperty(partOf inverseOf(hasPart) Transitive domain(owl:Thing))`
	
	`ObjectProperty(hasPart inverseOf(partOf) domain(owl:Thing))`
	
	`ObjectProperty(hasPart-directly super(hasPart)inverseOf(partOf-directly))`
	
	`ObjectProperty(partOf-directly super(partOf) inverseOf(hasPart-directly))`
	
	`Class(Class1 partial owl:Thing restriction(partOf someValuesFrom(Class2)))`
	
	`Class(Class3 partial owl:Thing restriction(partOf-directly someValuesFrom(Class4)))`

Relationships	
Relations to other modelling components	Relations to the following components: LP-DC / LP-PC and CP-PW-01.

7. Conclusions and Further Work

In this deliverable we present first a state of the art on previous works related to modelling different components (such as Ontologies, Mappings, Rules, and Modules, selected according to the NeOn metamodel identified in D1.1.1 [31]). Such state of the art includes processes followed for modelling ontologies, mappings, rules and work related to create modules. We also present some previous works on creating designs collaboratively because this collaboration issue is an important aspect of the NeOn project.

Taking into account the analysis of the state of the art, in this deliverable we present **the first version of the inventory of *NeOn Ontology Modelling Components* (OWL-based design patterns)**, that is, modelling components that permit teams to model OWL ontologies (a particular class, a particular object property, a concrete pattern which solves a particular problem, best practices, etc.), basing on the elements of the OWL ontology metamodel [31].

Such inventory has been created by reusing, integrating and adapting as much as possible existing and well-accepted best practices (W3C and Knowledge Web) and existing ontology design patterns.

The *OWL-based design patterns* presented in this deliverable are organized into three different types:

Logical ontology design patterns (LPs): untyped structures expressed only with logical vocabulary, and which solve modelling problems. Examples of LPs are the definition of a class, the definition of a class as subclass of another class, and the definition of n-ary relations.

Architectural ontology design patterns (APs): untyped structures expressed through a combination of LPs. This type of patterns is related to 'how the ontology looks like'. An example of AP is the taxonomy.

Content design patterns (CPs): typed structures expressed with a domain specific (non logical) vocabulary. A CP represents and solves a domain modelling problem and affects the (limited) part of the ontology dealing with such a problem. An example of CP is the description of a plan.

This work on modelling components has a strong relation with the collaborative design components represented in the C-ODO metamodel (NeOn ontology for ontology design)[28] [11] produced in WP2. With reference to C-ODO, *NeOn Ontology Modelling Components* correspond to those instances of the class *formal-expression*, which are also instances of *logical-design-pattern*, or *architectural-design-pattern*, or *content-design-pattern*. Moreover, the instances of *logical-design-pattern* are related to the instances of *ontology-element* that in the context of NeOn are the elements of the OWL ontology metamodel in the NeOn networked ontology metamodel [31].

The inventory of *NeOn Ontology Modelling Components* presented in this document will be extended in D5.1.2 (the subsequent deliverable to D5.1.1), providing collections of the rest of modelling components ("*NeOn Rule Modelling Components*", "*NeOn Mapping Modelling Components*", etc), using specific templates and taking into account the NeOn networked ontology metamodel (presented in D1.1.1 [31]).

After analizing the state of the art, we realized that no complete guidelines for modelling the different components of a network of ontologies (ontologies, rules, mappings, and modules) exist. The works in the current literature (related to ontologies) are mainly focused on resolving concrete problems during the task of modelling ontologies. Such work provides guidelines (best practices, patterns, etc.) for modelling particular problems that normally appear when modelling OWL

[28] http://ww.loa-cnr.it/ontologies/OD/OntologyDesign.owl

ontologies. So far, no work published provides workflows for telling people how to begin and continue with the modelling process of networks of ontologies.

Indeed, the main question that people (ontology engineers, domain experts, etc.) make when developing ontologies is the following: “how do we start the ontology modelling process?”. They want to know if they should begin by modelling classes and taxonomies, and then if they should include (i.e.) existential restrictions; or if they should begin by modelling relations between classes; etc. The real problem here is that there are no guidelines that explain people how to model ontologies (what kind of workflow they should to follow). However, it has been experimentally proved (by several groups) that there is no a unique way of starting to model ontologies, since this depends on task, expertise, available resources, etc.

For these reasons, our idea is to propose a set of general steps and a very simple and general workflow to help people to model ontologies (a kind of process-oriented guidelines). In D5.4.1. we will present such guidelines, using the modelling components of the first version of the inventory presented in D5.1.1. The first idea for the guidelines is shown in Figure 8.

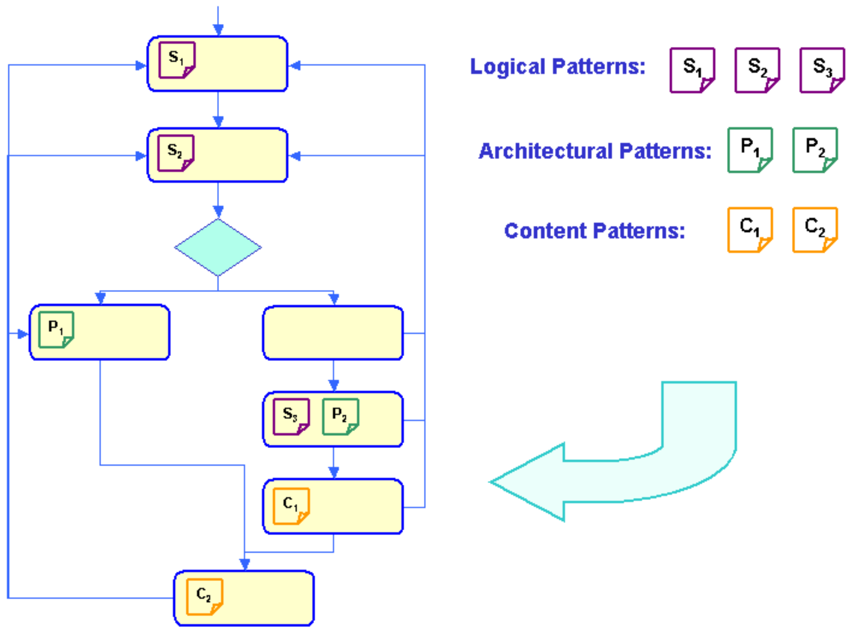


Figure 8. First Idea for the Process-Oriented Guidelines

Finally, we would like to mention that the *NeOn Ontology Modelling Components* presented in this deliverable are in practice ontology design patterns. However, we have already identified other type of patterns, that we name *operative ontology pattern (OP)*, and which are process-oriented patterns. For example, consider the case of a manager that wants to know how many employees are currently on sick leave in the enterprise he/she is working for, in order to take some strategic decision. This can be discovered by exploiting the LP ‘subclassOf relation’, in order to create, as a subclass of employees, a class of employees with certain properties, which captures the fact that an employee is sick. All employee individuals who satisfy those properties at that time can be inferred to be also individuals of the ‘sick employee’ class. This knowledge can be obtained by launching a classifier on the knowledge base. This kind of *“to infer implicit knowledge*” pattern is an OP. OPs are very useful and help users to exploit an ontology in an operative fashion, and they will be treated in two NeOn deliverables: D5.4.1 and D2.5.1 (within WP2).

The main distinction between NeOn ontology modelling components and OPs is that the former deals with the “design” of the ontology, while the latter deals with the “use” of the ontology (they include process-oriented solutions).

References

1. M.S. Ackerman. *The Intellectual Challenge of CSCW: The Gap Between Social Requirements and Technical Feasibility*. In John Carroll (ed.), HCI in the New Millennium, Addison-Wesley, 2001.
2. C. Alexander. *The timeless way of building*. Oxford University Press, New York (1979).
3. G. Antoniou, C. V. Damasio, B. Grosof, I. Horrocks, M. Kifer, J. Maluszynski, and P. F. Patel-Schneider. *Combining rules and ontologies: a survey*. Research report IST506779/Linköping/I3-D3/D/PU/b3, Linköping University, 2005. REWERSE Deliverable.
4. F. Baader, I. Horrocks, and U. Sattler. *Description logics as ontology languages for the semantic web*. In D. Hutter and W. Stephan, editors, Festschrift in honor of Jorg Siekmann, Lecture Notes in Artificial Intelligence. Springer, 2003.
5. W. Behrendt. METOKIS Deliverable D27. Final Report and Software Showcases. Available at: http://metokis.salzburgresearch.at/files/deliverables/metokis_d27_final_public_report_final.pdf. October 2005.
6. A. Borgida, R. J. Brachman. *Conceptual Modelling with Description Logics*. Description Logic Handbook 2003. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, Peter F. Patel-Schneider (Eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press 2003, ISBN 0-521-78176-0.
7. P. Bouquet, L. Serafini, and S. Zanobini. *Peer-to-Peer Semantic Coordination*. Journal of Web Semantics, 2(1), 2005.
8. S. Brockmans, P. Haase, P. Hitzler, and R. Studer. *A metamodel and uml profile for rule-extended owl dl ontologies*. In The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings, volume 4011 of Lecture Notes in Computer Science. Springer, 2006. Y. Sure and J. Domingue, editors. Pages 303–316.
9. S. Brockmans and P. Haase. *A Metamodel and UML Profile for Networked Ontologies. A Complete Reference*. Technical report, Universität Karlsruhe, April 2006. Available at: http://www.aifb.uni-karlsruhe.de/WBS/sbr/publications/ontology-metamodelling.pdf.
10. F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd, Chichester, UK (1996).
11. C. Catenacci, A. Gangemi, J. Lehmann, M. Nissim, V. Presutti, G. Steve, N. Guarino, C. Masolo, H. Lewen, K. Dellschaft, and M. Sabou. *NeOn Deliverable D2.1.1 Design rationales for collaborative development of networked ontologies - State of the art and the Collaborative Ontology Design Ontology*. February 2007. Available at: http://www.neon-project.org/.
12. P. Cimiano, S. Handschuh, and S. Staab. *Towards the Self-Annotating Web*. In Proc. of the 13th International Conference on World Wide Web (WWW'04), 2004.
13. A. Das, W. Wand, and D.L. McGuinness. *Industrial Strength Ontology Management*. In Proceedings of the International Semantic Web Working Symposium, 2001.
14. H.H. Do, E. Rahm. *COMA – a system for flexible combination of schema matching approaches*. In P. A. Bernstein *et al.*, editors, Proceedings of 28th International Conference on Very Large Data Bases (VLDB-2002), pages 610–621, Hong Kong, China, August 2002. Morgan Kaufmann Publishers.
15. A. Doan, P. Domingos, and A. Halevy. *Learning to match the schemas of data sources: A multistrategy approach*. VLDB Journal, 50:279–301, 2003.
16. M. Ehrig. *Ontology Alignment - Bridging the Semantic Gap*. PhD thesis, University of Karlsruhe, Universität Karlsruhe (TH), Institut AIFB, D-76128 Karlsruhe, 2006. Studer/Egle/Euzenat.
17. M. Ehrig, J. de Bruijn, D. Manov, and F. Martin-Recuerda. *State-of-the-art survey on ontology merging and aligning v1*. SEKT Deliverable 4.2.1, DERI Innsbruck, JUL 2004.

18. J. Euzenat, H. Stuckenschmidt, and M. Yatskevich. *Introduction to the Ontology Alignment Evaluation* 2005. In Proc. of the Integrating Ontologie WS, 2005.

19. J. Euzenat (Coordinator). *State of the art on ontology alignment*. Knowledge Web deliverable, D2.2.3, 2004.

20. M. Fernández-López, A. Gómez-Pérez. 2004. *Searching for a Time Ontology for Semantic Web Applications*. In Formal Ontology in Information Systems, A.C. Varzi and L. Vieu (Eds.), IOS Press.

21. M. Fernández-López, A. Gómez-Pérez, and N. Juristo. *METHONTOLOGY: From ontological art towards ontological engineering*. In Proceedings of the Workshop on Ontological Engineering, AAAI Spring Symposium (AAAI'97). AAAI Press, Menlo Park, California, 1997.

22. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA (1995).

23. A. Gangemi. *Ontology Design Patterns for Semantic Web Content*. Musen *et al.* (eds.): Proceedings of the Fourth International Semantic Web Conference, Galway, Ireland, 2005. Springer.

24. A. Gangemi, C. Catenacci, M. Ciaramita, J. Lehmann. *Ontology evaluation and validation: an integrated formal model for the quality diagnostic task*. 2005. Deliverable for ONTODEV project. Available at: http://www.loacnr.it/Files/OntoEval4OntoDev_Final.pdf.

25. A. Gangemi, C. Catenacci, and M. Battaglia. *Inflammation Ontology Design Pattern: an Exercise in Building a Core Biomedical Ontology with Descriptions and Situations*. D.M. Pisanelli (ed.) Ontologies in Medicine, IOS Press, Amsterdam (2004).

26. A. Gangemi, S. Borgo, C. Catenacci, J. Lehmann. *METOKIS Deliverable D07. Task Taxonomies for Knowledge Content*. June 2004. Available at: http://metokis.salzburgresearch.at/files/deliverables/metokis_d07_task_taxonomies_final.pdf.

27. F. Giunchiglia, P. Shvaiko, and M. Yatskevich. *Semantic Schema Matching*. In Proceedings of CoopIS'05, volume 3760 of LNCS, pages 347 – 360, 2005.

28. A. Gómez-Pérez, M. Fernández-López, and O. Corcho. *Ontological Engineering*. Springer, London, U.K., 2003.

29. A. Gómez-Pérez, N. Juristo, C. Montes, and J. Pazos. *Ingeniería del Conocimiento: Diseño y Construcción de Sistemas Expertos*. 1997. Ceura, Madrid, Spain.

30. N. Guarino, C.A. Welty. *Ontological Analysis of Taxonomic Relationships*. Alberto H. F. Laender, Stephen W. Liddle, Veda C. Storey (Eds.): Conceptual Modelling - ER 2000, 19th International Conference on Conceptual Modelling, Salt Lake City, Utah, USA, October 9-12, 2000, Proceedings. Lecture Notes in Computer Science 1920 Springer 2000, ISBN 3-540-41072-4.

31. P. Haase, S. Rudolph, Y. Wang, S. Brockmans, R. Palma, and J. Euzenat, M. d'Aquin. *NeOn Deliverable D1.1.1 Networked Ontology Model*. November 2006. Available at: http://www.neon-project.org/.

32. M. Horridge, H. Knublauch, A. Rector, R. Stevens, C. Wroe. *A Practical Guide To Building OWL Ontologies Using The Protege-OWL Plugin and CO-ODE Tools Edition 1.0*. The University Of Manchester. August 2004. Available at: http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf.

33. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. Technical report, World Wide Web Consortium (W3C), May 2004. W3C Member Submission, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

34. I. Horrocks. *OWL: A Description Logic Based Ontology Language*. Peter van Beek (Ed.): Principles and Practice of Constraint Programming - CP 2005, 11th International Conference,

CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings. Lecture Notes in Computer Science 3709 Springer 2005, ISBN 3-540-29238-1.

35. Q. Ji, W. Liu, G. Qi, and D. A. Bell. Lcs: *A linguistic combination system for ontology matching*. In J. Lang, F. Lin, and J. Wang, editors, KSEM, volume 4092 of Lecture Notes in Computer Science, pages 176–189. Springer, 2006.
36. M. Kifer, G. Lausen, and J. Wu. *Logical foundations of object-oriented and frame-based languages*. Journal of the Association for Computing Machinery, 1995.
37. P. Kollock. *Design Principle for Online Communities*. In Proceedings of the Harvard Conference on Internet and Society, 1996.
38. V. Lopez, E. Motta, and V. Uren. *PowerAqua: Fishing the Semantic Web*. In Proc. of the Third European Semantic Web Conference, 2006.
39. Y. Lu. *Roadmap for Tool Support for Collaborative Ontology Engineering*. Master thesis, XiAn Transportation University, Department of Computer Science, 2003. Available at: http://www.cs.uvic.ca/~chisel/thesis/YilingLu.pdf.
40. C. Mancini, S.J. Buckingham Shum. *Modelling Discourse in Contested Domains: A Semiotic and Cognitive Framework*. International Journal of Human-Computer Studies. (2006, in press).
41. 14. C. Masolo, A. Gangemi, N. Guarino, A. Oltramari and L. Schneider. *WonderWeb Deliverable D18: The WonderWeb Library of Foundational Ontologies*. 2004.
42. B. Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Universität Karlsruhe (TH), Institut AIFB, D-76128 Karlsruhe, 2006.
43. B. Motik, U. Sattler, and R. Studer. *Query answering for owl-dl with rules*. In Proceedings of the 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan, NOV 2004.
44. N. Noy. *Representing Classes As Property Values on the Semantic Web*. W3C Working Group Note 5 April 2005. Available at: http://www.w3.org/TR/swbp-classes-as-values/.
45. N.F. Noy, D.L. McGuiness. *Ontology Development 101: A Guide to Creating Your First Ontology*. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.
46. N. Noy, A. Rector. *Defining N-ary Relations on the Semantic Web*. W3C Working Group Note. 2006. Available at: http://www.w3.org/TR/swbp-n-aryRelations/.
47. N.F. Noy, M. A. Musen. *The PROMPT suite: interactive tools for ontology merging and mapping*. International Journal of Human-Computer Studies, 59(6):983–1024, 2003.
48. J.Z. Pan, L. Lancieri, D. Maynard, F. Gandon, R. Cuel, and A. Leger. *Knowledge Web Deliverable D1.4.2.v2. Success Stories and Best Practices*. January 2007. Available at: http://www.csd.abdn.ac.uk/~jpan/pub/TR/D142v2-final.pdf.
49. S. Pinto, S. Staab, C. Tempich. *DILIGENT: Towards a Fine-Grained Methodology towards Distributed, Loosely-Controlled and Evolving Engineering of Ontologies*. ECAI 2004.
50. R. Porzel, M. Baudis. *The Tao of CHI: Towards Effective Human-Computer Interaction*. In Proceedings of HLT-NAACL 2004, pp.209-216, 2004.
51. A.L. Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, and Chris Wroe. *OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns*. Enrico Motta, Nigel Shadbolt, Arthur Stutt, Nicholas Gibbins (Eds.): Engineering Knowledge in the Age of the Semantic Web, 14th International Conference, EKAW 2004, Whittlebury Hall, UK, October 5-8, 2004, Proceedings. Lecture Notes in Computer Science 3257 Springer 2004, ISBN 3-540-23340-7.
52. A.L. Rector, Chris Wroe, Jeremy Rogers, and Angus Roberts. *Untangling taxonomies and relationships: personal and practical problems in loosely coupled development of large ontologies*. Proceedings of the First International Conference on Knowledge Capture (K-CAP 2001), October 21-23, 2001, Victoria, BC, Canada. ACM 2001, ISBN 1-58113-380-4.

53. M. Sabou, M. D'Aquin, and E. Motta. *Using the Semantic Web as Background Knowledge for Ontology Mapping*. In Proceedings of the International Workshop on Ontology Matching (OM2006), collocated with the 5th International Semantic Web Conference (ISWC'06). November 5, 2006. Athens, Georgia, USA. CEUR-WS Vol-225.

54. M. Sabou, V. Lopez, E. Motta, and V. Uren. *Ontology Selection: Ontology Evaluation on the Real Semantic Web*. In Proceedings of the Evaluation of Ontologies on the Web Workshop, held in conjunction with WWW'2006, 2006.

55. M. Sabou, V. Lopez, E. Motta. *Ontology Selection on the Real Semantic Web: How to Cover the Queens Birthday Dinner?* In Proceedings of the European Knowledge Acquisition Workshop (EKAW), Podebrady, Czech Republic (2006).

56. B. Sereno, S.B. Shum, and E. Motta. *Claimspotter: An Environment to support Sensemaking with Knowledge Triples*. Technical Report KMI-04-29. In Proceedings of the International Conference of Intelligent User Interfaces, IUI2005, San Diego, CA, USA.

57. S.B. Shum, E. Motta, and J. Domingue. *Augmenting Design Deliberation with Compendium: The Case of Collaborative Ontology Design*. Position paper at the Workshop on Facilitating Hypertext Collaborative Modelling in conjunction with ACM Hypertext Conference, Maryland, June 11-12, 2002.

58. M. Schmidt-Schauss. *Subsumption in KL-ONE is Undecidable*. In Proceedings of the First International Conference on the Principles of Knowledge Representation and Reasoning (KR-89), pages 421–431. Morgan Kaufmann, Los Altos, 1989.

59. S. Staab, H.P. Schnurr, R. Studer, and Y. Sure. *Knowledge Processes and Ontologies*. IEEE Intelligent Systems 16(1):26–34. (2001).

60. H. Stuckenschmidt, F. van Harmelen, L. Serafini, P. Bouquet, and F. Giunchiglia. *Using C-OWL for the Alignment and Merging of Medical Ontologies*. In Proc. of the First Int. WS. on Formal Biomedical K. R. (KRMed), 2004.

61. A. Tate, S. Buckingham Shum, J. Dalton, C. Mancini, and A. Selvin. *Co-OPR: Design and Evaluation of Collaborative Sensemaking and Planning Tools for Personnel Recovery*. Techreport ID: KMI-06-07 2006.

62. J. D. Ullman. *Principles of Database & Knowledge-Base Systems Volume 1: Classical Database Systems*. W.H. Freeman & Company, 1988.

63. M. Uschold, M. Gruniger. (1996). *Ontologies: Principles, Methods and Applications*. Knowledge Engineering Review 11(2).

64. W. van Hage, S. Katrenko, and G. Schreiber. *A Method to Combine Linguistic Ontology-Mapping Techniques*. In Proc. of ISWC, 2005.

65. H. Wang, A. Rector, N. Drummond, M. Horridge, J. Seidenberg, N. Noy, M. Musen, T. Redmond, D. Rubin, S. Tu, and T. Tudorache. *Frames and OWL Side by Side*. 9th International Protégé Conference - July 23-26, 2006 - Stanford, California.

66. *Accepted Papers of the W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, DC, USA*, 2005. Available at: http://www.w3.org/2004/12/rules-ws/accepted.

67. *W3C Rule Interchange Format Working Group Charter*. Available at: http://www.w3.org/2005/rules/wg/charter, 2005.

Annex I. Related Terminology

In this annex important issues to be taken into account are defined.

- **Binary relationships** are modeled in DLs using roles, that are first class citizens, and attributes.
- **Cardinality Restrictions** state the minimum and maximum number of objects that can be related via a role (in DLs).
- **Concept Satisfiability**. Given an ontology O and a class A, verify whether there is a model of O in which the interpretation of A is a non-empty set.
- **Concept Subsumption**. Given an ontology O and two classes A, B, verify whether the interpretation of A is a subset of the interpretation of B in every model of O.
- **Concept**. *Classes of individuals* (people, institutions, etc) are normally modeled using primitive concepts in DLs. C*oncepts* or *classes* denote sets of individuals.
- **Closed World Assumption**. When absence of information is interpreted as negative information. In other words, when the information is always understood to be complete.
- **Defined Concept**. A *defined concept* is like an “if and only if” statement in logic. In other words, this kind of definition includes necessary and sufficient conditions for membership in the class.
- **Disjointness Axiom**. Two concepts are *disjoint* if an individual (or object) cannot be an instance of both concepts.
- **Domain Restrictions** state the kinds of objects that can be related via a role (in DLs).
- **Existential Restriction** specifies the existence of a (i.e. at least one) relationship along a given property to an individual that is a member of a specific class.
- **Functional Property**. If a property is *functional*, for a given individual, there can be at most one individual that is related to the individual via the property.
- **Individuals** represent objects in the domain that we are interested in. A synonym is *instance*.
- **Instance Checking**. Given an ontology O, an individual *a* and a class *A*, verify whether *a* is an instance of *A* in every model of O.
- **Open World Assumption**. When the absence of information only indicates lack of knowledge. In other words, when the information is in general viewed as being incomplete.
- **Primitive Concept**. A *primitive concept* includes only necessary (but not sufficient) conditions for membership. In contrast to defined concepts, primitive concepts support deductions in only one direction (like an “if” statement instead of an “if and only if” statement). A synonym of primitive concept is *atomic concept*.
- **Role** denotes binary relationships between individuals.
- **Universal Restriction** constrains the relationships along a given property to individuals that are members of a specific class.