NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

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

D1.1.3 NeOn Formalisms for Modularization: Syntax, Semantics, Algebra

Deliverable Co-ordinator: **Mathieu d'Aquin (OU)**

Deliverable Co-ordinating Institution: **Knowledge Media Institute, the Open University**

Other Authors: **Peter Haase, Sebastian Rudolph, Jérôme Euzenat, Antoine Zimmermann, Martin Dzbor, Marta Iglesias, Yves Jacques, Caterina Caracciolo, Carlos Buil Aranda, Jose Manuel Gomez**

The goal of this document is to come up with a formalism for ontology modularization, including syntaxes and the fundamental properties of a semantics of such a formalism. Furthermore we introduce operators to create, combine and manipulate ontology modules and give formal definitions for these operators based on the semantics of ontology modules. The definition of the NeOn formalism for modularization and of the operators to manipulate ontology modules are guided by a number of use cases and examples, from NeOn cases studies and other work packages.

Document Identifier:	NEON/2008/D1.1.3/V1.1	Date due:	February 29, 2008
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	February 29, 2008
Project start date	March 1, 2006	Version:	V1.1
Project duration:	4 years	State:	Final
		Distribution:	public

NeOn Consortium

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

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

Change Log

Version	Date	Amended by	Changes
0.1	01-02-2007	Sebastian Rudolf	Set up original document
0.2	10-06-2007	Mathieu d'Aquin, Antoine Zimmermann, Peter Haase	Initial Structure, sanity check
0.3	15-12-2007	Mathieu d'Aquin	Change in Structure, added Use-cases and State of the art
0.4	25-01-2008	Peter Haase	Template update
0.5	30-01-2008	Mathieu d'Aquin (+ other authors)	Integration of various contributions
1.0	07-02-2008	Mathieu d'Aquin	Corrections, version sent to QA
1.1	25-03-2008	Mathieu d'Aquin	Corrections according to QA, version sent to WP leader

Executive Summary

The goal of this document is to come up with a formalism for ontology modularization, including syntaxes and the fundamental properties of a semantics of such a formalism. Furthermore we introduce operators to create, combine and manipulate ontology modules and give formal definitions for these operators based on the semantics of ontology modules.

Through these elements, we intend to provide a generic formalism that would gather under a common framework the different aspects of ontology modularization (language to specify modules, formal properties, an algebra of operators to create, combine and manipulate modules) with the aim of meeting the requirements of a broad range of application scenarios, having different views and different needs for modularization (in particular within NeOn).

To achieve this, we first identify a number of use cases for ontology modularization, that are motivated and illustrated by concrete example scenarios from the NeOn case studies and other work packages. We then describe previous work that has been targeting different aspects related to our formalism, to finally present the three main elements of our formalism, as motivated and informed by the use cases and related work: syntaxes for modular ontologies that integrate with the NeOn metamodels, the formal properties of this language and its semantics, and operators for combining, creating and manipulating ontology modules.

Contents

List of Figures

Chapter 1

Introduction

One of the major problems that hamper ontology engineering, maintenance and reuse in the current approaches is that ontologies are not designed in a way that facilitates these tasks. To some extent, the problem faced by ontology engineers can be seen as similar to the one faced by software engineers. In both cases, facilitating the management of a system (software or ontology) requires to identify components, modules, that can decoupled from this system, to be exploitable in a different context and integrated with different components. In other terms, building an ontology (and a software) as a combination of independent, reusable modules reduces the effort required for its management, in particular in a collaborative and distributed environment.

This idea has lead to the general notion of modular software in software engineering and is currently gaining more and more attention within the ontology community, as the ontology modularization problem. First approaches have been devised, promoting the development of local ontologies, linked together by mappings [BGvH+03a, KLWZ03]. Another direction of research in the field of ontology modularization concerns the extraction of significant modules from existing ontologies (see e.g. [dSSS07] for an overview).

However, as the definition for a good software module is already vague [Par72], there is no clear agreement on the criteria for decomposing an ontology into modules. Indeed, as shown in [dSSS07], what constitutes a module is subjective and is intrinsically dependent on the application scenario in which ontology modules and modular ontologies are required. Moreover, while a number of studies have been published on different aspects of modularization (languages for modular ontologies, techniques to extract modules from ontologies, formal properties of ontology modules), these elements tend to be disconnected from each other and no complete modularization framework have yet been proposed for ontologies.

For these reasons, we intend to provide a generic formalism that would gather under a common framework the different aspects of ontology modularization (language to specify modules, formal properties, an algebra of operators to create, combine and manipulate modules) with the aim of meeting the requirements of a broad range of application scenarios, having different views and different needs for modularization (in particular within NeOn).

To achieve this, we first identify a number of use cases for ontology modularization, that are motivated and illustrated by concrete example scenarios from the NeOn case studies and other work packages (Chapter 2). We then describe previous work that has been targeting different elements related to our formalism (Chapter 3). The following chapters introduce the three main aspects of the NeOn formalism for ontology modularization, as motivated and informed by the use cases and related work. In Chapter 4, three syntaxes are presented for the modular ontology language: an abstract syntax, a metamodel that integrates with the NeOn metamodel for network ontologies [HBP+07] and a concrete syntax that is presented as an extension of the OMV ontology

metadata vocabulary [HSH+05b]. The formal properties of this language and its semantics are presented in Chapter 5. In Chapter 6, operators for combining ontology modules, creating ontology modules (extracting modules from ontologies or decomposing ontologies into modules) and manipulating ontology modules are presented. Finally, some discussion points about future work, particularly concerning the implementation of the formalism, are raised in Chapter 7.

Chapter 2

Use Cases for Modularization of Ontologies

Before defining a formalism for ontology modularization, it is necessary to better understand the scenarios in which modularizing an ontology is useful or required. This is a difficult task as *modularization of ontologies* can be apprehended differently by different persons, in different applications. In this chapter we identify a number of use cases for ontology modularization that appear in concrete applications within the NeOn case studies or other technical work packages. These use cases will be used in the following sections to guide and illustrate the definition of the NeOn formalism for ontology modularization and of the algebra for manipulating modules.

2.1 Designing Modular Ontologies

The idea of ontology modularization is primarily inspired from the domain of software engineering where modularization refers to the design of software as the combination of self-contained components, easier to build, reuse and maintain than a program made of one, often large and intricate piece of code. Therefore, the most obvious scenario in which ontology modularization is involved is the case of the construction of an ontology not as a monolithic model, but taking benefit from the properties of modular systems: reusability, extensibility and maintainability. In addition, developing ontologies as a set of self-contained modules helps in building more distributed applications, improving the scalability of some ontology based tasks such as reasoning. Of course, in this case, different ontology modules, which may come from different sources, have to be related together in order to be used jointly.

Example 1 (WP7): FAO collects statistics concerning several aspects of fisheries, including captures, aquaculture production, catches, fleets, trade of commodities and consumption. Each piece of statistical data is referenced by the following dimensions: time (in years), space (land and/or water areas) and the variable(s) representing the observed object (e.g. biological species). The data used to indicate these dimensions are called reference data and are (hierarchically) organized in reference tables and stored into a relational database. The first set of fisheries ontologies created for WP7 were based on that reference data (cf. [CG07]).

When modeling reference data as ontologies, it is important that each ontology be devoted to model an (homogeneous) piece of domains (or subdomains) of the reference data. This approach would constitute a “modular design” as each ontology to include in the network would correspond to a piece of the fisheries domain. In many cases, this modeling is already partially embodied into the organization of the reference tables. Examples are the hierarchy of biological species, the division of water bodies into areas used for statistical reporting of catch and production, the classification of

fishing vessels and that one of fishing gears. As discussed in [CG07], a number of ingredients are necessary in order to effectively define and use these ontologies, including automatic methods to create documentation concerning the ontologies and versioning mechanisms.

Although it may be easy to identify homogeneous "pieces" of reference data that can be modeled as individual ontologies, the resulting ontologies will also need to be connected to one another. For example a fishing techniques should always be considered together with the fishing gear used and the vessel types on which the gear is mounted. Also, commodities are naturally linked to the biological species the originate from, and data about fish catch needs to be presented together with the geographical area where the catch happened. Therefore, mechanisms should be in place in order to allow linking of ontologies and their joint exploitation.

Addressing Example 1: In the following, the syntaxes (Chapter 4) and the semantics (Chapter 5) of a language for specifying modules is defined. It allows the ontology designer to encapsulate ontology content into modules, to import external modules and to related modules with each other, hence providing complete support for the design of modular ontologies.

2.2 Partial Import and Reuse

While ideally ontologies would be built in a modular way, most of existing ontologies have not been designed with modularity in mind, hampering their integration –their reuse– in applications other than the one they have been built for. In addition, a significant module in a given application may be too large in another application, and it is not possible to anticipate the requirements of any possible scenario while building modules. Therefore, it is often required to be able to reuse and integrate only a part of a given ontology, by identifying the relevant elements within the ontology and reuse them together with all the information required for their semantic definition, without importing and without having to commit to the entire ontology. Ontology modularization techniques are needed to extract from potentially large and complex ontologies, modules that are relevant and adequate to the task at hand.

Example 2, (WP8): In the context of the invoicing use case, the main objective is to offer a tool that will allow the end-user to map their invoices (instances of a pre-existing or not invoice model) to the invoice reference ontology described in [GPBH$^+$07]. Such a tool should provide access only to the concepts of the ontology that are relevant to the task of the current user. For instance, it is not desirable to provide access to the class "HazardousItemType" when a user in its invoices is just providing toilet paper.

Therefore, a modularization technique is required for showing to the users the parts of the invoice reference ontology they need. More concretely, this would be used for extracting the EDIFACT module from the invoice reference ontology. In this test case, iSOCO provides the invoice reference ontology and from it a module with just the concepts and relations that are needed by the EDIFACT conceptualization are presented. This is particularly useful when an extension of a upper level ontology is performed. Due to its generic conceptualization there are some parts of an upper level ontology which will not be used at certain moments (an end user will most probably not use certain generic concepts, he will focus in the most interesting, specific concepts). To provide a subset/module of this upper level ontology will be highly necessary.

Addressing Example 2: The language for designing modular ontologies (Chapters 4 and 5) integrates the notion of partial import by allowing to specify the elements of imported modules that have to be considered, and therefore, to ignore the others. In addition, as part of a module algebra in Chapter 6, operators to extract modules from existing ontologies are defined.

2.3 Improving Performance

By separating knowledge elements into significant components, modularization allows to focus only to the elements that are relevant for a given application at a given time. Therefore, one obvious use case for modularization is to improve performance, by reducing the amount of knowledge that have to be manipulated by ontology-based tools, including reasoners and editors.

Example 3 (WP7): Data about biological species is used for a number of statistics, such as catch, production, trade of fisheries commodities, distribution of fish stocks etc. (see [CG07], Sec. 4). The backbone of the FAO resources concerning biological species is the ASFIS (Aquatic Science and Fisheries Information System) list of biological species. This list of species is also organized hierarchically (by means of taxonomic codes) and stored into the FIGIS database together with a number of relevant pieces of information, such as: names in various languages, codes according to international classifications (ISO).

The resulting (stand-alone) ontology (see [CG07], Sec. 6.3) is very large (12Mb) and consequently difficult to use, as loading and visualizing it are very memory-consuming tasks that make working with the ontology very slow, when not impossible. However, there are natural "fragments" of this ontology that, for specific tasks and by specific users, could be considered instead of the entire ontology. It would be then useful to identify these pieces and be able to select them out of the entire available data/ontology. In the following we provide some examples of these fragments:

- Different classification systems aim at capturing different aspects of a species. For example, taxonomic codes are meant to capture the biological features of a species, such as the taxonomic chain they belong to, while the ISCAAP (International Standard Statistical Classification of Aquatic Animals and Plants) classification classifies species according to their commercial value. Note that while each species as a unique taxonomic code and each taxonomic code only describes one species, typically more than one species is given the same ISSCAAP code (ISCAAP is actually a classification into groups).

- Other types of fragments have a more complex definition, as they imply a grouping of species according to other criteria. For example, when investigating on the state of fisheries in a certain geographical area, a biologist will typically start by gathering data about the species that can be found in that area.

  In these cases, more than one ontology may be used to define these modules as currently the WP7 ontologies for species and geographical division of water areas are currently distinct. So, in order to be able to only select that “group” of species (possibly together with the taxonomic chain they belong to and their commercial classification according to the ISSCAAP coding), mechanisms should be in place in order to connect species and geographical areas, and only select the species found in a given area.

- Considerations similar to those made above may be applied to modules of the ontology of species based on the basis of the fishing techniques used.

- In some cases, it may also be very useful to be able to identify and select entire "branches" of a taxonomy. Let an example of this be a line of classification of biological species (e.g., for a given species, the family, order and group it belongs to or, conversely, all the biological species classified "under" a given group).
- Finally, another example of relevant module would only contain the species that are used to produce fish meal. This type of module would be defined on the basis of the information currently contained in two ontologies, the one of biological species and the one of commodities.

Addressing Example 3: Operators for extracting modules from ontologies defined in Chapter 6 are useful for working only on the part of an ontology that is relevant, improving performance. The focus here should be on the flexibility of the operators, which should be customizable to the particular needs of the application.

2.4 Facilitating the Exploration and Maintenance of the Ontology

A small module that focuses on a particular topic or domain is obviously easier to apprehend for an ontology designer or an expert of the domain than a large, heterogeneous ontology. In the same line of idea, maintaining and validating ontologies is facilitated when only significant self-contained modules need to be looked at.

Example 4 (WP7): Typically, ontology maintenance work takes place within a workflow where two roles are assigned: subject expert and validators (cf [SCCJ07] Sec 2.3). According to this workflow, any update (addition/modification) to the ontology remains in a “to be approved” status until it is validated by a validator.
Although the ontology may be very large, usually only parts of it is affected by update by subject experts, so only these parts actually need to be validate by validators. In order to facilitate validators’ tasks, it is advisable to give them the possibility to load and/or visualize only the part of the ontology that requires validation. Therefore, it would be very useful to support validators with mechanisms that extract out of the entire ontology only the module that includes the “to be validated” elements, together with the *relevant* elements “around” them. We consider relevant elements all those that should be taken into consideration by validators in order to accomplish their task: they typically include classes above and below the line of subclass of the class under consideration, and the classes linked (as domain/range) by a number of selected properties.

Addressing Example 4: The features of the formalism for modular ontologies and operators already mentioned should provide support for showing parts of ontologies. Here the focus should be on formally defining what should be included into the module, as part of the semantic context of the considered entities.

2.5 Modularization as a Means to Ontology Customization

As mentioned in deliverable D4.4.1 [DKG+07], the proposition to consider access rights is based on the capacity to identify within an ontology a sub-set of ontological entities that need to be treated (for whatever reason) differently than other parts of the ontology. A classic situation that has originally led to investigating access rights and authorities comes from the requirement of individuals or

organizations to preserve certain confidentiality due to sensitivity of data content. Even if several people access the same information, they may see different versions of it, different level of detail, sensitivity, and similarly. Hence, it makes sense to replicate this situation and to create a number of partitions on the ontological model describing such an organizational information system. One advantage of not just denying access at the request stage but rather removing certain modules from a large system/ontology/KB is in the content confinement [DKG+07].

Example 5 (WP4): Imagine a situation where an organization manages an ontology (or a network of ontologies) on a particular aspect related to the organization's interest (say fisheries or medical drugs) –let us label this ontology as O_{shared}. Due to latest developments in the field, subject Alice starts conceptualizing a new chunk of knowledge that leads her to introducing and/or amending certain entities in that existing ontology. Let us denote this chunk as M_{alice}. Obviously, Alice may want to see her changes as if they were already part of the official ontology; e.g. to do some tests. She wants to work with the O_{shared}, however. What she may thus do is committing her amendments in M_{alice} into the official O_{shared} ontology.

Using the access control capability of the shared repository, Alice does her amendments directly in the official O_{shared} because she is normally permitted to do such changes, but she is not yet happy with the current state of her module M_{alice}. Hence, she defines ontology O_{shared} as including module M_{alice}, but she would simultaneously restrict the access to M_{alice} to herself. Since this is a work in progress and perhaps a minor conceptual amendment (e.g. translation of term), Alice feels she may work on the official branch/version of O_{shared}. She only needs that 'lock' on her new additions temporarily – imagine she only wants to consult with Don whether a particular translation is correct or not.

In this situation there is a shared ontology O_{shared}, which is used by Alice and many of her co-workers for very specific purposes and tasks. For sake of this example, assume that Bob is among those co-workers who use O_{shared} and Don is among those who cannot access the ontology in question. We now have one ontology (which includes Alice's new module M_{alice} among other aspects), but our three users see three different things:

Alice sees O_{shared} including the module M_{alice};

Bob sees O_{shared} not including the module M_{alice};

Don sees O_{shared} as being empty

Now Alice needs Don's advice on some issues in her module. In a physical collaborative setting, Alice would simply visit Don in his office and ask for his opinion. In the virtual setting, this can be emulated by Alice granting Don a temporary authority to access her module M_{alice} (or possibly $M_{smaller} \subset M_{alice}$ or even M_{larger} such that $M_{alice} \subset M_{larger} \subset O_{shared}$). As a result, Don will be able to access a particular module (say, M_{larger}), but not the whole ontology.

In the meantime, nothing whatsoever changed for Bob (and any other co-worker) – they still interact with the original, sanitized O_{shared} which hides any of Alice's changes and Don's annotations. At some point Alice and Don reach agreement, and Alice feels confident that now she may make an official proposal for the *extension of* O_{shared}. She simply 'cuts' Don's temporary authorities, and enables her co-workers to obtain authorities appropriate to a specific extension protocol, workflow, or business process. Only at this stage would Bob and other become aware of a change; however, this would already be presented as an officially different ontology – say, $O_{extended}$ integrating O_{shared} and M_{alice}.

Addressing Example 5: Being able to specify a module and its "boundaries" is an important aspect of this use case and is handled by the syntaxes and semantics of the module formalism (Chapters 4 and 5). Moreover, extracting from modules parts that are relevant to the user and parts that should be hidden from him/her is realized through a number of operators of the algebra (Chapter 6). The algebra also provides the essential operations to combine, merge and, in general, manipulate modules.

Chapter 3

State-of-the-art in Ontology Modularization

In the following we provide an overview of existing formalisms for ontology modularization. We first look at approaches for importing ontologies and ontology modules and then, to formalisms that allow the definition of modules and of relations between them. One important part of a modularization formalism concerns operators to create, combine and, more generally, manipulate ontology modules. We then consider the important number of techniques that have been developed for extracting modules from existing ontologies, to finally look at the few existing studies dedicated to the definition of an ontology algebra.

3.1 Knowledge Import

3.1.1 OWL Import

The OWL ontology language provides limited support to modular ontologies: an ontology document –identified via its ontology URI– can be imported by another document using the `owl:imports` statement. The semantics of this import statement is that all definitions contained in the imported ontology document are included in the importing ontology, as if they were defined in the importing ontology document. It is worth mentioning that `owl:imports` is directed –only the importing ontology is affected– and transitive –if A imports B and B imports C, then A also imports the definitions contained in C. Moreover, cyclic imports are allowed (e.g. A imports B and B imports A).

One of the most commonly mentioned weaknesses of the importing mechanism in OWL is that it does not provide any support for partial import [VOM02, PSZ06]. Even if only a part of the imported ontology is relevant or agreed in the importing ontology, every definitions are included. Moreover, there is no logical difference between imported definitions and proper definitions in the importing ontology: they share the same interpretation.

3.1.2 Partial/Semantic Import

Grau et al. [GHKS07] propose a logic-based notion of modularity that allows the modeler to specify the external signature of their ontology, i.e. whose symbols that are reused from some other ontology. The authors define two restrictions on the usage of the external signature, a syntactic and a slightly less restrictive semantic one, each of which is decidable and guarantees a certain kind of black-box behavior that enables the controlled merging of ontologies. To achieve this, certain constraints on the usage of the external signature need to be imposed: in particular, merging ontologies should be *safe* in the sense that they do not produce unexpected results such as new

inconsistencies or subsumptions between imported symbols. For this reason, the authors introduce the notion of *conservative extensions* to define modularity of ontologies, and then prove that a property of some ontologies, called *locality*, can be used to achieve modularity.
A similar notion of partial import is introduced in [PSZ06]: The authors propose a new import primitive, called semantic import, to facilitate partial ontology reuse.

3.1.3 Modularity in Classical Description Logics

Some authors [GK07] argue that modularity can be achieved without introducing new formal languages. In particular they propose translation of so called modular ontology languages (DDL, E-connections, etc.) into standard DL and describe new reasoning services to ensure modularity. Although this might be satisfactory from a logical point of view, it does not take into account engineering aspects like encapsulation, separation of ontologies from mappings, potential heterogeneity of modules, etc.

3.1.4 P-DL

P-DL, Package-based Description Logics [BCH06d], use importing relations to connect local models. In contrast to OWL, which forces the model of an imported ontology to be completely embedded in a global model, the P-DL importing relation is *partial* in that only commonly shared terms are interpreted in the overlapping part of local models. Here, modules are called packages and subsets of the elements (entities) contained in a package can be imported by other packages. The semantics of P-DL is given as the follows: the *image domain relation* between local interpretations $\mathcal{I}_i$ and $\mathcal{I}_j$ (of package P_i and P_j) is $r_{ij} \subseteq \Delta_i \times \Delta_j$. P-DL importing relation is:

- one-to-one: for any $x \in \Delta_i$, there is at most one $y \in \Delta_j$, such that $(x, y) \in r_{ij}$, and vice versa.
- compositionally consistent: $r_{ij} = r_{ik} \circ r_{jk}$, where $\circ$ denotes function composition. Therefore, semantic relations between terms in i and terms in k can be inferred even if k doesn't directly import terms from i.

Thus, a P-DL model is a virtual model constructed from partially overlapping local models by merging "shared" individuals.
P-DL also supports selective knowledge sharing by associating ontology terms and axioms with "scope limitation modifiers (SLM)". A SLM controls the visibility of the corresponding term or axiom to entities on the web, in particular, to other packages. The scope limitation modifier of a term or an axiom t_K in package K is a boolean function $f(p, t_K)$, where p is a URI of an entity, the entity identified by p can access t_K iff $f(p, t) = true$. For example, some representative SLMs can be defined as follows:

- $\forall p, public(p, t) := true$, means t is accessible everywhere.
- $\forall p, private(p, t) := (t \in p)$, means t is visible only to its home package.

P-DL semantics ensures that distributed reasoning with a modular ontology will yield the same conclusion as that obtained by a classical reasoning process applied to an integration of the respective ontology modules [BCH06c]. However, reported result [BCH06a] only supports reasoning in P-DL as extensions of the $\mathcal{ALC}$ DL. Reasoning algorithms for more expressive P-DL TBox, as well as for ABox reasoning, are still to be investigated.

3.2 Linking Modules

3.2.1 Distributed Description Logics

Unlike import mechanisms that include elements from some modules into the considered one, Distributed Description Logics (DDLs) [BS02] adopt a *linking mechanism*, relating the elements of "local ontologies" (called *context*) with elements of external ontologies (contexts). Each context M_i is associated to its own local interpretation. Semantic relations are used to draw correspondences between elements of local interpretation domains. These relations are expressed using *bridge rules* of the form:

- $i : \phi \xrightarrow{\sqsubseteq} j : \psi$ (*into* rule), with semantics: $r_{ij}(\phi^{I_i}) \subseteq \psi^{I_j}$
- $i : \phi \xrightarrow{\sqsupseteq} j : \psi$ (*onto* rule), with semantics: $r_{ij}(\phi^{I_i}) \supseteq \psi^{I_j}$

where $I_i = (\Delta_i, .^{\mathcal{I}_i})$ (respectively $I_j = (\Delta_j, .^{\mathcal{I}_j})$) is the local interpretation of M_i (respectively M_j), ϕ and ψ are formulae, and r_{ij} is a *domain relation* mapping elements of the interpretation domain of M_i to elements of the interpretation domain of M_j ($r_{ij} \subseteq \Delta_i \times \Delta_j$). We only discuss bridge rules between concepts (meaning that ϕ and ψ are concept names or expressions) since it is the only case that has reported reasoning support [ST05].

Bridge rules between concepts cover one of the most important scenarios in modular ontologies: they are intended to assert relationships, like concept inclusions, between elements of two different local ontologies. However, mainly because of the local interpretation, they are not supposed to be read as classical DL axioms. In particular, a bridge rule only affect the interpretation of the target element, meaning for example that $i : \phi \xrightarrow{\sqsubseteq} j : \psi$ is not equivalent to $j : \psi \xrightarrow{\sqsupseteq} i : \phi$.

Arbitrary domain relations may not preserve concept unsatisfiability among different contexts which may result in some reasoning difficulties [BCH06c]. Furthermore, while subset relations (between concept interpretations) is transitive, DDL domain relations are not transitive, therefore bridge rules cannot be *transitively reused* by multiple contexts. Those problems are recently recognized in several papers [BCH06b, BCH06c, SSW05b, SSW05a] and it is proposed that arbitrary domain relations should be avoided. For example, domain relations should be one-to-one [SSW05a, BCH06c] and non-empty [SSW05b].

3.2.2 Variants of DDL

C-OWL: C-OWL [BGvH$^+$03b] derives from DDL by particularising the ontology language to OWL, and by enriching the family of bridge rules. However, the semantics of C-OWL is basically the same as the one of DDL.

DDL with hybrid rules: Sometimes, it happens that a concept in an ontology is represented as a role in another ontology. That is why DDL bridge rules were extended in [GST07] to allow expressing hybrid rules like $i{:}C \xrightarrow{\sqsubseteq} j{:}R$ where C is a concept and R is a role.

DDL revisited: A variant of Distributed Description Logics was proposed in [Hom07]. It defines a new kind of bridge rules called conjunctive bridge rules, which are written $i{:}\phi \xrightarrow{\supseteq} j{:}\psi$ with ϕ and ψ concepts from two different ontologies. The semantics of such conjunctive rules is defined as follows: a distributed interpretation satisfies $i{:}\phi \xrightarrow{\supseteq} j{:}\psi$ with ϕ iff for each conjunctive bridge rules

$i:\Phi \xrightarrow{\supseteq} j:\Psi$ in the system, $r_{ij}(\phi^{\mathcal{I}_i} \cap \Phi^{\mathcal{I}_i}) \supseteq \psi^{\mathcal{I}_j} \cap \Psi^{\mathcal{I}_j}$. These particular rules discard an unintuitive consequence of the semantics of DDL bridge rules. For example, if in an ontology O_1 it is specified that $\mathrm{NonFlying}_1 \equiv \neg\mathrm{Flying}_1$ and $\mathrm{Bird}_1 \sqsubseteq \mathrm{Flying}_1$, the two bridge rules $1:\mathrm{Bird}_1 \xrightarrow{\exists} 2:\mathrm{Penguin}_2$ and $1:\mathrm{NonFlying}_1 \xrightarrow{\exists} 2:\mathrm{Penguin}_2$ would not lead to any inconsistency in classical DDL. The goal of this variant of DDL is to avoid this kind of counterintuitive result.

3.2.3 $\mathcal{E}$-Connection

While DDLs are focused on one type of relation between module elements, concept inclusion, the $\mathcal{E}$-connection approach [KLWZ03, GPS04] allows to define *link properties* from one module to another. For example, if a module M_1 contains a concept named $1{:}Fish$ and a module M_2 contains a concept named $2{:}Region$, one can connect these two modules by defining a link property named $livesIn$ between $1{:}Fish$ and $2{:}Region$.

Formally, given ontology modules $\{L_i\}$, a (one-way binary) link $E \in \mathcal{E}_{ij}$, where $\mathcal{E}_{ij}, i \neq j$ is the set of all links from the module L_i to the module L_j, the following syntax and semantics can be used to construct a concept in module L_i linking through a restriction to a concept C in module L_j (noted $j{:}C$):

- $\exists E.(j{:}C) : \{x \in \Delta_i | \exists y \in \Delta_j, (x,y) \in E^M, y \in C^M\}$
- $\forall E.(j{:}C) : \{x \in \Delta_i | \forall y \in \Delta_j, (x,y) \in E^M \rightarrow y \in C^M\}\}$
- $\leq nE.(j{:}C) : \{x \in \Delta_i | \#(\{y \in \Delta_j | (x,y) \in E^M, y \in C^M\}) \leq n\}$
- $\geq nE.(j{:}C) : \{x \in \Delta_i | \#(\{y \in \Delta_j | (x,y) \in E^M, y \in C^M\}) \geq n\}$

where $M = \langle \{m_i\}, \{E^M\}_{E \in \mathcal{E}_{ij}} \rangle$ is a model of the $\mathcal{E}$-connected ontology, m_i is the local model of L_i; C is a concept in L_j, with interpretation $C^M = C^{m_j}$; $E^M \subseteq \Delta_i \times \Delta_j$ is the interpretation of a $\mathcal{E}$-connection E.

$\mathcal{E}$-connection restricts the terms of modules, as well as their local domains, to be disjoint. This can be a serious limitation in some scenarios, particularly because, to enforce domain disjointness, subclass relations cannot be declared between concepts of two different modules.

3.2.4 Integrated Distributed Description Logics

IDDL [Zim07] is another formalism for distributed reasoning upon networked DL knowledge base. Similarly to DDL, an IDDL interpretation allocates a different interpretation to each ontology but instead of relating domains directly, they are correlated in another domain called global domain of interpretation. The intuition behind this formalism is that ontology mappings may be provided by third party agents which assert correspondences from a point of view encompassing both mapped ontologies. With that perspective in mind, using directional bridge rules like DDL is quite unsatisfactory. A consequence of the semantics of this formalism is the transitivity of ontology mappings.

This formalism instantiates a generic distributed semantics presented in [ZE06] in the case of ontologies represented in description logics. The notion of equalising function –i.e. an abstract function that is part of the global interpretation and map elements of a local domain to elements of the global domain– is used to correlate local domains of interpretation from different ontologies into a unique global domain. The generic semantics is used in Chapter 5.

3.3 Extracting Modules from Existing Ontologies

We consider in this section techniques and tools that have been developed to help users in extracting or creating modules from existing, and potentially large scale ontologies. We start our analysis by briefly introducing notations and distinguishing two major types of techniques: Ontology module extraction techniques and ontology partitioning. We then describe different techniques in each of these categories. A more complete analysis of these tools can be found in [dSSS07] and a proposal for a common framework to unify these techniques can be found in [dDMT07].

Notations. We consider an ontology O as a set of axioms (subclass, equivalence, instantiation, etc.) and the signature $Sig(O)$ of an ontology O as the set of entity names occurring in the axioms of O, i.e. its vocabulary.

In the following, we deal with several approaches for ontology modularization, having different assumptions about the definition of an ontology module. The assumption we adopt as a basis for our discussion is that a module is considered to be a significant and self-contained sub-part of an ontology. Therefore, a module $M_i(O)$ of an ontology O is also a set of axioms (an ontology), such that $Sig(M_i(O)) \subseteq Sig(O)$.

Ontology Partitioning. The task of partitioning an ontology is the process of splitting up the set of axioms into a set of modules $\{M_1, \cdots, M_k\}$ such that each M_i is an ontology and the union of all modules is semantically equivalent to the original ontology O. Note that some approaches being labeled as *partitioning* methods do not actually create *partitions*, as the resulting modules may overlap. There are several approaches for ontology partitioning that have been developed for different purposes.

The approach of [MMAU03] aims at improving the efficiency of inference algorithms by localizing reasoning. For this purpose, this technique minimizes the shared language (i.e. the intersection of the signatures) of pairs of modules. A message passing algorithm for reasoning over the distributed ontology is proposed for implementing resolution-based inference in the separate modules. Completeness and correctness of some resolution strategies is preserved and others trade completeness for efficiency.

The approach of [GPSK05] partitions an ontology into a set of modules connected by ε-Connections. This approach aims at preserving the completeness of local reasoning within all created modules. This requirement is supposed to make the approach suitable for supporting selective use and reuse since every module can be exploited independently of the others.

A tool that produces sparsely connected modules of reduced size was presented in [SK04]. The goal of this approach is to support maintenance and use of very large ontologies by providing the possibility to individually inspect smaller parts of the ontology. The algorithm operates with a number of parameters that can be used to tune the result to the requirements of a given application.

Module Extraction. The task of module extraction consists in creating a new module by reducing an ontology to the sub-part that covers a particular sub-vocabulary. This task has been called segmentation in [SR06] and traversal view extraction in [NM04]. More precisely, given an ontology O and a set $SV \subseteq Sig(O)$ of terms from the ontology, a module extraction mechanism returns a module M_{SV}, supposed to be the relevant part of O that covers the sub-vocabulary SV ($Sig(M_{SV}) \supseteq SV$). Techniques for module extraction often rely on the so-called *traversal*

approach: starting from the elements of the input sub-vocabulary, relations in the ontology are recursively "traversed" to gather relevant (i.e. related) elements to be included in the module.

Such a technique has been integrated in the PROMPT tool [NM04], to be used in the Protégé environment. This approach recursively follows the properties around a selected class of the ontology, until a given distance is reached. The user can exclude certain properties in order to adapt the result to the needs of the application.

The mechanism presented in [SR06] starts from a set of classes of the input ontology and extracts related elements on the basis of class subsumption and OWL restrictions. Some optional filters can also be activated to reduce the size of the resulting module. This technique has been implemented to be used in the Galen project and relies on the Galen Upper Ontology.

In [Stu06a], the author defines a viewpoint as being a sub-part of an ontology that only contains the knowledge concerning a given sub-vocabulary (a set of concept and property names). The computation of a viewpoint is based on the definition of a viewpoint dependent subsumption relation.

Inspired from the previously described techniques, [dSM06] defines an approach for the purpose of the dynamic selection of relevant modules from online ontologies. The input sub-vocabulary can contain either classes, properties, or individuals. The mechanism is fully automatized and is designed to work with different kinds of ontologies (from simple taxonomies to rich and complex OWL ontologies) and relies on inferences during the modularization process.

Finally, the technique described in [DTI07] is focused on ontology module extraction for aiding an Ontology Engineer in reusing an ontology module. It takes a single class as input and extracts a module about this class. The approach it relies on is that, in most cases, elements that (directly or indirectly) make reference to the initial class should be included.

3.4 Ontology Algebra

As modular ontologies are made of the combination of different ontology modules, operators are required to support the ontology designer in composing modules, creating them, and more generally, manipulating them. There have been a few studies on possible operators in an ontology algebra and, since an ontology module is essentially an ontology, these can be a source of inspiration for an ontology module algebra.

In [Wie94], Wiederhold defines a very simple ontology algebra, with the main purpose of facilitating ontology-based software composition. He defines a set of operators applying set-related operations on the entities described in the input ontologies, and relying on equality mappings ($=$) between these entities. More precisely, the three following operators are defined.

$Intersection(O_1, O_2) \rightarrow O$	create an ontology O containing the common (mapped) entities in O_1 and O_2.
$Union(O_1, O_2) \rightarrow O$	create an ontology O containing the entities of O_1 and O_2, and merging the common ones.
$Difference(O_1, O_2) \rightarrow O$	create an ontology O containing only the entities of O_1 that are not mapped to entities of O_2

In the same line of ideas, but in a more formalized and sophisticated way, [MBHR04] describes a set of operators for model management, as defined in the RONDO platform [MRB03]. The goal of model management is to facilitate and automatize the development of metadata-intensive applications by relying on the abstract and generic notion of *model* of the data, as well as on the idea

of *mappings* between these models. An essential part of a platform for model management is a set of operators to manipulate and combine these models and mappings. [MBHR04] focuses on formalizing a core set of operators: Match, Compose, Merge, Extract, Diff and Confluence. Match is particular in this set. It takes 2 models as an input and returns a mapping between these models. It inherently does not have a formal semantics as it depends on the technique used for matching, as well as on the concrete formalism used to describe the models and mappings. Merge intuitively corresponds to the Union operator in [Wie94]: it takes two models and a mapping and creates a new model that contains the information from both input models, relying on the input mapping. It also creates two mappings from the created model to the two original ones. Extract creates the sub-model of a model that is involved in a mapping and Diff the sub-model that is not involved in a mapping. Finally, compose and confluence are mapping manipulation operators creating mappings by merging or composing other mappings.

[KFWA06] defines operators for combining ontologies created by different members of a community and written in RDF. This paper first provides a formalization of RDF to describe set-related operators such as intersection, union and difference. It also adds other kind of operators, such as the quotient of two ontologies O_1 and O_2 (collapsing O_2 into one entity and pointing all the properties of O_1 to entities of O_2 to this particular entity) and the product of two ontologies (inversely, extending the properties of from O_1 to O_2 to all the entities of O_2). It is worth mentioning that such operators can be related to the ones of relational algebras, used in relational database systems.

Note finally that the OWLTools[1] that are part of the KAON2 framework include operators such as diff, merge and filter working at the level of ontology axioms. For example, merge creates an ontology as the union of the axioms contained in the two input ontologies.

[1] `http://owltools.ontoware.org/`

Chapter 4

Syntaxes and Metamodel

In this chapter, we propose the definition of a formalism for ontology modularization, focusing on the aspects related to the syntax. We first look at the requirements for such a formalism, as extracted from the use cases and from the state-of-the-art. Three definitions of the language in terms of syntax are then proposed: an abstract syntax –used to define the elements of the formalism and employed as a notation in this report, the metamodel of the modularization formalism –used to integrate this formalism with the other parts of the NeOn knowledge model and as a reference for implementation– and finally, a proposal for a concrete syntax based on OMV –used to actually encode the description of ontology modules.

4.1 Requirements for a Module Definition Language

A Module is an Ontology. As shown in the previous overview, there is generally no clear distinction between the notion of ontology and the one of ontology module. A modular ontology is made of smaller local ontologies that can be seen as self-contained and inter-related modules, combined together for covering a broader domain. Indeed, an ontology is not inherently a module, but rather plays the role of a module for other ontologies because of the way it is related to them in an ontology network. In other terms, an ontology module is a self-contained ontology, seen according to a particular perspective, namely *reusability*. The content of an ontology module does not differ from the one of an ontology, but a module should come with additional information about how to reuse it, and how it reuse other modules.

Encapsulation / Information Hiding. The idea of encapsulation is crucial in modular software development, but has not really been studied and implemented in the domain of ontologies yet. In software engineering, it relies on the distinction between the implementation, i.e. internal elements manipulated by the developer of the module, and the interface, i.e. the elements that are exposed to be reused. This distinction between interface and implementation cannot be clearly stated when using ontology technologies like OWL. However, the essential role of a module interface is to guide the reusability of the module, by exposing reusable elements and hiding intermediary internal ones (the "implementation details"). By defining the set of reusable entities of an ontology module (the *export interface*), the developer of this module provide entry points to it, and clearly states which are the elements that can be "safely reused" (e.g. the ones that are considered stables). Elements that are hidden behind the interface can then evolve, be re-designed or changed, without affecting the importing modules relying on this export interface.

Partial Import. As already mentioned, the `owl:imports` mechanism has been criticized in several papers for being "global" (see e.g. [VOM02, PSZ06]): it is not possible when using this mechanism to import only the relevant and useful elements in the importing ontology. Allowing partial import has many advantages, among which *scalability* is probably the most obvious. For this reason, some intermediary solutions have been recently proposed, using, prior to import, ontology partitioning [SK04, GPSK05] techniques or some forms of reduction to a subvocabulary [SR06, Stu06b, dSM06]. We believe that the set of elements that are used in an importing module should be explicitly stated in the module definition, so that the influence of the imported module is clarified. The semantics of the module definition language should reflect the idea of partial import by "ignoring" the definitions that are not related to the imported elements (the *import interface*), preventing the importing module to deal with irrelevant knowledge, and giving the developer of such a module the possibility to ignore the parts of the imported modules he/she does not want to commit to.

Links Between Modules. The formalisms for modular ontologies presented in the previous section can be divided in two main approaches: importing and linking. The previous requirements are focused on the importing approach, whereas languages like C-OWL and $\mathcal{E}$-connection exclusively deal with the linking approach. In the NeOn framework, these two aspects are relevant, and should be considered together: even when they are not imported, elements from different modules can be related through mappings. The NeOn metamodel already provides the required elements for expressing mappings, and these can easily be considered as a part of the content of an ontology module. However, it is important to take this aspect into account when designing the semantics of the module description language, as well as the operations for manipulating modules, so that the two approaches, importing and linking, are well integrated. Indeed, scenario where, for example, there exist mappings between imported modules are not hard to imagine.

4.2 Abstract Syntax

The goal of this section is to come up with an abstract syntax for the ontology modularization formalism, identifying the necessary information to be accommodated in an ontology module as well as structural properties of a modularized networked ontology. This will be done on a solid formal basis which will enable us to define a corresponding semantics at a later stage.

We start by defining sets of identifiers being used for unambiguously referring to ontology modules and mappings that might be distributed over the Web. Obviously, in practice, URIs will be used for this purpose. So we let

- Id_{Modules} be a set of MODULE IDENTIFIERS and
- Id_{Mappings} be a set of MAPPING IDENTIFIERS, where a mapping is a set of relations (correspondences) between entities of two different ontologies.

Next we introduce generic sets describing the used ontology language. They will be instantiated depending on the concrete ontology language formalism used (e.g., OWL). Hence, let:

- Nam be a set of NAMED ELEMENTS.
  In the case of OWL, Nam will be thought to contain all class names, property names and individual names.

- $Elem$ be a the set of ONTOLOGY ELEMENTS.
  In the OWL case $Elem$ would contain e.g. all complex class descriptions. Clearly, $Elem$ will depend on Nam (or roughly speaking: Nam delivers the "building blocks" for $Elem$).
- We use $L : 2^{Nam} \rightarrow 2^{Elem}$ to denote the function assigning to each set P of named elements the set of ontology elements which can be generated out of P by the language constructs[1],
- For a given set O of ontology axioms, let $\mathrm{Sig}(O)$ denote the set of named elements occurring in O, so it represents those elements the axioms from O deal with.

Having stipulated those basic sets in order to describe the general setting, we are now ready to state the notion of an ontology module on this abstract level.

Definition 1 *An* ONTOLOGY MODULE $\mathcal{OM}$ *is a tuple* $\langle id, Imp, \mathcal{I}, M, O, E\rangle$ *where*

- $id \in Id_{\mathrm{Modules}}$ *is the identifier of* $\mathcal{OM}$
- $Imp \subseteq Id_{\mathrm{Modules}}$ *is a set of identifiers of imported ontology modules (referencing those other modules whose content has to be (partially) incorporated into the module),*
- $\mathcal{I}$ *is set* $\{I_{id}\}_{id \in Imp}$ *of* IMPORT INTERFACES*, with* $I_{id} \subseteq Nam$ *(characterizing which named elements from the imported ontology modules will be "visible" inside* $\mathcal{OM}$*),*
- $M \subseteq Id_{\mathrm{Mappings}}$ *is a set of identifiers of imported mappings (referencing – via mapping identifiers – those mappings between ontology modules, which are to be taken into account in* $\mathcal{OM}$*),*
- O *is a set of* ONTOLOGY AXIOMS *(hereby constituting the actual content of the ontology),*
- $E \subseteq \mathrm{Sig}(O) \cup \bigcup_{id \in Imp} \mathcal{I}_{id}$ *is called* EXPORT INTERFACE *(telling which named entities from the ontology module are "published", i.e., can be imported by other ontology modules).*

As a simple example, let us consider an ontology module $\mathcal{OM}_i$ (with module identifier i) completely importing another ontology module $\mathcal{OM}_k$ (with module identifier k) and exporting everything:
$\mathcal{OM}_i = \langle i, \{k\}, \{\mathrm{Sig}(O_k)\}, \emptyset, O_i, \mathrm{Sig}(O_i) \cup \mathrm{Sig}(O_k)\rangle$
Note that, in order to simplify the notation, we will not specify explicitly an identifier for the module: a module $\mathcal{OM}_i$ will be considered as implicitly having "$\mathcal{OM}_i$" as identifier and will so be written: $\mathcal{OM}_i = \langle Imp_i, \mathcal{I}_i, M_i, O_i, E_i\rangle$.

In a further step we formally define the term mapping (which is supposed to be a set of directed links, correspondences, between two ontology modules establishing semantic relations between their entities).

Definition 2 *A* MAPPING M *is a tuple* $\langle s, t, C\rangle$ *with*

- $s, t \in Id_{\mathrm{Modules}}$*, with* s *being the identifier of the source ontology module and* t *being the identifier of the target ontology module,*
- C *is a set of* CORRESPONDENCES *of the form* $e_1 \rightsquigarrow e_2$ *with* $e_1, e_2 \in Elem$ *and* $\rightsquigarrow \in R$ *for a fixed set* R *of* CORRESPONDENCE TYPES[2]

[1] In most cases – and in particular for OWL – $L(P)$ will be infinite, even if P is finite.
[2] In accordance with the NeOn metamodel, this set will be fixed to $R = \{\sqsubseteq, \sqsupseteq, \equiv, \perp, \not\sqsubseteq, \not\sqsupseteq, \not\equiv, \not\perp\}$

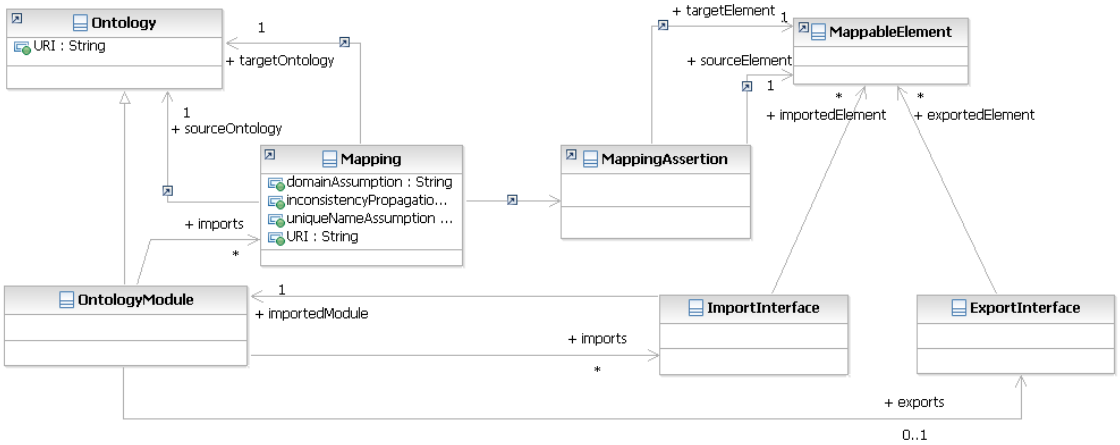

Figure 4.1: Metamodel extensions for ontology modules.

4.3 Metamodel

We propose a generic metamodel for modular ontologies according to the design considerations discussed above. The metamodel is a consistent extension of the metamodels for OWL DL ontologies and mappings [HBP+07].

Figure 4.1 shows elements of the metamodel for modular ontologies. The central class in the metamodel is the class `OntologyModule`. A module is modeled as a specialization of the class `Ontology`. The intuition behind this modeling decision is that every module is also considered an ontology, enriched with additional features. In other words, a module can also be seen as a role that a particular ontology plays. In addition, an ontology provides (at most) one `ExportInterface` and a set of `ImportInterface`. The interfaces define the elements that are exposed by the imported module and reused by the importing module. The elements that can be reused by modules are `MappableElement`s (defined in the mapping metamodel). A mappable elements is either an `OWLEntity` or a `Query` over an ontology, meaning that entities can be exposed in interfaces either directly or as the results of queries.

The export interface, modeled via the `exports` association, exposes the set of `OntologyElement`s that are intended to be reused by other modules.

The reuse of elements from one module by another module is represented via the `ImportInterface`. The association `imports` relates the importing module with the definition of the imports interface. The association `importedModule` refers to the module that is being imported, while the association `importedElement` refers to the element in the imported ontology being reused. In this sense, the `ImportInterface` can be seen to realize the ternary relationship between the importing ontology, the imported ontology, and the elements to be reused.

Additionally, a `Module` also provides an `imports` relationship with the `Mapping` class, which is used to relate different ontology modules via ontology mappings.

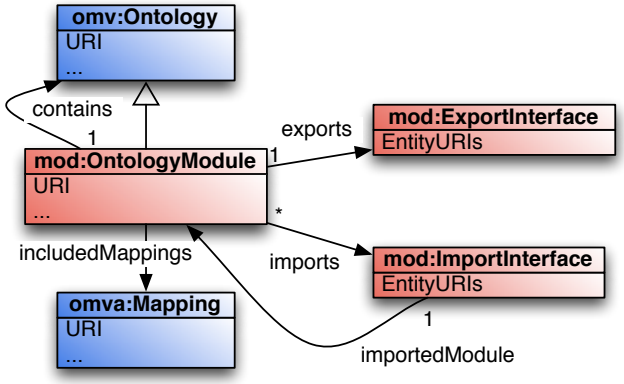

Figure 4.2: Overview of the OMV extension for Ontology Modules.

4.4 Concrete Syntax

In order for the modularization formalism to be usable, it requires at least one concrete syntax that implements the elements of the abstract syntax and of the metamodel at a technological level. Ideally, this syntax should integrate with OWL in a non-intrusive and backward compatible way, to keep the definition of modules as flexible as possible. In particular it is important that standard tools for OWL that would not support our modularization mechanism could ignore the definition of modules and continue to work in the same way, even if they would obviously not take benefit from the features provided by modularization.

We made the choice to implement this concrete syntax as an extension of OMV [HSH+05a]. OMV is a ontology metadata vocabulary, and it could appear strange to define modules as ontology metadata, but according the above definitions, an ontology module is nothing but an ontology associated with additional information regarding interfaces and mappings. Of course, these “metadata” would have an influence on the semantics of the module, so this choice is still questionable. However, OMV already includes definitions for ontologies and mappings, and we would anyway have to define a metadata descriptor for modules that would include the same information.

Figure 4.2 describes the OMV extension for modularization (blue classes are classes already in OMV or in the mapping extension, and the red ones are new). It is built in accordance with the metamodel. For example, a module `http://example.org/A`, encapsulating an ontology `http://example.org/O`, importing several entities from modules `http://example.org/B` and `http://example.org/C`, exporting the entities `http://example.org/A#X`, `http://example.org/A#Y` and `http://example.org/B#Z`, and including the mappings `http://example.org/M` and `http://example.org/N` would be described in the following way:

```
<mod:OntologyModule rdf:ID="http://example.org/A">
  <mod:contains>
     <omv:Ontology rdf:about="http://example.org/O" />
  </mod:contains>
  <mod:exports>
     <mod:ExportInterface>
        <mod:interfaceElement rdf:resource="http://example.org/A#X" />
        <mod:interfaceElement rdf:resource="http://example.org/A#Y" />
        <mod:interfaceElement rdf:resource="http://example.org/B#Z" />
     </mod:ExportInterface>
  </mod:exports>
  <mod:imports>
     <mod:ImportInterface>
```

```
        <mod:importedModule rdf:resource="http://example.org/B" />
        <mod:interfaceElement rdf:resource="http://example.org/B#U" />
        <mod:interfaceElement rdf:resource="http://example.org/B#V" />
        <mod:interfaceElement rdf:resource="http://example.org/B#Z" />
      </mod:ImportInterface>
      <mod:ImportInterface>
        <mod:importedModule rdf:resource="http://example.org/C" />
        <mod:interfaceElement rdf:resource="http://example.org/C#T" />
      </mod:ImportInterface>
   </mod:imports>
   <mod:includedMapping>
      <omva:Mapping rdf:about="http://example.org/M" />
   </mod:includedMapping>
   <mod:includedMapping>
      <omva:Mapping rdf:about="http://example.org/N" />
   </mod:includedMapping>
</mod:OntologyModule>
```

4.5 Examples

One of the goals of the NeOn modularization language is to provide a mean for ontology engineers to develop ontologies in a modular way, specifying the behavior and role of the ontology modules involved in this development. Example 1 Section 2.1 corresponds to such a scenario: the fishery ontology is made of several modules, corresponding to different sub-domains of the fishery domain (fish species, vessels, techniques, etc.). Each of these modules can be specified using the proposed syntaxes, to indicate their dependencies, interconnections and contributions (exports) to the modular ontology. For example, the *species* module would be declared as taking its content from the species ontology (`http://www.fao.org/aims/aos/fi/species_v1.0.owl`) and exporting the entities corresponding to species of fishes of interest for the application (that can be all of them). In the proposed abstract syntax, this would correspond to[3]:

$$species\_onto = \texttt{http://www.fao.org/aims/aos/fi/species\_v1.0.owl}$$
$$species\_entities = \{\texttt{species}, \texttt{Goldspotted spinef}, \texttt{Marbled octopus}, \ldots\}$$
$$species\_module = \langle \emptyset, \emptyset, \emptyset, species\_onto, species\_entities \rangle$$

Now considering that the (OWL translation of the) AGROVOC thesaurus also contains descriptions of fish species, it can be envisaged to replace the species module of the fishery ontology by an alternative one, having AGROVOC for content and exporting from it only entities corresponding to fishery species:

$$agrovoc = \texttt{http://www.fao.org/agrovoc/}$$
$$species\_entities = \{\texttt{species}, \texttt{Goldspotted spinef}, \texttt{Marbled octopus}, \ldots\}$$
$$species\_module = \langle \emptyset, \emptyset, \emptyset, agrovoc, species\_entities \rangle$$

This alternative provides the same elements as the previous one and so, can play the same role in the application: since the actual content of the module is encapsulated, it can be easily replaced or modified without having to re-consider the applications relying on the module. Note that if the content of AGROVOC needs to be adapted to fit the module specification, another way to achieve the same result is to create a module importing AGROVOC, together with an ontology (the content of the module) defining additional axioms for the adaptation of AGROVOC.

[3] note that the identifiers of the entities have been replace by the name of the species

Once all the modules of the considered application are specified, an overall module can be built by importing the “component modules”:

$$
\begin{aligned}
&species\_onto = \texttt{http://www.fao.org/aims/aos/fi/species\_v1.0.owl}\\
&species\_entities = \{\texttt{species}, \texttt{Goldspotted spinef}, \texttt{Marbled octopus}, \dots\}\\
&vessels\_onto = \texttt{http://www.fao.org/aims/aos/fi/vessels\_v1.0.owl}\\
&vessels\_entities = \{\dots\}\\
&land\_areas\_onto = \dots\\
&fishery\_module = \langle\{species\_onto, vessels\_onto, land\_areas\_onto, \dots\},\\
&\qquad\qquad \{species\_entities, vessels\_entities, \dots\}, \emptyset, \emptyset, exported\_entities\rangle
\end{aligned}
$$

Of course, one of the interest of modular design is that the modules can be combined differently for different applications, importing only the relevant ones, and even only the relevant parts of them. We can easily imagine, building another module for an application only interested in some particular fishery gears and in some particular vessels (using these gears):

$$
\begin{aligned}
&gears\_onto = \texttt{http://www.fao.org/aims/aos/fi/gears\_v1.0.owl}\\
&gears\_entities = \{\textit{gears of interest}\dots\}\\
&vessels\_onto = \texttt{http://www.fao.org/aims/aos/fi/vessels\_v1.0.owl}\\
&vessels\_entities = \{\textit{vessels of interest}\dots\}\\
&module = \langle\{gears\_onto, vessels\_onto\}, \{gears\_entities, vessels\_entities\},\\
&\qquad \emptyset, \emptyset, exported\_entities\rangle
\end{aligned}
$$

Finally, in most of the cases, the content of imported modules need to be *aligned*, so that they can be used jointly. This is achieved by specifying mappings between the entities of the imported module. Here for example, a mapping can be specified in the overall module to align the *fishing area* module with the *land area module*, or to relate *gears* with the associated *vessels*:

$$
\begin{aligned}
&gears\_onto = \texttt{http://www.fao.org/aims/aos/fi/gears\_v1.0.owl}\\
&gears\_entities = \{\textit{gears of interest}\dots\}\\
&vessels\_onto = \texttt{http://www.fao.org/aims/aos/fi/vessels\_v1.0.owl}\\
&vessels\_entities = \{\textit{vessels of interest}\dots\}\\
&vessels\_gears\_mapping = \texttt{http://www.fao.org/mappings/vessels\_gears}\\
&module = \langle\{gears\_onto, vessels\_onto\}, \{gears\_entities, vessels\_entities\},\\
&\qquad \{vessels\_gears\_mappings\}, \emptyset, exported\_entities\rangle
\end{aligned}
$$

Example 2 Section 2.2 also provides a good illustration of one of the principles of the modularization language as described above: partial import. Indeed, in this example, a module is to be created for representing invoice related information, by reusing the part of the EDIFACT ontology that concerns invoices. Therefore, this module would import the EDIFACT ontology restricted, through the import

interface, to the entities related to the *Invoice* class. The content of this module can integrate these imported entities and the ontology designer can decide whether or not they have to be exported, exposing them as reusable elements or hiding them as implementation artifact.

Chapter 5

Semantics

In order to reason with modular ontologies, we have to define the semantics of the module definition language we are using, i.e., define interpretations and models of a module upon which entailment is defined.

Before describing the actual semantics we define for modular ontologies, we take a look at the requirements wrt semantics (Section 5.1). Then, the semantics of the local content of a module is defined in a classical model theoretic way (Section 5.2). Mappings between imported modules have their own semantics described in Section 5.3. According to the semantics of these two components, we define interpretations and models of a module by using a generic semantics proposed in [ZE06].

5.1 Requirements for the semantics of modular ontologies

The main problem when dealing with the semantics of networked ontologies is about heterogeneity. Since we want to connect and reason with ontologies that may have been developed independently, possible discrepancies may exist between two ontologies. Not only differences in terms but also in the modelling of a knowledge domain. A possible way to treat this problem is by revising knowledge when incoherence exists. This issue is discussed in deliverable D1.2.1 and D1.2.2 about resolving inconsistencies. However, it is not always possible to revise knowledge because one may not be able to enforce revision for a module developed separately, possibly by another party.

So another solution to the problem is to use or define non standard semantics for distributed or modular ontologies. Chapter 3 already discussed existing formalisms. We can sort these formalisms according to their robustness to heterogeneity. Robust formalisms are the ones which impose the loosest between interpretations of the local ontologies. Conversely, less robustness means tighter connections. The list of formalisms from the tightest to the loosest is as follows: DL $\rightarrow$ P-DL $\rightarrow$ IDDL $\rightarrow$ DDL/$\mathcal{E}$-Connection $\rightarrow$ no connection. The tightest formalism is classical DL (like OWL) which enforces each module to be interpreted in the same domain. P-DL may have different domains but strongly connected. IDDL allows arbitrary domains but connect them to a global domain via functions, which allows mapping composition while keeping somewhat high robustness. DDL and $\mathcal{E}$-connection connect interpretation very loosely, but bridge rules and links are not transitional. The loosest formalism possible (which we call “no connection”) corresponds to having completely separate and unrelated interpretations for each ontology in a distributed system. This is not very useful for modularisation but may be useful to obtain separate answers according to different view points.

For our purposes, we need a formalism for reasoning with modules that may have been developed separately. Interpreting all the modules as one big ontology would not be appropriate, so this ex-

clude the classical DL semantics. P-DL is much more appropriate since it allows partial reuse of ontologies and each module may have a different interpretation. However there is no support for ontology mapping. Therefore, semantic relations between modules must be represented by axioms in the importing module. Unfortunately, the semantics of P-DL imposes that imported concepts have the very same interpretation in the importing and imported module. Consequently, if a concept has a certain number of instances, it must have the very same instances in the importing module. Although this is adequate when the whole modular ontology is designed in a well delimited environment, it reduces robustness to heterogeneity quite a lot. In particular, the use cases presented in Chapter 2 are not totally compliant with this feature. DDL is more adapted to heterogeneous environments. Yet, bridge rules have several disadvantages with respect to the situation of networked ontologies. First, they are not transitive, which restrains their reusability. Second, they express knowledge about the system *from one particular ontology point of view*. This implies that the knowledge they are describing only concerns the ontology toward which the rules are directed. Consequently, bridge rules are not appropriate for representing mappings, which are often produced from a standpoint encompassing both mapped ontologies. $\mathcal{E}$-connection, though different in its principles, suffer from the same disadvantages.

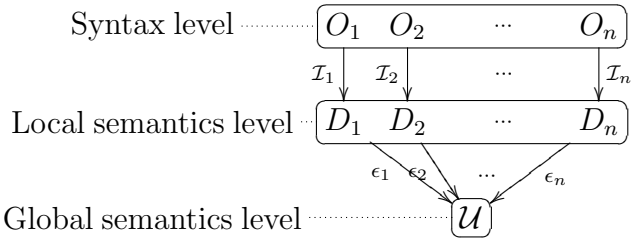

Figure 5.1: This figure shows the interpretations of local ontologies, which are correlated into a global domain through the equalizing function ϵ. Mappings are interpreted in the global domain.

IDDL offers a good compromise between robustness to heterogeneity and knowledge propagation. It is mostly based on the principle that ontology mappings are produced by third party tools or agents, so they have to be treated at a separate level. It distinguishes reasoning locally with one ontology and reasoning with the whole network of ontologies and mappings by relying on local interpretations for ontologies, that are mapped into a unique global domain (see Figure 5.1 where $\mathcal{I}_i$ represents a local interpretation for O_i, relying on the local domain D_i, and $\mathcal{U}$ is the global domain, integrating a local domain D_i through an "equalising function" ϵ_i). For the purpose of modular ontologies, it means that imported ontologies behave as a network of ontologies, and the importing module represents the global system.

5.2 Semantics of the local content of modules

Interpreting the local content of a module is equivalent to interpreting axioms of a non-modular ontologies. Since the formalism used to write axioms in our module framework is based on OWL, this local semantics corresponds to a description logic semantics.

Definition 3 (Interpretation) *Given a set of ontology elements $Elem$ (individuals, classes and properties), an* INTERPRETATION *of $Elem$ is a pair $\langle \Delta, [\![.]\!] \rangle$, where*

Δ is a non-empty set, called the DOMAIN*,*

[.] *is a function from* $Elem$ *to* $\Delta \cup \mathcal{P}(\Delta) \cup \mathcal{P}(\Delta \times \Delta)$*, where* $\mathcal{P}(x)$ *is the part set of* x*.*

In our specific case (considering OWL), the function **[.]** maps

- an individual $\mathtt{a}$ to an element of Δ: **[**$\mathtt{A}$**]** $\in \Delta$
- a class $\mathtt{C}$ to a subset of Δ: **[**C**]** $\subseteq \Delta$
- a property $\mathtt{p}$ to a binary relation between elements of Δ: **[**$\mathtt{p}$**]** $\subseteq \Delta \times \Delta$

In fact, an interpretation of the named elements Nam, uniquely defines an interpretation of the ontology elements by applying inductive interpretation rules:

- **[**$C \sqcap D$**]** $=$ **[**C**]** $\cap$ **[**D**]**,
- **[**$C \sqcup D$**]** $=$ **[**C**]** $\cup$ **[**D**]**,
- **[**$\exists R.C$**]** $= \{x | \exists y. y \in$ **[**C**]** $\wedge \langle x, y \rangle \in$ **[**R**]**$\}$,
- etc.

Interpretations are related to axioms thanks to the satisfaction relation $\models$.

Definition 4 (Satisfaction) *An interpretation* $\langle \Delta,$ ***[.]*** $\rangle$ *satisfies:*

- *an axiom* $\mathtt{C} \sqsubseteq \mathtt{D}$ *if* ***[***$\mathtt{C}$***]*** $\sqsubseteq$ ***[***$\mathtt{D}$***]***
- *an axiom* $\mathtt{C}(\mathtt{a})$ *if* ***[***$\mathtt{a}$***]*** $\in$ ***[***$\mathtt{C}$***]***
- *an axiom* $\mathtt{p}(\mathtt{a}, \mathtt{b})$ *if* (***[***$\mathtt{a}$***]***, ***[***$\mathtt{a}$***]***) $\in$ ***[***$\mathtt{p}$***]***

If an interpretation I *satisfies an axiom* α*, it is denoted by* $I \models \alpha$*.*

The local content of a module is characterized by a set of axioms.

Definition 5 *An interpretation is a model of a set of axioms* O *if it satisfies all the axioms in* O*. The set of models of a set of axioms* O *is denoted* $Mod(O)$*.*

5.3 Satisfied mapping

A mapping connects entities from 2 different ontologies or modules. Interpreting them implies interrelating both ontology (or module) interpretations.
Entities appearing in a correspondence can be interpreted according to the ontology language semantics. Since each ontology may be interpreted in different domains, we define a notion of equalising function which helps making these domains commensurate.

Definition 6 (Equalising function) *Let* Ω *be a set of ontologies and for all* $o \in \Omega$*,* $I_o = \langle \Delta_o,$ ***[.]***$_o \rangle$ *be an interpretation of* o*. An equalising function* ϵ *for* $(I_o)_{o \in \Omega}$ *assigns to each* o *a function* $\epsilon_o : \Delta_o \rightarrow \Delta$ *to a common global domain of interpretation* Δ*.*

Besides, the mapping language defines a set of relation symbols that are used to express relations between ontology entities. The interpretation of such relation is defined by the mapping language semantics, according to the global domain of interpretation. More precisely, each relation symbol $r \in \mathfrak{R}$ and each global domain Δ is associated to a binary relation $r^\Delta \subseteq \Delta \times \Delta$.

In this deliverable, the mapping language is characterized by the relation symbols $R = \{\sqsubseteq, \sqsupseteq, \equiv, \perp, \not\sqsubseteq, \not\sqsupseteq, \not\equiv, /\perp\}$. The binary relations associated to them are: set inclusion $r^\Delta = \{(X,Y) \in \Delta \times \Delta \mid X \subseteq Y\}$, set containment $r^\Delta = \{(X,Y) \in \Delta \times \Delta \mid X \supseteq Y\}$, set equality $r^\Delta = \{(X,Y) \in \Delta \times \Delta \mid X = Y\}$, set disjunction $r^\Delta = \{(X,Y) \in \Delta \times \Delta \mid X \cap Y = \emptyset\}$, and their complements.

Using these notions, we can determine whether a correspondence is satisfied by the interpretations of the mapped ontologies.

Definition 7 (Satisfied correspondence) *Let $c = \langle e_1, e_2, r\rangle$ be a correspondence in a mapping between O_1 and O_2. A correspondence is satisfied by two interpretations $\mathcal{I}_1 = \langle \Delta_1, [\![.]\!]_1\rangle$ and $\mathcal{I}_2 = \langle \Delta_2, [\![.]\!]_2\rangle$ of O_1 and O_2 respectively, if there exists an equalising function ϵ for $(\mathcal{I}_1, \mathcal{I}_2)$ over global domain Δ such that $(\epsilon_1([\![e_1]\!]_1), \epsilon_2([\![e_2]\!]_2)) \in r^\Delta$. This is written $\mathcal{I}_1, \mathcal{I}_2 \models_\epsilon c$.*

For instance, consider the correspondence $c = \langle \mathtt{Cottage}_1, \mathtt{Building}_2, \sqsubseteq\rangle$, then $\mathcal{I}_1, \mathcal{I}_2 \models c$ iff $\epsilon_1([\![\mathtt{Cottage}_1]\!]) \subseteq \epsilon_2([\![\mathtt{Building}_1]\!])$.

Definition 8 (Satisfied mapping) *A mapping A of ontologies O_1 and O_2 is satisfied by a pair of interpretations $\langle \mathcal{I}_1, \mathcal{I}_2\rangle$ if there exists an equalising function ϵ of $\langle \mathcal{I}_1, \mathcal{I}_2\rangle$ such that for each $c \in A$, $\mathcal{I}_1, \mathcal{I}_2 \models_\epsilon c$.*

Note that a mapping can be satisfied by interpretations that are not themselves models of the local ontologies. This is useful when one needs to determine consistency of a mapping, but do not have access to the ontologies. Moreover, this also ensures encapsulation at the mapping level, since it prevents mapping satisfiability to be dependent on a particular ontology implementation.

5.4 Global interpretation of modules

The interpretation of a module is recursively defined in function of the interpretations of its imported modules.

This recursive definition assumes that there is no cycle in the import chain, so each chain eventually leads to a base module with no import. Detection of cycles should be syntactically checked, since this definition is not well founded otherwise. If one thinks in term of software engineering, this is not a major limitation. Indeed, when a new module is designed, it has to import existing modules. This way, it is not possible to have cyclic references.

Definition 9 (Base module interpretation) *Let $\mathfrak{M} = \langle \emptyset, \emptyset, \emptyset, O, E\rangle$ be a base module. An interpretation of $\mathfrak{M}$ is a local interpretation $\mathcal{I}$ of the content O of $\mathfrak{M}$, with domain of interpretation $\mathcal{D}$.*

A module interpretation is defined recursively according to the import chain.

Definition 10 (Module interpretation) *Let $\mathfrak{M} = \langle M, I, A, O, E\rangle$ be a module. An interpretation of $\mathfrak{M}$ is a triple $\mathfrak{I} = \langle \mathcal{I}, (\mathcal{I}_m)_{m\in M}, \epsilon\rangle$ such that:*

- *For each imported module* $m \in M$*,* $\mathcal{I}_m$ *is a* module interpretation *of* m *over domain of interpretation* $\mathcal{D}_m$*;*
- ϵ *is an equalising function for* $(\mathcal{I}_m)_{m\in M}$*, over a global domain of interpretation* Δ*;*
- $\mathcal{I} = \langle \Delta, [\![.]\!] \rangle$ *is a (local) interpretation of the content* O *of* $\mathfrak{M}$*, with domain of interpretation* Δ*.* Δ *is also called the* domain of interpretation *of module* $\mathfrak{M}$*;*
- *the interpretation of the imported terms* $t_m \in I_m$ *of module* $m \in M$ *is defined by* $[\![t_m]\!] = \epsilon_m([\![t_m]\!]_m)$*.*

In order for an interpretation to satisfy a module, there are three conditions:

1. the local interpretation must be a model of the content of the module, i.e., all local axioms must be satisfied;
2. the imported modules must be satisfied by their respective interpretations;
3. the mappings between the imports must be satisfied by the respective pairs of interpretations;

Definition 11 (Model of a module) *Let* $\mathfrak{M} = \langle M, I, A, O, E \rangle$ *be a module and* $\mathfrak{I} = \langle \mathcal{I}, (\mathcal{I}_m)_{m\in M}, \epsilon \rangle$ *a module interpretation of* $\mathfrak{M}$*.* $\mathfrak{I}$ *is a* model *of* $\mathfrak{M}$ *(written* $\mathfrak{I} \models \mathfrak{M}$*) iff:*

- *for each imported module* $m \in M$*,* $\mathcal{I}_m \models m$ *(i.e., each imported module is locally satisfied);*
- $\mathcal{I}$ *is a model of* O *(i.e., the local content of* $\mathfrak{M}$ *is satisfied);*
- *for each pair of modules* $m, m' \in M$*,* $\mathcal{I}_m, \mathcal{I}_{m'} \models A_{m,m'}$ *(i.e., all mappings are satisfied).*

The set of all the models of a module $\mathfrak{M}$ *is written* $\mathrm{Mod}(\mathfrak{M})$ *too.*

The notion of models is essential for automatic deduction in modular ontologies. It serves to define which formulas are semantic consequences of a module (i.e., entailment).

5.4.1 Consequences of a module

In order to reason with modular ontologies, we have to define what are the semantic consequences of a module. They are defined as follows:

Definition 12 (Consequences of a module) *Let* $\mathfrak{M} = \langle M, I, A, O, E \rangle$ *be a module. Let* δ *be an axiom built upon the signature of the content of* $\mathfrak{M}$ *(which includes the import interfaces of* I*).* δ *is a consequence of* $\mathfrak{M}$*, written* $\mathfrak{M} \models \delta$ *iff for all* $\langle \mathcal{I}, (\mathcal{I}_m)_{m\in M}, \epsilon \rangle \in \mathrm{Mod}(\mathfrak{M})$*,* $\mathcal{I} \models \delta$*.*

Obviously, if a formula is a consequence of the content ontology of a module, then it is a consequence of the module itself.[1] Additionally, it is desirable to derive knowledge about the imported terms according to the imported modules knowledge. However, if something is true about a concept C in a module, it is not necessarily true in another module that imports C. For instance, in a description logic knowledge base, if a module m is such that $m \models \neg C \sqsubseteq D$, it does not follow that, considering a module $\mathfrak{M} = \langle \{m\}, \{C, D\}, \emptyset, \emptyset, \emptyset \rangle$, $\mathfrak{M} \models \neg C \sqsubseteq D$, as the domains of interpretation of $\mathfrak{M}$ and m may not be the same.

In order to characterize formulas that can be propagated from a module to its importers, we define a general notion of locality, inspired by [GHKS07]. A formula is semantically local when its satisfiability in a module implies its satisfiability in a module that imports its terms.

[1] Note that only the consequences related to the exported terms are useful to an external module that imports them.

Definition 13 (Semantic locality) *Let m be a module. Let α be a formula written in terms of the export interface. α is* semantically local *iff for all modules $\mathfrak{M}$ that uses m and import the terms of α:*

$$m \models \alpha \longrightarrow \mathfrak{M} \models \alpha$$

A module m is semantically local iff all its axioms are local.

As seen in [GHKS07], locality can be computationally checked in description logic $\mathcal{SHOIQ}$. Semantic locality is clearly a desirable property to design "safe" modules. Indeed, in a semantically local module, what is true of a term in the module, is also true in a module that imports it.

So we have given a semantics to ontology modules that is different from owl:import semantics. In particular, it hides imported module implementation and uses standard mapping semantics to connect modules together. Since we do not specify the concrete language used for ontologies and ontology mappings, our framework can be applied to expressive languages, not restricted to a predefined set of semantic relations. Moreover, separating mappings from the ontology allows for the manipulation of mappings independently. These features are not present in previous work on modular ontology formalisms.

Chapter 6

Algebra

In this chapter, we identify and explain operators that are useful in manipulating and combining modules. We provide the semantics of these operators relying on the notion of consequence defined in the previous chapter ($\models$), alternative definitions that comply with this semantics, and illustrative examples of the usefulness of these operators.

We distinguish three main categories of operators: binary operations for composing modules (intersection, union, difference), operators for extracting sub-modules (decomposition, reduction) and other kind of operators including purely syntactic operations for facilitating the manipulation of modules (interface completion, collapse) and the match operator.

6.1 Binary Module Composition Operators

These operators, inspired by operators on sets, produce a module from the composition of two modules. Their signature is then always $Module \times Module \rightarrow Module$. These are the operators that are common in all the ontology algebras described in Section 3.4. They are indeed useful in many scenarios, in particular when designing modular ontologies from existing modules, like it is the case for instance in Example 1 Section 2.1: the union of all the modules corresponding to sub-domains of fisheries provides an initial version of the fisheries ontology.

6.1.1 Union

Description The *Union* operator creates a new module by merging the content of two other ones.

Semantics for any axiom α, $Union(M_1, M_2) \models \alpha$ *if* $M_1 \models \alpha \vee M_2 \models \alpha$

Properties commutative, associative, idempotent.

Example In Section 4.5, creating a module from $vessels\_module$ and $gears\_module$ can be done by union: $Union(vessels\_module, gears\_module)$.

Possible definitions A simple way to comply with the semantics of the union operator is that the created module imports the ones that are combined:

- $\mathcal{OM}_i = \langle Imp_i, \mathcal{I}_i, M_i, O_i, E_i \rangle$ and
- $\mathcal{OM}_j = \langle Imp_j, \mathcal{I}_j, M_j, O_j, E_j \rangle$

$Union(\mathcal{OM}_i, \mathcal{OM}_j) = \langle \{\mathcal{OM}_i, \mathcal{OM}_j\}, \{E_i, E_j\}, \emptyset, \emptyset, E_i \cup E_j \rangle$

Another way to achieve *Union* is to actually copy and merge the contents of the modules. This should be computationally more complicated and less flexible, but the newly created module can then evolve independently of the original ones.

- $\mathcal{OM}_i = \langle Imp_i, \mathcal{I}_i, M_i, O_i, E_i \rangle$ and
- $\mathcal{OM}_i = \langle Imp_j, \mathcal{I}_j, M_j, O_j, E_j \rangle$

$Union(\mathcal{OM}_i, \mathcal{OM}_j) =< k, \emptyset, \emptyset, M_i \cup M_j, O_i \cup O_j, E_i \cup E_j >$. Below, a *Collapse* operator is defined that retrieves all the information concerning a module locally. Combined with the first (import) definition of Union, it provides similar properties to this second definition.

6.1.2 Difference

Description The difference of two modules corresponds to the part of the first module that is not in the second one.

Semantics for any axiom α, $Difference(M_1, M_2) \models \alpha$ *iff* $M_1 \models \alpha \wedge M_2 \not\models \alpha$

Properties not commutative, not associative, $Difference(M, M) = empty\_module$

Example The difference operator can be useful to spot changes between different versions of a module (cf. Example 4 Section 2.4).

Possible Definitions Approximated definitions can be realized by applying set differences to the elements of the modules. However, a definition complying entirely with the semantics of the difference operator is more difficult to achieve and requires the use of a reasoner.

6.1.3 Intersection

Description Intersection extracts the common part of two modules.

Semantics for any axiom α, $Intersection(M_1, M_2) \models \alpha$ *iff* $M_1 \models \alpha \wedge M_2 \models \alpha$

Properties commutative, not associative, idempotent.

Example Finding the common part of two modules can be useful to extract a common reusable *pattern* employed by both. In addition, it can provide the basis for the definition of other operators.

Definition If both Union and Difference are defined, intersection can be easily computed in the following way:

$$Intersection(\mathcal{OM}_1, \mathcal{OM}_2) = Difference(Union(\mathcal{OM}_1, \mathcal{OM}_2), \\ Union(Difference(\mathcal{OM}_1, \mathcal{OM}_2), Difference(\mathcal{OM}_2, \mathcal{OM}_1)))$$

Otherwise, approximations can be achieved using the set-intersection of the elements of the modules, or a reasoner could be used.

6.2 Module Extraction Operators

They are two main approaches for extracting modules from bigger modules or ontologies: decomposing it into a set of significant parts or reducing its content according to a particular signature, relevant for a given application [dSSS07]. As explained in the previous chapters, ideally, ontologies would be designed in a modular way, so that they would be easier to maintain, reuse, etc. However, most of existing ontologies are not designed with modularity in mind. In addition, an appropriate module in one application, or one scenario, may need to be reduced, decomposed and reorganized for other applications and scenarios. This is particularly clear in several of our use cases that require the extraction of modules for improving performances (Section 2.3), facilitating the maintenance and exploration of ontologies (Section 2.4), or customizing them (Section 2.5).

6.2.1 Reduction/Module Extraction

Description This operator reduces the content of a module according to a particular interface. It is supposed to keep only the axioms that have an influence on the interpretation of the entities in the interface.

Signature $Reduce : Module \times Interface \rightarrow Module$

Semantics for any axiom α, $Reduce(M, I) \models \alpha$ *iff* $M \models \alpha \wedge Sig(\alpha) \subseteq I$.

Properties $Reduce(\mathcal{OM}, Sig(\mathcal{OM})) = \mathcal{OM}$, $Reduce(\mathcal{OM}, \emptyset) = empty\_module$

Examples Within the modularization framework, this operator is useful to support the definition of other operators. It can also be used to *compile* the content of a module with respect to its export interface. More importantly, a number of our use cases explicitly rely on the feature provided by the reduce operator. In particular, Example 3 Section 2.3 details possible ways in which large ontologies (such as FAO's species ontology) could be reduced to improve the performance of ontology-based tools. In the same way, in Example 4 Section 2.4, the reduce operator can be used to reduce a modified ontology to the part that is related to the change: in a sense, compiling the *semantic context* of the entities that have been modified.

Possible definitions [GHKS07] provides a formalization of the notion of modularity –based on the notion of *conservative extension*– that complies with the semantics of the reduction operator as defined here. However, as shown by Example 3 (Section 2.3), there can be many different ways to extract a module from an ontology, depending on the requirements of the particular scenario in which this operator is used and on how it defines an appropriate module. For this reason, there have been a number of different techniques to extract modules, relying on different criteria and resulting in different modules. [dDMT07] defines a common framework for ontology module extraction techniques that can be *parametrized* according to application requirements. Extraction procedures can be specified using transformation rules to be applied on the ontology. A number of existing techniques have already been reformulated within this framework, and new ones, tailored to particular applications, can easily be created.

6.2.2 Reduction by Hiding

Description This operator can be seen as the inverse of the previous one. Instead of reducing a module to the wanted signature, it reduces the module by removing the given signature, providing a module that does not mention the elements of this signature.

Signature $Hide : Module \times Interface \rightarrow Module$

Semantics for any axiom α, $Hide(M, I) \models \alpha$ *iff* $M \models \alpha \wedge Sig(\alpha) \cap I = \emptyset$.

Properties $Hide(\mathcal{OM}, \emptyset) = \mathcal{OM}$, $Hide(\mathcal{OM}, Sig(\mathcal{OM})) = empty\_module$

Usages examples This operator is directly motivated by the need for access right support for ontologies and ontology modules (see Example 5, Section 2.5)), as it can be used to reduce an ontology to the part a user has the right to see and manipulate.

Possible definitions This operator can be implemented by removing from the content axioms of the module the elements given as input, replacing them by more general elements not in this interface. A similar procedure is described and studied in [Stu06a]. In addition, with an appropriate definition of the Reduce operator, Hide could be defined by the difference between the original module and its reduction according to the given signature: $Hide(\mathcal{OM}, E) = Difference(\mathcal{OM}, Reduce(\mathcal{OM}, E))$.

6.2.3 Decomposition/Partitionning

Description This operator divides an existing module into parts that should correspond to significant components. The resulting modules should be related by mappings.

Signature $Decompose : Module \rightarrow 2^{Module}$

Semantics for any axiom α, $M \models \alpha$ *iff* $Union(Decompose(M)) \models \alpha$

Properties in some definition, the decomposition result in a partition, meaning that $Intersection(Decomposition(M)) = empty\_module$, but this is not always the case.

Example Decomposing an existing ontology into modules facilitates the maintenance of the ontology and helps in using it, making possible its exploration "by pieces" and the distribution of reasoning mechanisms (see Sections 2.3 and 2.4).

Possible definitions In the literature, several approaches for semantics preserving partitioning have been described. For example, [SK04] proposes a partitioning technique based on the structural properties of the ontology. [GPSK05] presents an approach for generating logical modules as self-contained units within an ontology and that can be safely extracted without adding or removing entailments in the signature of other modules.

6.3 Match and Other Syntactic Operations

Here we present operators that do not fall into the previously defined categories. First the Match operator is useful in integrating modules, as it creates mappings between the content of two modules. The following ones are said to be syntactic in the sense that they have no influence on the semantics of the modules they are applied to: the resulting modules have the same set of consequences as the given modules. These operators are useful as they simplify the use and manipulation of ontology modules.

Description Given two modules, the Match operator returns a mapping that holds between the modules. The mappings are correspondences between the elements of the export interfaces of the modules.

Signature $Match : Module \times Module \rightarrow Mapping$

Semantics The Match operator inherently does not have a formal semantics. Instead, it finds correspondences based on heuristics.

Example Creating mappings between modules is useful in any scenario where different modules need to be integrated, in particular when designing modular ontologies (see Example 1 Section 2.1).

Possible Definitions For matching of modules, one can directly reuse the variety of approaches known from the field of ontology matching and alignment. The concrete implementation of this operator would rely on the alignment server described in [EdSZ07] and on the matching techniques it will host.

6.3.1 Collapse

Description Collapsing corresponds to the creation of a new module that contains the same knowledge as the original one, but that physically integrates imported elements.

Signature $collapse : Module \rightarrow Module$

Example This can be useful to copy a module locally and use it offline.

Possible Definition Without mappings in the considered module, a simple definition for the Collapse operator could be:

- $\mathcal{OM} = \langle Imp, \mathcal{I}, M, O, E \rangle$

$collapse(\mathcal{OM}) =< \emptyset, \emptyset, \emptyset, O \cup \bigcup_{\mathcal{OM}_i \in Imp} content(Reduce(Collapse(\mathcal{OM}_i), I_i)), E)$,
where $content(\langle Imp, \mathcal{I}, M, O, E \rangle) = O$ and Reduce is the operator defined above. Note that this definition is recursive and that it would terminate when reaching modules without import. Therefore, it does not allow cycles in the graph based on the import relation between modules.

The introduction of mappings would make the definition dependent on the formal semantics used for the modularization formalism (see Chapter 5). Under certain conditions, it is possible to transform mapping assertions into local axioms in the content of the collapsed modules.

6.3.2 Interface Completion

Description Interface completion takes two modules and creates the import interface between these two modules.

Signature $completeInterface : Module \times Module \rightarrow Interface$

Definition

- $\mathcal{OM}_i = \langle Imp_i, \mathcal{I}_i, M_i, O_i, E_i \rangle$
- $\mathcal{OM}_j = \langle Imp_j, \mathcal{I}_j, M_j, O_j, E_j \rangle$

$CompleteInterface(\mathcal{OM}_i, \mathcal{OM}_j) = Sig(O_i) \cap Sig(O_j)$

Example When building a module, Interface Completion allows the developer to compute the import interfaces so that the module is valid and minimizes the dependency to the imported module. It can be useful for example when building a module from a OWL ontology that uses the $owl{:}imports$ mechanism, to reduce the part that is imported from external ontologies.

Chapter 7

Discussion

In this deliverable, we have defined a new formalism for specifying, combining and manipulating modules. The different elements we have been looking at are: first different syntaxes for this formalism, then the semantics of this formalism, and finally, a set of operators to manipulate modules described in this formalism and constituting an ontology module algebra. The definition of this language and of these operators is directly motivated by a number of use cases and example scenarios occurring in NeOn, and aims at providing a general framework to support these different visions and needs for ontology modularization. This is not the first attempt at defining a formalism for modular ontologies and the work presented in this deliverable has been informed by previous studies. Indeed, elements from related work have been integrated to provide a broader, more generic framework that is the NeOn formalism for ontology modularization.

There are several aspects that are important in modular ontologies and that have not been considered in this deliverable. In particular, the implementation of this formalism will be described in the next NeOn deliverable on ontology modularization (D1.1.4). Moreover, other elements studied in other tasks of NeOn work package 1 (T1.2 on managing inconsistencies and T.1.3 on change propagation) should be emphasized in a module ontology framework and will require particular attention. Finally, the definition of the NeOn formalism for ontology modules relates to (and impacts on) several other activities within the NeOn project, in different work packages.

Implementation. The implementation of the NeOn formalism for ontology modularization first requires the integration of the modularization language into the datamodel of NeOn, in particular as implemented within the NeOn toolkit. Plugins for the NeOn toolkit should be provided to help ontology engineers in specifying modules for their ontologies. Apart from the operators, these plugins should provide mechanisms to facilitate the definition of modules. It should be possible to simply transform a standard OWL ontology into an ontology module carrying the same semantics. In addition, a reasoner handling the definition of ontology modules –as specified using the proposed concrete syntax– should be provided and should comply with the chosen semantics.

Moreover, the operators of the module algebra will be implemented as a series of plugins for the NeOn toolkit, but, beyond the simple implementation of the techniques required to execute the operators, the ability to create modules dynamically from operator-based *scripts* is needed (in a mechanism similar to the creation of views in relational database systems). It is important indeed that the implementation of the algebra makes possible the management of both materialized modules, which could be manipulated independently of their original ontologies (like in example 3 Section 2.3 for improving performances), and of dynamic modules, which correspond to encapsulations of the ontologies they are based on (like in example 1 Section 2.1 where the content of the modules are stored in databases).

Further considerations. Incompatibility between modules –i.e., combination of modules that create inconsistencies– may be encountered when integrating modules from various sources. The definition of the operators for combining modules do not take into consideration this aspect and assume that the modules that are integrated are compatible. Also, the dynamic aspect of modules is an important elements to consider in the future. Indeed, sophisticated mechanisms are required for making sure that changes operated on a module would not affect other modules relying on it, or that changes are propagated from imported modules to importing ones and from modified modules to modules that have been extracted from them.

At a more general level, as part of the work realized in work package 5, methodologies for designing modular ontologies should be elaborated to provide guidelines and best practices on the basis of the module specification language and of the operators for manipulating modules. In particular, the availability of tools to support modularization impacts on the task of knowledge reuse, also considered in work package 2. Moreover, it is believed that the ontology module formalism provides a formal basis in many different tasks where “components of ontologies” have to be manipulated. Ontology customization (WP4) has already been mentioned as a possible use-case, and generally relates to the definition of modules dependent on the context in which they will be used. This relation between modules and context should be explored further (in collaboration with WP3). In a more concrete way, ontology design patterns (and more specifically, content design patterns) as defined in [PGD+08] can be considered as ontology modules: they are reusable parts, components, of particular significance. Therefore, some of the operations described in [PGD+08] to manipulate content design patterns (clone, composition, import, etc) could be implemented as particular instances of operators defined in the module algebra.

Bibliography

[BCH06a] J. Bao, D. Caragea, and V. Honavar. A distributed tableau algorithm for package-based description logics. In *the 2nd International Workshop On Context Representation And Reasoning (CRR 2006), co-located with ECAI 2006*. 2006.

[BCH06b] J. Bao, D. Caragea, and V. Honavar. Modular ontologies - a formal investigation of semantics and expressivity. In *R. Mizoguchi, Z. Shi, and F. Giunchiglia (Eds.): Asian Semantic Web Conference 2006, LNCS 4185*, pages 616–631, 2006.

[BCH06c] J. Bao, D. Caragea, and V. Honavar. On the semantics of linking and importing in modular ontologies. In *I. Cruz et al. (Eds.): ISWC 2006, LNCS 4273 (In Press)*, pages 72–86. 2006.

[BCH06d] J. Bao, D. Caragea, and V. Honavar. Towards collaborative environments for ontology construction and sharing. In *International Symposium on Collaborative Technologies and Systems (CTS 2006)*, pages 99–108. IEEE Press, 2006.

[BGvH+03a] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, and H. Stuckenschmidt. C-OWL: Contextualizing ontologies. In *Second International Semantic Web Conference ISWC'03*, volume 2870 of *LNCS*, pages 164–179. Springer, 2003.

[BGvH+03b] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, and H. Stuckenschmidt. C-owl: Contextualizing ontologies. In *International Semantic Web Conference*, pages 164–179, 2003.

[BS02] A. Borgida and L. Serafini. Distributed description logics: Directed domain correspondences in federated information sources. In *CoopIS/DOA/ODBASE*, pages 36–53, 2002.

[CG07] C. Caracciolo and A. Gangemi. D7.2.2 revised and enhanced fisheries ontologies. NeOn Deliverable 7.2.2, NeOn Consortium, 2007.

[dDMT07] M. d'Aquin, P. Doran, E. Motta, and V. Tamma. Towards a parametric ontology modularization framework based on graph transformation. In *Second International Workshop on Modular Ontologies (WoMO'2007)*, 2007.

[DKG+07] M. Dzbor, A. Kubias, L. Gridinoc, A. Lopez-Cima, and C. Buil Aranda. D4.4.1 : The role of access rights in ontology customization. Neon deliverable, NeOn Consortium, 2007.

[dSM06] M. d'Aquin, M. Sabou, and E. Motta. Modularization: a key for the dynamic selection of relevant knowledge components. In *Workshop on Modular Ontologies*, 2006.

[dSSS07] M. d'Aquin, A. Schlicht, H. Stuckenschmidt, and M. Sabou. Ontology modularization for knowledge selection: Experiments and evaluations. In Roland Wagner, Norman Revell, and Günther Pernul, editors, *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, volume 4653 of *Lecture Notes in Computer Science*, pages 874–883. Springer, 2007.

[DTI07] P. Doran, V. Tamma, and L. Iannone. Ontology module extraction for ontology reuse: An ontology engineering perspective. In *Proceedings of the 2007 ACM CIKM International Conference on Information and Knowledge Management*, 2007.

[EdSZ07] J. Euzenat, M. d'Aquin, M. Sabou, and A. Zimmermann. D3.3.1: Matching ontologies for context. NeOn Deliverable 3.3.1, NeOn Consortium, 2007.

[GHKS07] B. Cuenca Grau, I. Horrocks, Y. Kazakov, and U. Sattler. A logical framework for modularity of ontologies. In *Proc. of 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 298–303, 2007.

[GK07] B. Cuenca Grau and O. Kutz. Modular ontology languages revisited. In *Workshop on Semantic Web for Collaborative Knowledge Acquisition (SWeCKa'2007*, 2007.

[GPBH+07] J. M. Gòmez-Pérez, C. Buil, G. Herrero, T. Pariente, A. Baena, J. Candini, and J. C. Dalm. D8.3.1 ontologies for the pharmaceutical case studies. NeOn Deliverable 8.3.1, NeOn Consortium, 2007.

[GPS04] B. Cuenca Grau, B. Parsia, and E. Sirin. Working with multiple ontologies on the semantic web. In *International Semantic Web Conference*, pages 620–634, 2004.

[GPSK05] B. Cuenca Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Automatic partitioning of owl ontologies using -connections. In *Description Logics*, 2005.

[GST07] C. Ghidini, L. Serafini, and S. Tessaris. On relating heterogeneous elements from different ontologies. In Boicho N. Kokinov, Daniel C. Richardson, Thomas Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2007, Roskilde, Denmark, August 20-24, 2007, Proceedings*, volume 4635 of *Lecture Notes in Computer Science*, pages 234–247. Springer, 2007.

[HBP+07] P. Haase, S. Brockmans, R. Palma, J. Euzenat, and M. d'Aquin. D1.1.2 updated version of the networked ontologymodel. NeOn Deliverable 1.1.2, NeOn Consortium, 2007.

[Hom07] M. Homola. Distributed Description Logics Revisited. In *Proc. of the 20th International Workshop on Description Logics DL'07*. Bolzano University Press, 2007.

[HSH+05a] J. Hartmann, Y. Sure, P. Haase, R. Palma, and M. C. SuÂůrez-Figueroa. OMV – ontology metadata vocabulary. In Chris Welty, editor, *ISWC 2005 Workshop on Ontology Patterns for the Semantic Web*, NOV 2005.

[HSH+05b] J. Hartmann, Y. Sure, P. Haase, R. Palma, and M.-C. SuÃąrez-Figueroa. Omv – ontology metadata vocabulary. In Chris Welty, editor, *ISWC 2005 - In Ontology Patterns for the Semantic Web*, NOV 2005.

[KFWA06] S. Kaushik, C. Farkas, D. Wijesekera, and P. Ammann. An algebra for composing ontologies. In *Formal Ontology in Information Systems (FOIS)*, 2006.

[KLWZ03] O. Kutz, C. Lutz, F. Wolter, and M. Zakharyaschev. E-connections of description logics. In *Description Logics Workshop, CEUR-WS Vol 81*, 2003.

[MBHR04] S. Melnik, P. A. Bernstein, A.Y. Halevy, and E. Rahm. A semantics for model management operators. Microsoft technical report, June 2004.

[MMAU03] B. MacCartney, S. McIlraith, E. Amir, and T.E. Uribe. Practical Partition-Based Theorem Proving for Large Knowledge Bases. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[MRB03] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *Proc. SIGMOD*, pages 193–204, 2003.

[NM04] N.F. Noy and M.A. Musen. Specifying Ontology Views by Traversal. In *Proc. of the International Semantic Web Conference (ISWC)*, 2004.

[Par72] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[PGD+08] V. Presutti, A. Gangemi, S. David, G. Aguado de Cea, M. C. Suárez-Figueroa, E. Montiel-Ponsoda, and M. Poveda. D2.5.1: A library of ontology design patterns: reusable solutions for collaborative design of networked ontologies. NeOn Deliverable 2.5.1, NeOn Consortium, 2008.

[PSZ06] J.Z. Pan, L. Serafini, and Y. Zhao. Semantic import: An approach for partial ontology reuse. In *Workshop on Modular Ontologies*, 2006.

[SCCJ07] M. Iglesias Sucasas, C. Baldassarre C. Caracciolo, and Y. Jaques. D7.1.2 revised specifications of user requirements for the fisheries case study. NeOn Deliverable 7.1.2, NeOn Consortium, 2007.

[SK04] Heiner Stuckenschmidt and Michel C. A. Klein. Structure-based partitioning of large concept hierarchies. In *International Semantic Web Conference*, pages 289–303, 2004.

[SR06] J. Seidenberg and A. Rector. Web ontology segmentation: Analysis, classification and use. In *Proceedings of the World Wide Web Conference (WWW)*, Edinburgh, June 2006.

[SSW05a] L. Serafini, H. Stuckenschmidt, and H. Wache. A formal investigation of mapping languages for terminological knowledge. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence - IJCAIÂě05*, Edinburgh, UK, August 2005.

[SSW05b] H. Stuckenschmidt, L. Serafini, and H. Wache. Reasoning about ontology mappings. Technical report, Department for Mathematics and Computer Science, University of Mannheim ; TR-2005-011, 2005.

[ST05] L. Serafini and A. Tamilin. Drago: Distributed reasoning architecture for the semantic web. In *European Semantic Web Conference - ESWC*, pages 361–376, 2005.

[Stu06a] H. Stuckenschmidt. Toward Multi-Viewpoint Reasoning with OWL Ontologies. In *Proc. of the European Semantic Web Conference (ESWC)*, 2006.

[Stu06b] H. Stuckenschmidt. Towards multi-viewpoint reasoning with OWL ontologies. In *European Semantic Web Conference*, 2006.

[VOM02] R. Volz, D. Oberle, and A. Maedche. Towards a Modularized Semantic Web. In *Semantic Web Workshop*, Hawaii, 2002.

[Wie94] G. Wiederhold. An algebra for ontology composition. In *Monterey Workshop on Formal Methods*, 1994.

[ZE06] A. Zimmermann and J. Euzenat. Three Semantics for Distributed Systems and their Relations with Alignment Composition. In *Proc. of 5th International Semantic Web Conference (ISWC'06)*, volume 4273 of *LNCS*, pages 16–29. Springer, 2006.

[Zim07] A. Zimmermann. Integrated Distributed Description Logics. In *Proc. of the 20th International Workshop on Description Logics DL'07*. Bolzano University Press, 2007.