NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

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

D1.1.4 NeOn Formalism for Modularization: Implementation and Evaluation

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

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

Other Authors: **Peter Haase, Chan Le Duc, Antoine Zimmermann**

The topic of ontology modularization corresponds to a variety of tasks including the design of modular ontologies and the modularization of existing ontologies. In a previous deliverable, we defined the formal foundation for the support of some of these activities within NeOn, in the form of a formalism and a set of operators for defining, creating, combining and manipulating modules. In this document, we describe the implementation of a set of plugins for the NeOn Toolkit, particularly focusing on the task of specifying modules, combining modules, partitioning ontologies into a set of modules and extracting modules from ontologies. This set of interoperable plugins provide an interactive environment for ontology engineers to cope with ontology modularization. Other aspects related to ontology modularization, namely methodologies and reasoning for modular ontologies, will be treated in different deliverables and are therefore only briefly considered in this document.

Document Identifier:	NEON/2008/D1.1.4/V1.0	Date due:	31 October 2008
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	31 October 2008
Project start date	March 1, 2006	Version:	V1.0
Project duration:	4 years	State:	Final
		Distribution:	public

NeOn Consortium

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

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

Change Log

Version	Date	Amended by	Changes
0.1	25-06-2008	Mathieu d'Aquin	Set up original document
0.2	10-09-2008	Mathieu d'Aquin	Extended Structure
0.3	10-10-2008	Chan LeDuc	Chapter 4 - Reasoning
0.4	15-10-2008	Mathieu d'Aquin	Move and summarize the content of Chapter 4
0.5	24-10-2008	Mathieu d'Aquin	Chapter 4 - Modularizing
0.6	28-10-2008	Mathieu d'Aquin	Chapter 5 - Extracting
0.7	05-11-2008	Mathieu d'Aquin	Abstract and Introduction
0.8	06-11-2008	Peter Haase	Module Specification, Composition, Extraction
0.9	10-11-2008	Mathieu d'Aquin	Methodological support (chap 6), Conclusion, Executive Summary, Partitioning experiments
1.0	05-12-2008	Mathieu d'Aquin and Peter Haase	Corrections

Executive Summary

Ontology modularization has attracted more and more attention in the recent years, as there is a clear need for such approaches to facilitate the management, evolution and distribution of large, complex ontologies. However, while there have been many papers in the literature investigating different aspects of modularization, there is still a need for a complete environment allowing ontology engineers to create, edit, manipulate, extract, decompose and combine ontology modules.

The work presented here follows the formalism described in [dHR+08], which provided the foundation of the NeOn support for ontology modularization. We describe the implementation and validation of a set of tools integrated within the NeOn Toolkit for ontology engineers to tackle the various tasks related to ontology modules, modular ontologies and ontology modularization in a controlled and interactive way.

We can distinguish two main activities related to ontology modularization: specifying modules at design time or modularizing existing non-modular ontologies. A first set of plugins consider the first aspect. It is possible within the NeOn Toolkit to specify ontology modules, editing interfaces and imported modules. In addition, this plugin provides the API at the basis of all the other plugins for modularization support in the NeOn Toolkit. An extension of the OntoModel plugin is also provided that allows ontology engineers to create multiple, modular diagrams for ontologies. Finally, another plugin provides various, simple operators (union, intersection, difference) to combine modules with each other.

Since many ontologies are built without modularity in mind, there is a need for tools to support the creation of modules from existing ontologies. For this purpose, we developed an ontology partitioning technique integrated in the NeOn Toolkit. This plugin decomposes an ontology into a set of modules, which are organized according to their dependency. In other applications, it is preferable to extract one focused and specified module rather than decomposing the ontology. We propose an interactive plugin for creating and refining modules extracted from ontologies, according to user-specified criteria. We also implement a formal technique to extract logically sound and complete modules for particular entities. Our experiments have shown that this technique can be used efficiently to improve the performance of reasoning tasks such as justification.

In a nutshell, we provide a complete set of tools integrated in the ontology engineering environment of the NeOn Toolkit and that can work together in supporting the complex tasks related to ontology modularization. We also started to investigate aspects beyond ontology engineering tools, like reasoning approaches for modular ontologies and methodological guidelines for modularizing ontologies.

Contents

1 Introduction **8**
1.1 Overview of the NeOn Toolkit Support for Ontology Modularization 8

I Designing Modular Ontologies **11**

2 Specifying Ontology Modules **12**
2.1 The NeOn Metamodel for Modular Ontologies 12
2.1.1 Abstract Definition and Notation 12
2.1.2 Metamodel 13
2.1.3 Concrete Representation with OMV 14
2.2 The NeOn plugin Support for Specifying Modules 15
2.3 Example Usage: Specifying the Fishery Ontologies Modules 16

3 Combining Modules **20**
3.1 Ontology Combination Operators 20
3.1.1 Existing Ontology Algebras 20
3.1.2 Supported Operators 21
3.2 The NeOn Toolkit Support for Combining Modules 22
3.3 Example Usage 22

II Creating Modules from Non-Nodular Ontologies **23**

4 Modularizing Ontologies **24**
4.1 Ontology Partitioning Techniques 24
4.2 A Dependency-Based Ontology Partitioning Algorithm 25
4.2.1 Definition of the Approach 25
4.2.2 Partitioning Algorithm 26
4.3 Implementation of the NeOn Toolkit Support for Modularizing Ontologies 28
4.4 Experiments on Applying the Technique 29

5 Extracting Modules From Ontologies **31**
5.1 Ontology Module Extraction Techniques 32
5.2 A Plugin for Interactive Module Extraction based on Multiple Operators 32
5.3 A Plugin for Locality-based Modules 33
5.4 Example Uses and Evaluation 35

III Discussion **38**

6 Other Aspects of Modularization **39**
6.1 Methodological Guidelines for Modularization 39
6.2 Reasoning with Modules 39
6.2.1 Reasoning with IDDL Modules 40
6.2.2 Implementation 41

7 Conclusion **42**

Bibliography **43**

List of Figures

1.1 Overview of the modularization plugins for the NeOn Toolkit. 9

2.1 Metamodel extensions for ontology modules. 14
2.2 Overview of the OMV extension for Ontology Modules. 15
2.3 Specifying the imports relationship between modules 17
2.4 Specifying the interface 18
2.5 Diagrams for Ontology Modules in Ontomodel 19

3.1 Combining modules. 22

4.1 Enforcing good properties for the dependency structure of modules. 26
4.2 The ontology partitioning plugin for the NeOn Toolkit. 28
4.3 Execution time of the algorithm depending on the size of the original ontology. 30

5.1 Screenshot of the ontology module extraction plugin. 33
5.2 The time performance of three algorithms for finding all justifications. 37

6.1 Workflow for the task of modularizing an existing ontology. This figure summarizes the activities realized as part of this task, as they will be described in deliverable 5.4.2. 40

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+03, 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).

From these research studies, it can already be seen that there are several tasks one may consider when having to cope with ontology modularization. An obvious distinction concerns, on the one hand, approaches that tackle the design of modular ontologies (modularization *a priori*) and, on the other hand, approaches that intend to introduce modularity in ontologies that have not been designed in a modular way (modularization *a posteriori*).

In [dHR+08] we defined the formal foundation of the NeOn support for modularization with the aim of supporting both these views. A basic module definition language was defined, based on the notions of mapping, partial import and encapsulation, and a set of operators were identified as potentially useful either to combine ontology modules or to create modules from existing ontologies.

In the following, we present the implementation of a set of plugins for the NeOn Toolkit to support the activities related to ontology modularization. In a first part, we present developed tools to support *a priori* modularization, including specifying ontology modules (Chapter 2) and combining modules with each other (Chapter 3). In a second part, tool support for *a posteriori* modularization is described, including plugins for partitioning existing ontologies into a set of modules (Chapter 4) and for extracting modules from ontologies according to some user-defined criteria (Chapter 5). Finally, we briefly summarize current activities to support other tasks related to ontology modularization within NeOn, which will be treated in other deliverables (Chapter 6).

However, before entering into the details of the implementation of each individual tool, the next section provides an overview of this set of plugins and of their organization/interaction.

1.1 Overview of the NeOn Toolkit Support for Ontology Modularization

Figure 1.1 provides a general overview of the different plugins and software components involved in the NeOn toolkit support for ontology modularization. The central component in this diagram is the NeOn Module

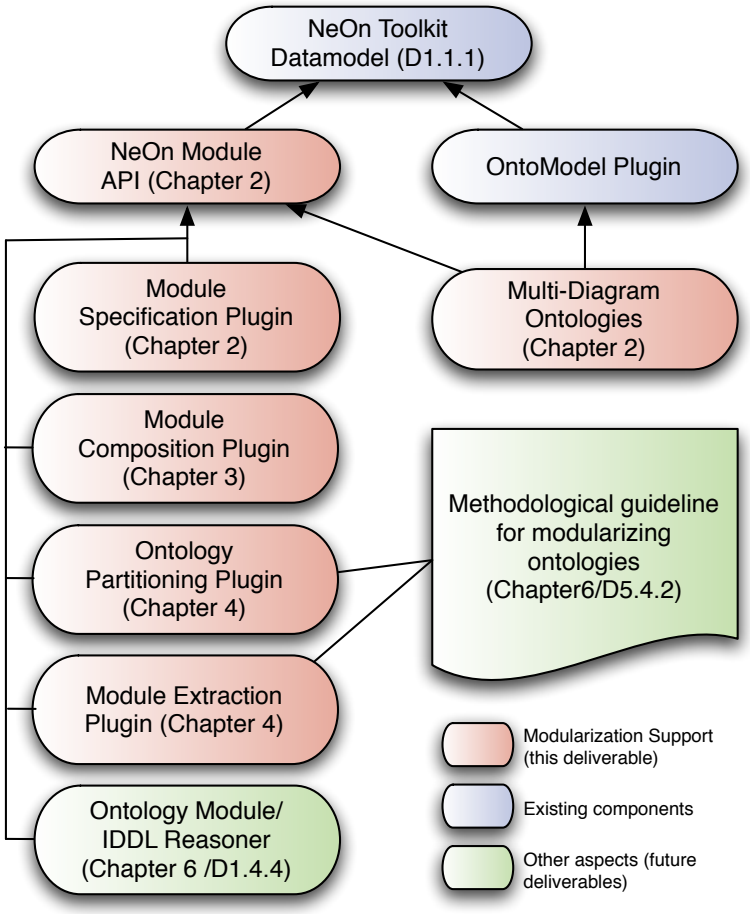

Figure 1.1: Overview of the modularization plugins for the NeOn Toolkit.

API, which provides the basic structures for creating a manipulating modules programmatically within the environment of the NeOn Toolkit. This API extends the existing NeOn Toolkit datamodel for ontologies and is associated with the features included in the NeOn Toolkit for specifying modules. As part of these particular set of features, an extension of the OntoModel plugin is also provided to describe multiple diagram for ontologies, allowing a simple definition a modular design for ontologies.

All the other plugins related to modularization are based on the module API. These include the plugin for specifying ontology modules, which allows the user to describe a module in terms of imports and interfaces, composing ontology modules, which allows the user to put together ontology modules to create new ontologies, the plugin for partitioning ontologies, which allows the user to decompose an ontology into modules, and the plugin for extracting modules from ontologies, which allows the user to extract from an ontology a part that satisfies certain given criteria.

Finally, as mentioned above, some aspects related to the support for ontology modularization will be fully treated in future deliverables. This includes the integration of a reasoner for ontology module based on the semantics of IDDL, as described in [dHR$^+$08], and the definition of methodological guidelines for various aspects of ontology modularization, starting from the task of modularizing existing ontologies.

Part I

Designing Modular Ontologies

Chapter 2

Specifying Ontology Modules

In this chapter we describe how the specification of modules according to the NeOn metamodel for modular ontologies is supported within the NeOn infrastructure. At the basis, we have realized a *core module plugin* that provides a *module API*, which is used by other plugins providing modularization support for the programmatic specification and manipulation of modules. Further, we provide GUI-level plugins for the specification of modules by an ontology engineer within the editor.

2.1 The NeOn Metamodel for Modular Ontologies

In this section we briefly recapitulate the basics of NeOn model for modular ontologies as defined in Deliverable D1.1.3 [dHR+08].

2.1.1 Abstract Definition and Notation

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 $\text{Sig}(O)$ denote the set of named elements occurring in O, so it represents those elements the axioms from O deal with.

[1] In most cases – and in particular for OWL – $L(P)$ will be infinite, even if P is finite.

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*

- $id \in Id_{\text{Modules}}$ *is the identifier of* $\mathcal{OM}$
- $Imp \subseteq Id_{\text{Modules}}$ *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* $\mathcal{OM}$*),*
- $M \subseteq Id_{\text{Mappings}}$ *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),*
- $E \subseteq \text{Sig}(O) \cup \bigcup_{id \in Imp} \mathcal{I}_{id}$ *is called* EXPORT INTERFACE *(telling which named entities from the ontology module are "published", i.e., can be imported by other ontology modules).*

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*

- $s, t \in Id_{\text{Modules}}$*, 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]

2.1.2 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 2.1 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.

[2] In accordance with the NeOn metamodel, this set will be fixed to $R = \{\sqsubseteq, \sqsupseteq, \equiv, \perp, \not\sqsubseteq, \not\sqsupseteq, \not\equiv, \not\perp\}$

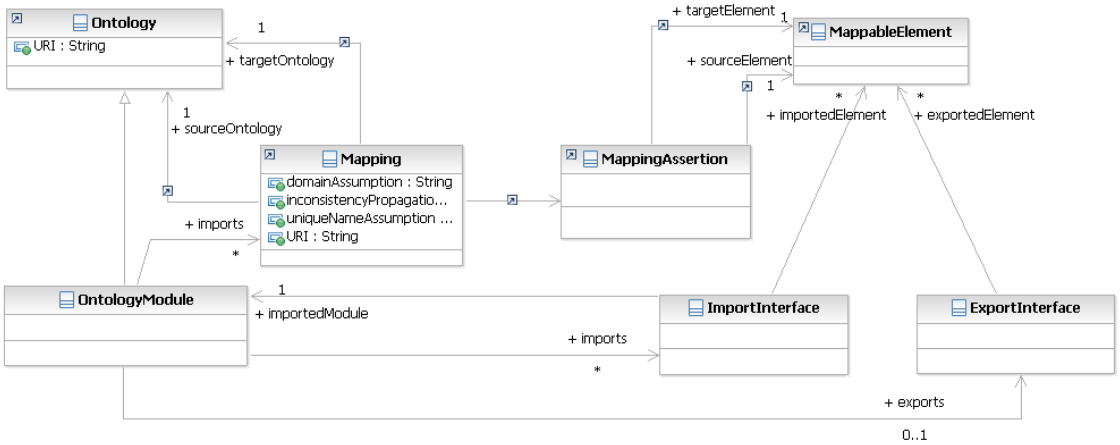

Figure 2.1: Metamodel extensions for ontology modules.

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.

2.1.3 Concrete Representation with OMV

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+05]. 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 2.2 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.

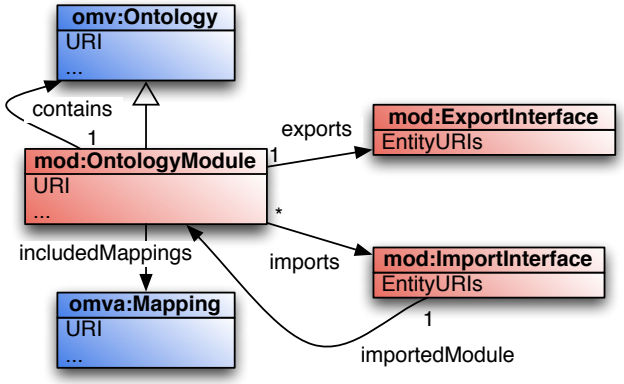

Figure 2.2: Overview of the OMV extension for Ontology Modules.

2.2 The NeOn plugin Support for Specifying Modules

As the core plugin, we have realized an API for the specification and manipulation of modules. This API is generated directly from the metamodel for modular ontologies. Essentially, every class of the metamodel is represented as a Java class in the API. The ontology class (from which the module class inherits), is connected via the delegator pattern with the ontology class of the datamodel API of the core NeOn Toolkit.

In the following, we exemplarily show the interface for the module class in the API:

```
public interface OntologyModule extends Ontology {

   public void addImportIterface(ImportInterface importInterface);
   public void removeImportIterface(ImportInterface importInterface);
   public Set<ImportInterface> getImportIterfaces();

   public void setExportInterface(ExportInterface exportInterface);
   public ExportInterface getExportInterface();

   public void addMapping(Mapping mapping);
   public void removeMapping(Mapping mapping);
   public Set<Mapping> getMappings();

   public String getModuleUri();
   public void setModuleUri(String uri);
   public void saveOntologyModule(String modulePath);
}
```

We see that the `OntologyModule` class is realized as an extension of the `Ontology` class (i.e., as specified in the metamodel, it inherits from the `Ontology` class). Further, we see the getter/setter methods to for the attributes and relationships of the `OntologyModule` class.

The plugin also provides a default implementation. In this implementation, the module information is persisted using the OMV-based syntax described in Section 2.1.3.

The API plugin is intended to be used by other plugins providing modularization support. For the specification of plugins within the editor of the NeOn Toolkit, we have developed a GUI plugin extending the editor with module support. For the specification of the import relationship between modules, we rely on the existing mechanisms in the NeOn Toolkit to import an ontology into another (c.f. Imports Section in Figure 2.3). As explained before, the intuition is to reuse the existing modeling constructs on the ontology level, by treating a

module as (specialized) ontology.

For specifying the interfaces, an extension to the entity properties page named *Module Interface* (c.f. Figure 2.4 has been realized. In this page it is now possible, to define the import and export interfaces: For the module itself, it is possible to specify the signature by selecting the respective classes and properties to be exported. Similarly, for all modules it is possible to select the subset of classes and properties that should be imported by the module.

While this module specification plugin aims at the specification of the module *metadata*, we have also extended existing plugins to support modular ontology design on the *content* side.. As an example we here mention the OntoModel plugin, a plugin for the visual (UML-based) modeling of ontologies. The basic idea is that in OntoModel, the ontology can be specified in UML-like diagrams. A drawback of the existing implementation was that there was a one-to-one correspondence between ontology and diagram, i.e. the entire ontology always had to be shown at once in the diagram. Obviously, this approach did not scale for modelling large ontologies. This, support for modularization was needed. We extended OntoModel in two ways:

- We introduced multiple-diagram modeling, i.e. a one-to-many relationship between model and diagram. Each diagram can show a subset (a module) of the overall model. (c.f. Figure 2.5
- We introduced module extraction: Starting from a diagram, the user can select a subset of the ontology, based on which a new module should be extracted. Extraction techniques (c.f. Chapter 5) have been implemented to be able to identify and select the parts of the ontology that are relevant.

More details about the extensions of OntoModel to support modularization can be found in [Nab08] and on the plugin website at `http://www.neon-toolkit.org/wiki/index.php/OntoModel`.

2.3 Example Usage: Specifying the Fishery Ontologies Modules

We will now describe the usage of the plugins using a small example with ontologies from the FAO case study.

In this example, we start out with two initially isolated ontologies that are to be related in a modular way. The ontologies are the *Gear ontology* and the *Vessels ontology*, as described in Deliverable D7.2.2 [CG07]. The idea is to import the gears ontology is imported as a module into the vessels ontology, specifying the relevant interfaces.

Figure 2.3 shows how the import relationship between the two ontologies is specified: In the *Imports* section we see that the vessels module imports the gears module.

In Figure 2.4 we see how the interfaces are specified in the extension to the entity properties page named *Module Interface*. In the upper part we see the definition of the exports interface, where the elements `hasVessClassGRT, hasVessClassPower, hasName, hasID` are selected to be exported. Similarly, we see how the all imported modules it is possible to select the subset of classes and properties that should be imported by the module. In the Figure 2.4, the engineer has specified that the properties `hasID, hasName, hasDescription` should be imported from the gears modules.

The corresponding serialization of the module based on OMV looks like this:

```
<rdf:RDF
    xml:base="http://modules.ontoware.org/newModule1226068707828.owl"
    xmlns:a="http://modules.ontoware.org/2008/10/ontology#"
    xmlns:b="http://omv.ontoware.org/2005/05/ontology#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<owl:Ontology rdf:about=""/>
```

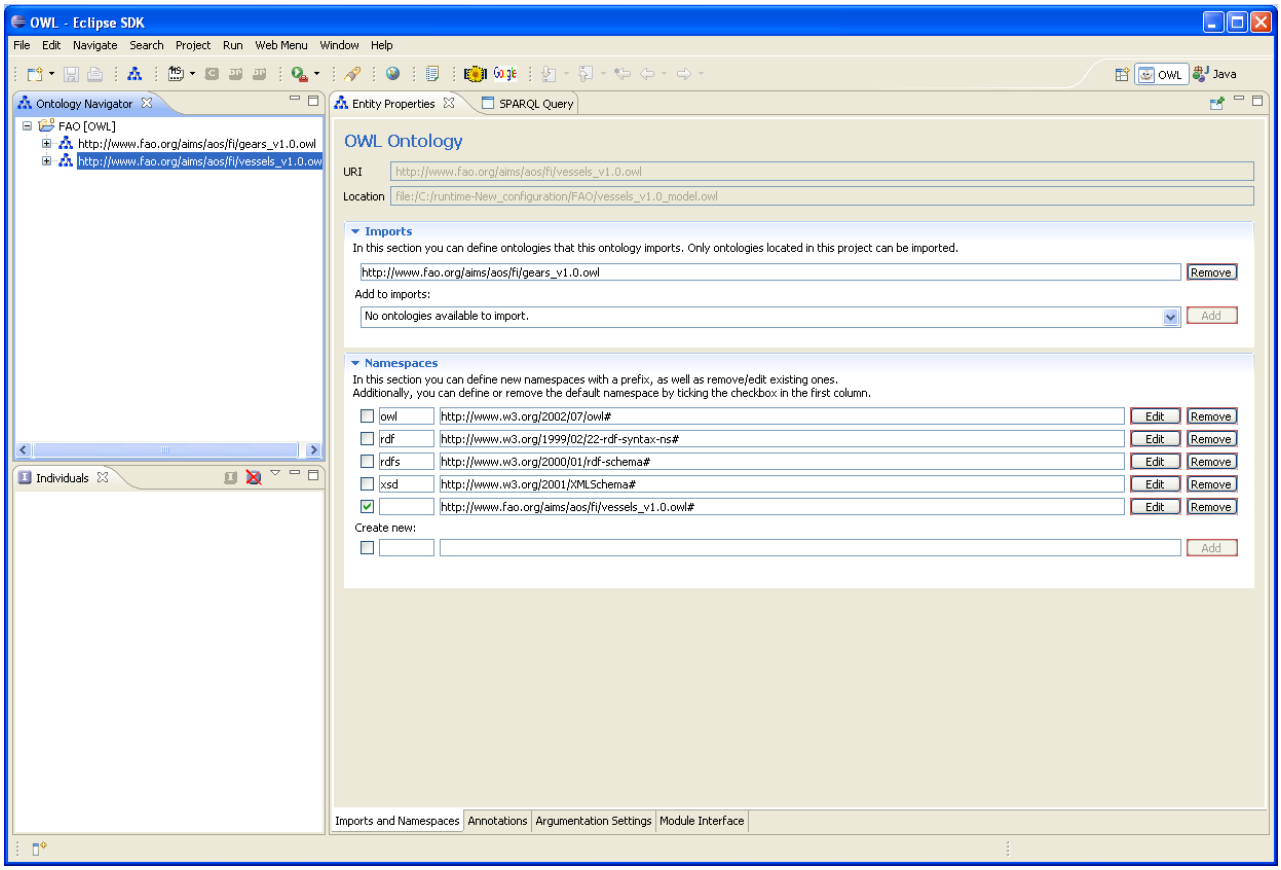

Figure 2.3: Specifying the imports relationship between modules

```
<a:OntologyModule rdf:ID="ex_module">
    <a:contains rdf:resource="&c;vessels_v1.0.owl"/>
    <a:exports rdf:resource="#exportInterfaceIndi"/>
    <a:imports rdf:resource="#importInterfaceIndi0"/>
</a:OntologyModule>

<a:ExportInterface rdf:ID="exportInterfaceIndi">
    <a:hasElement rdf:resource="&d;hasID"/>
    <a:hasElement rdf:resource="&d;hasName"/>
    <a:hasElement rdf:resource="&d;hasVessClassGRT"/>
    <a:hasElement rdf:resource="&d;hasVessClassPower"/>
</a:ExportInterface>

<a:ImportInterface rdf:ID="importInterfaceIndi0">
    <a:hasElement rdf:resource="&e;hasDescEN"/>
    <a:hasElement rdf:resource="&e;hasID"/>
    <a:hasElement rdf:resource="&e;hasName"/>
    <a:hasImportedModule rdf:resource="&c;gears_v1.0.owl"/>
</a:ImportInterface>
...
</rdf:RDF>
```

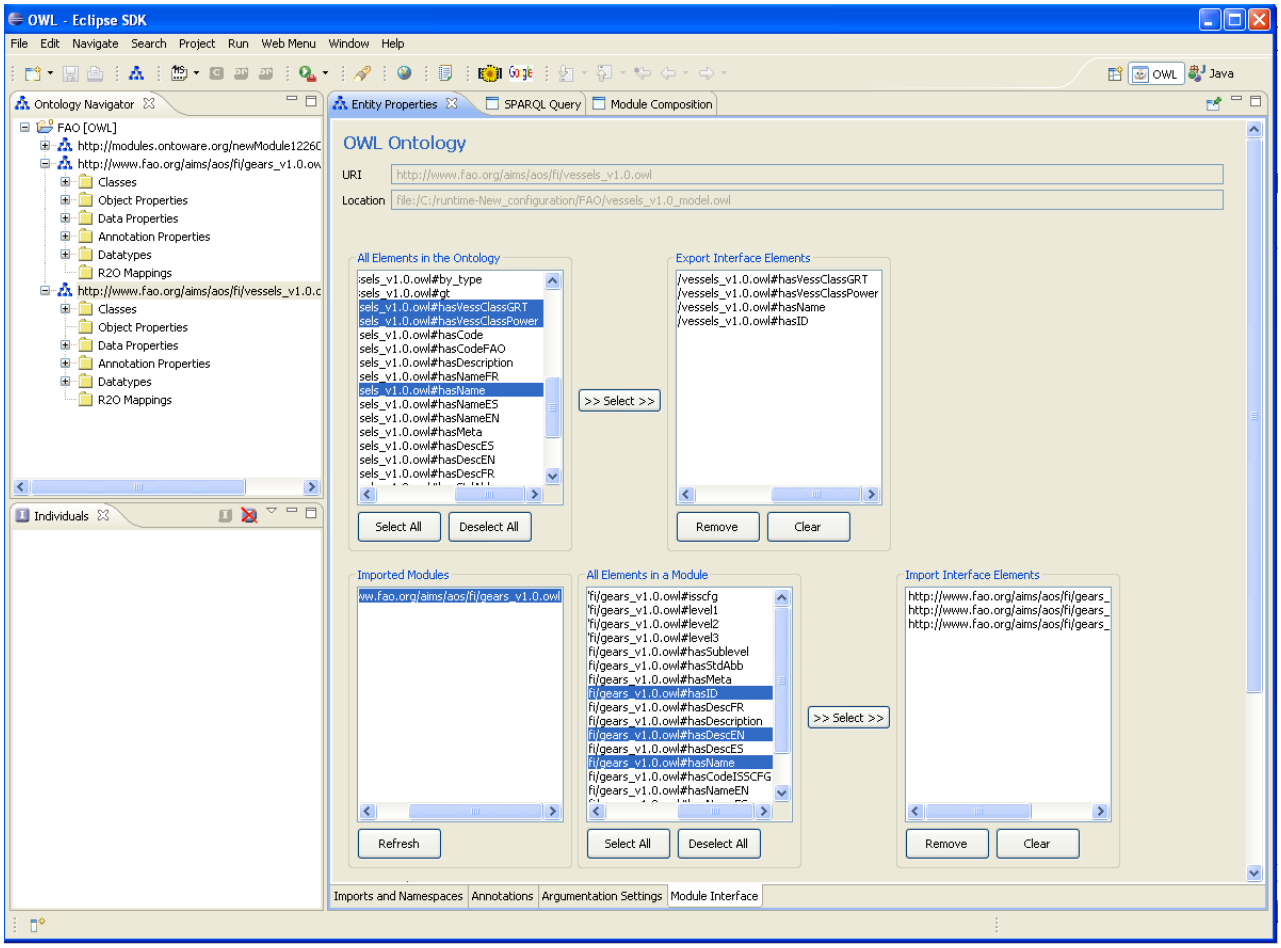

Figure 2.4: Specifying the interface

Figure 2.5 shows (one aspect of) the modularization support within OntoModel. We use a large ontology (species ontology) with a thousands of individuals. This ontology is way too large to be visualized within a single diagram. In the figure we see a diagram that only shows a module (fragment) of the original ontology.

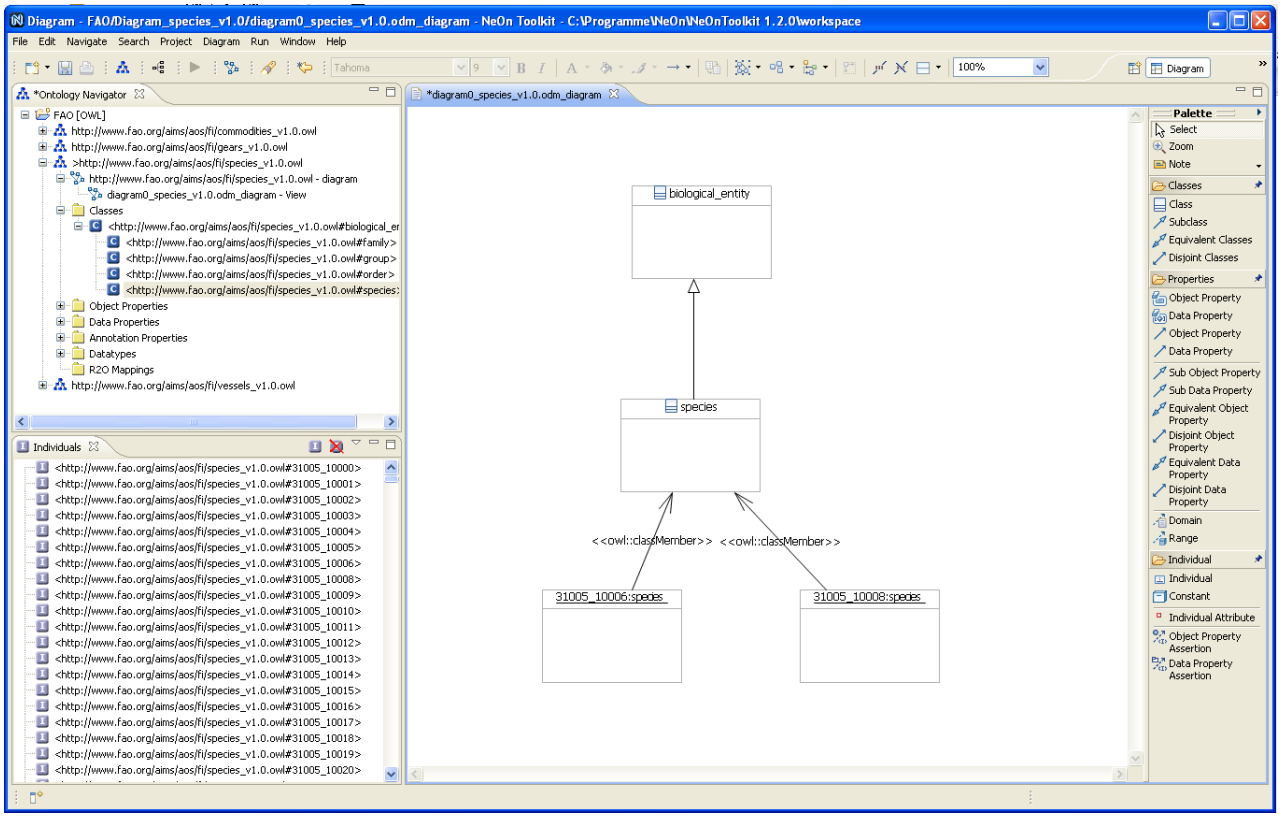

Figure 2.5: Diagrams for Ontology Modules in Ontomodel

Chapter 3

Combining Modules

In this chapter, we describe the support for the combination of modules. The work builds on a module algebra with a set of ontology combination operators. This algebra was introduced in [dHR+08] and is briefly recapitulated in the following section. Subsequently, we describe how the algebra has been realized as a plugin to the NeOn Toolkit.

3.1 Ontology Combination Operators

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.

3.1.1 Existing Ontology Algebras

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.

3.1.2 Supported Operators

Inspired by the work described above and relying on the metamodel for ontology modules in NeOn, we defined the three following module combination operators to be implemented in the plugin.

Union 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.

Definition A simple way to comply with the semantics of the union operator is that the created module includes all the axioms in the two combined modules.

Difference 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, $Difference(M, M) = empty\_module$

Definition This operator can be approximated by applying set differences to the axioms of the combined modules.

Intersection 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.

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 axioms of the modules.

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

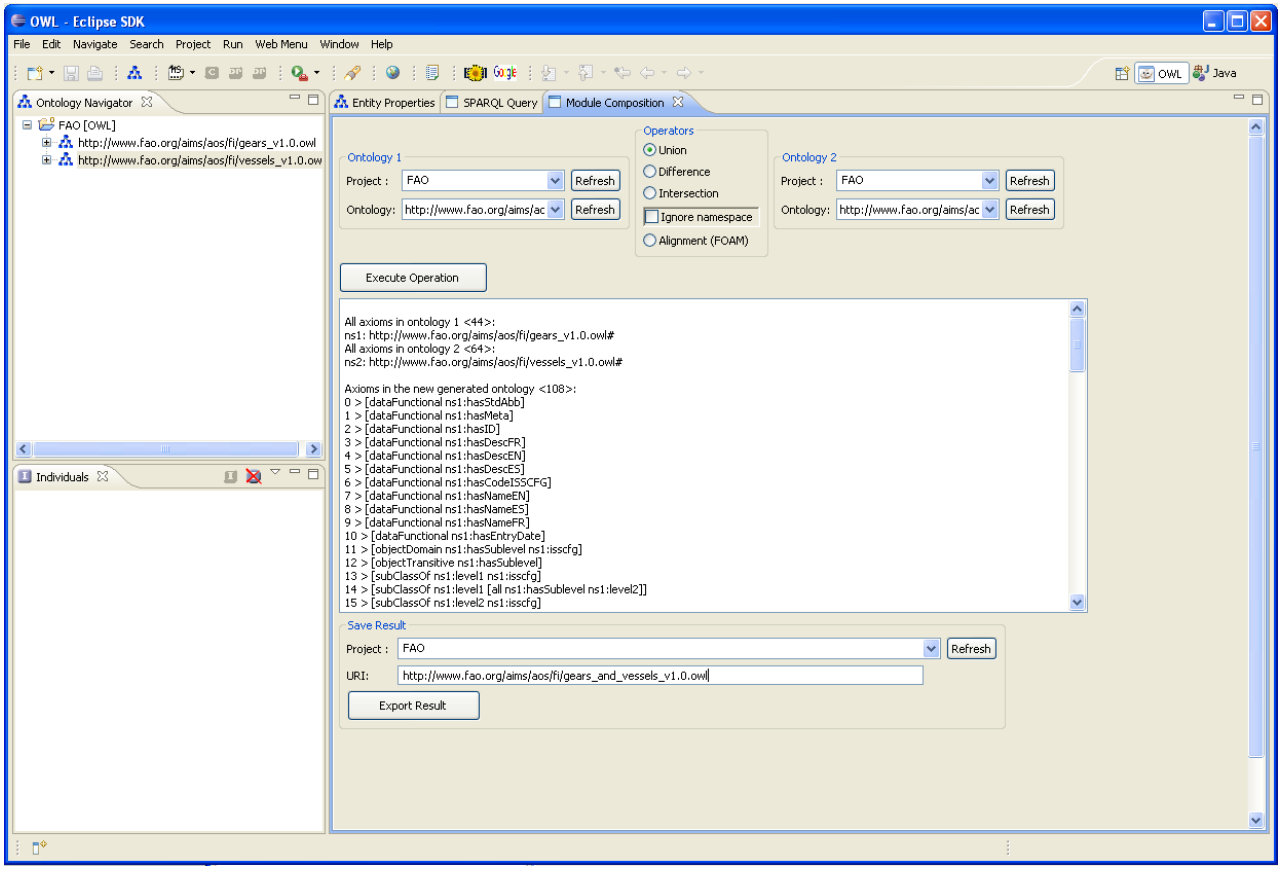

Figure 3.1: Combining modules.

3.2 The NeOn Toolkit Support for Combining Modules

The module algebra is implemented in a dedicated plugin, which is realized as a new NeOn Toolkit view. As shown in Figure 3.1, in this view the user selects the two ontologies that serve as input for the operators. In the field between the two ontologies the user selects the operator to be applied. In addition to the combination operators mentioned above, the plugin also supports alignment as an operator, which allows to relate modules via mappings. Depending on the operator chosen, the result will be either a new module (for union, diff, intersection), or an alignment (for align).

Finally, the user can specify whether the application of the operators should be sensitive to differences in the namespace. If not, the operators only consider local names. This is for example relevant for the difference operator applied to two versions of the same ontology – as often the namespace changes from one version to another (and thus all elements in the ontology), a difference based on the fully qualified names would not be very meaningful.

The result of applying the operators can be saved to an ontology project.

3.3 Example Usage

Figure 3.1 shows the application of the operators to ontologies from the FAO case study. As input, the vessels module and the gears module are selected, the modules are combined via the union operator to a single module. Please note the difference with the previous example in Section 2.3, where a combination was specified via an imports relationship between the two modules. In contrast, the union operator generates a single module.

Part II

Creating Modules from Non-Nodular Ontologies

Chapter 4

Modularizing Ontologies

Modularizing ontologies refers to the process of manually, automatically or semi-automatically creating a set of modules from an existing, non-modular and potentially large ontology. It is often referred to as *ontology partitioning*[1], since they create a set of modules, each containing a sub-set of the axioms and entities of the original ontology, and such that the union of these modules corresponds to the original ontology.

In [dHR+08], we proposed the definition of a *decompose* operator for modules as follows:

Description This operator divides an existing module into parts that should correspond to significant components.

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

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

Properties in some definitions, 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.

In this chapter, after a quick reminder about existing techniques for automatic partitioning of ontologies, we propose a novel algorithm that has the particularity of taking into account the dependency between modules. It results in modularization with dependency structures having good properties from a knowledge engineering perspective. We then detail the implementation of this operator as a NeOn Toolkit plugin, before presenting the experiments realized for the validation of the approach.

4.1 Ontology Partitioning Techniques

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.

[1] Note that some approaches being labeled as *partitioning* methods do not actually create *partitions*, as the resulting modules may overlap.

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.

4.2 A Dependency-Based Ontology Partitioning Algorithm

It is very complicated to evaluate partitioning techniques. Most of the techniques result in a bag of modules, that would need to be inspected by experts of the domain and ontology engineers to check if the result match what was expected. In realistic cases, defining what to expect and checking it is not even feasible. Indeed, there is only a handful of measures one can consider to characterize a bag of modules, being themselves bags of axioms [dSSS07]. Moreover, not considering the properties of the *structure* of the modularization– i.e., how modules relate to and depend on each other–hampers the usability of the modularization, in particular for the purpose of facilitating the maintenance of an ontology. For this reason, we introduce a new algorithm for partitioning ontologies which is primary based on enforcing good properties on the dependency structure of the resulting modularization.

4.2.1 Definition of the Approach

Our approach to ontology partitioning is based on basic requirements concerning the resulting modularization and its structure. We consider that the result of the partitioning process should not only be a bag of modules, but should also provide the relations between them in terms of dependency. In addition, some good properties for this structure should be enforced, in order to facilitate the manipulation and maintenance of the modularization.

As our approach is based on the dependency structure of modules, we need to define this relation of dependency. We consider a module m_1 to be dependent on a module m_2 if there is at least one entity in m_1 which definition or description depends on at least one entity in m_2. The definition or the description of an entity A depends on an entity B whenever B participates to the axioms defining or describing A, e.g., in the following axioms

$$A \equiv B$$
$$A \sqsubseteq B$$
$$A \equiv \exists p.B$$
$$A \sqsubseteq \forall p.B$$
$$etc.$$

A is dependent on B and p.

From this definition, we can see that if a module m_1 depends on a module m_2, it means that m_1 should import m_2. Therefore, using this notion of dependency, our approach provides a complete modular structure, with for each resulting module, information about the necessary imports and interfaces. The result is indeed a graph of modules based on this dependency relation.

Another particularity of our approach is that, not only we want to provide a dependency structure for the resulting set of modules, but we also want this structure to have good properties in order to be *efficient* in facilitating further engineering of the obtained modular ontology. In other terms, as shown in Figure 4.1, we do not want this structure to be any arbitrary (directed) graph, but to respect 2 major rules:

Rule 1 (no cycle): There should not be any cycle in the dependency graph of the resulting modularization. This means that, if we note $m_1 \rightarrow m_2$ the dependency relation between two modules m_1 and m_2, and $m_1 \xrightarrow{*} m_2$ the transitive closure of the dependency relation, there should not exist any module m_1

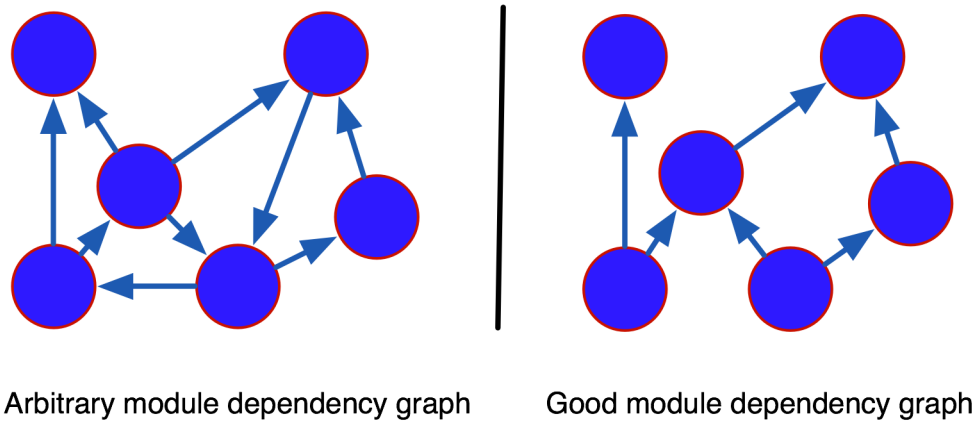

Figure 4.1: Enforcing good properties for the dependency structure of modules.

such that $m_1 \xrightarrow{*} m_1$. The rational for this rule is that we are trying to reproduce the natural situation where modules would be reused. Creating bidirectional interdependencies between reused modules is a bad practice as it introduces additional difficulties in case of an update of one of the modules or when distributing modules [**?**].

Rule 2 (no transitive dependency): If a module reuse another one, it should not, directly or indirectly reuse a module on which the reused one is dependent. More formaly, there should not be any modules m_1, m_2 and m_3 such that $m_1 \rightarrow m_2$, $m_2 \xrightarrow{*} m_3$ and $m_1 \rightarrow m_3$. Indeed, when this situation arises, it means that the organization of modules into *layers* have not been enforced, so that a module is reusing other module at different levels of the same branch of the dependency graph. Besides producing unnecessary redundancies in the dependency structure, this could also cause difficulties for the evolution and distribution of the module by creating "concurrent propagation paths" leading to the same module.

In addition, in order to ensure not only that the structure of the modularization respects good properties, but also that individual modules are easy to manage and to handle, we add two rules on the characteristics of each module:

Rule 3 (size of the modules): A module should not be smaller than a given threshold. Indeed, Initial experiments have shown that applying only the two rules above can result in very small modules. Too small modules can be hard to manage, as it can result in having to consider too many different modules for a given task (e.g., update) [dSSS07]. Note that, even if it could sometimes be useful, a rule based on the maximum size of a module would not be applicable as it would contradict rules 1 or 2. In this case, it would be recommended to use the extraction techniques described in the next chapter to reduce the size of the modules considered too big.

Rule 4 (intra-connectedness): Entities within a module should be connected with each other. This is a very simple and natural rule to follow. Indeed, there is no reason for entities that are completely disconnected, directly or indirectly, to end-up in the same module.

4.2.2 Partitioning Algorithm

Having the above rules defined, our algorithm for partitioning ontologies is reasonably straightforward. It basically consists in starting from an initial modularization with as many modules as entities in the ontology. From this initial modularization, the algorithm iteratively enforces Rules 1 and 2, merging modules when

necessary. At the end of this step a modularization that respects Rules 1, 2 and 4 is obtained. The last task consists in merging modules that are too small according to the given threshold, ensuring that this merging ends up in modules that respect Rule 4.

In the following algorithm:

- a module contains a set of entities, and all the axioms attached to these entities
- $|m|$ with m a module, refers to the size of the module in number of entities (note that we consider an ontology and an ontology module) to contain individuals, as well as classes and properties)
- merging two modules consists in creating a new module, containing the union of the entities of the original modules and importing all the modules imported by them
- similarity between modules is computed by comparing the number of imported modules and the size of the modules.

Algorithm: Dependency-based ontology partitioning
Input: Ontology O, Integer ts (size threshold)
Returns: Set of Ontology Modules (with import information)

1. Obtain the list le of entities of O
2. For each entity e in le
   (a) create a new module m containing e
   (b) find the entities e depends on and add the corresponding modules to the imported modules of m
   (c) add m to the list of modules lm
3. For all the modules m in lm such that $m \xrightarrow{*} m$
   (a) merge all the modules on the path from m to m into a module mm
   (b) remove all the modules on the path from m to m from lm
   (c) add mm to lm
4. For all the modules m_1, m_2 and m_3 in lm such that $m_1 \rightarrow m_2$, $m_2 \xrightarrow{*} m_3$ and $m_1 \rightarrow m_3$
   (a) merge all the modules on the path from m_2 to m_3 into a module mm
   (b) remove all the modules on the path from m_2 to m_3 from lm
   (c) add mm to lm
5. For all the modules m in lm such that $|m| < ts$
   (a) merge m with the module m_2 that is the most similar with m amongst the set of modules imported by m or importing m, into a module mm
   (b) remove m and m_2 from lm
   (c) add mm to lm

   return lm

The result of the algorithm is a list of modules, each containing a sub-set of the entities of the original ontology (without overlap) and potentially importing other modules of the modularization. It can easily be shown that this modularization respect Rules 1, 2 and 3. Rule 4 is enforced simply because, anytime in the algorithm two modules are merged, there is a dependency relation between them. This dependency relation is derived from relations between entities. Therefore, the modules resulting from the merging necessarily contains entities that are related with each other.

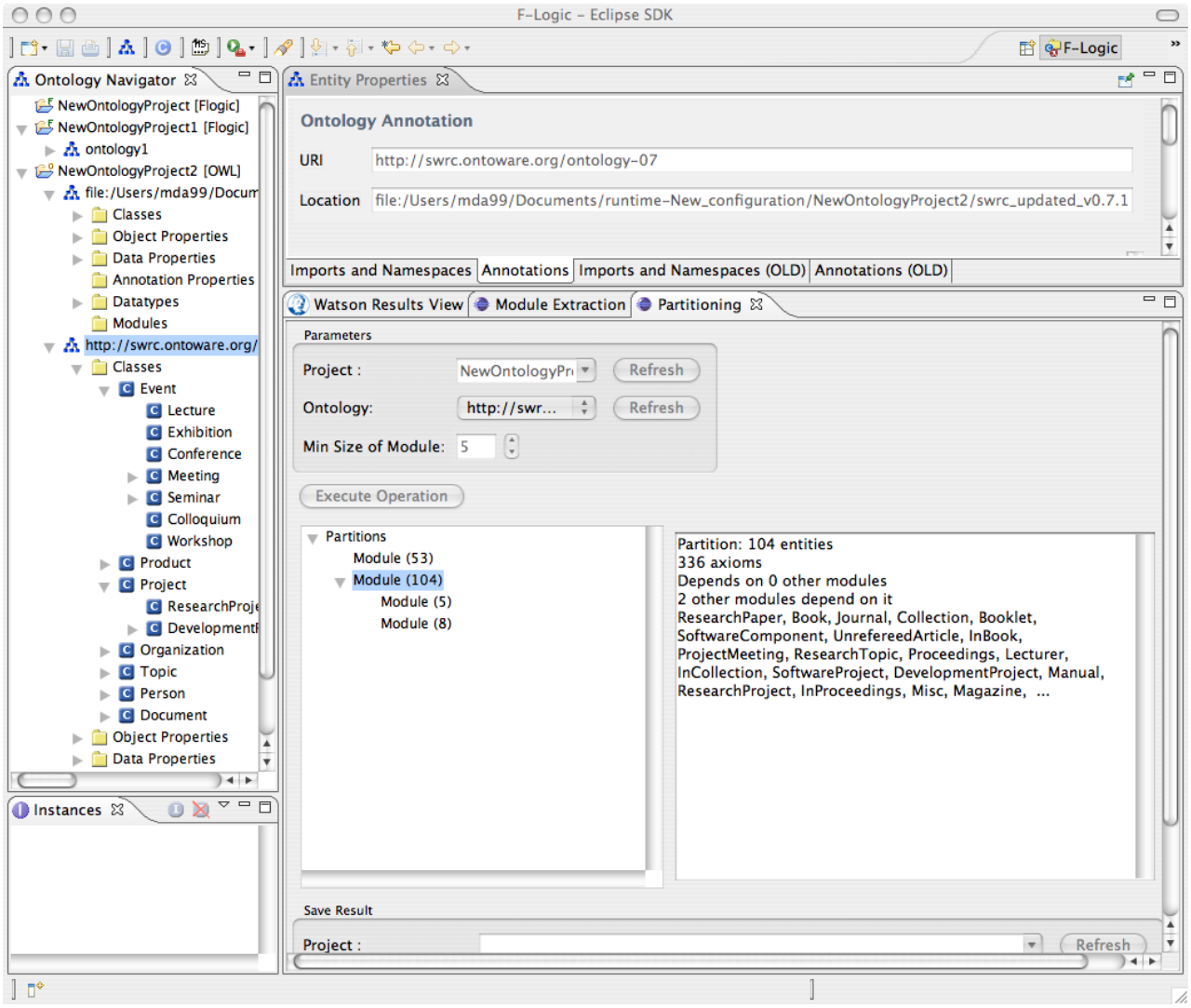

Figure 4.2: The ontology partitioning plugin for the NeOn Toolkit.

4.3 Implementation of the NeOn Toolkit Support for Modularizing Ontologies

The algorithm presented in the previous section has been implemented as a plugin for the NeOn Toolkit. It relies on the *Module API* presented in Chapter 2, as it implements a modularization operator and produce ontology modules represented in terms of this API.

Concretely, this plugin takes the form of a view within the environment of the NeOn Toolkit, which allows the user to select the ontology to modularize, specify the threshold for the minimum size of the modules and execute the algorithm (see figure 4.2). The result of the algorithm is them presented as a tree, with each node corresponding to a created module (details of the module are shown when selecting the corresponding node). The plugin allows the user to save and integrate to the current ontology project each module individually.

An interesting aspect of the implementation within the NeOn Toolkit is that it allows a very flexible and customizable modularization process. Indeed, it is possible to re-run the algorithm with different parameters, save only the modules that are relevant according to the ontology engineer, and use the module composition plugin presented in Chapter 3 to manipulate and customize the modularization, until a satisfactory, well-suited modularization is obtained.

min_size	nb modules	time (ms)	min entities	max entitites	graph
2	11	27,481	2	200	3 levels
3	10	27,623	3	204	3 levels
4	8	27,451	5	207	3 levels
5	8	27,469	5	207	3 levels
7	6	26,707	7	212	2 levels
10	5	27,164	17	219	2 levels
20	3	27,333	181	254	2 levels

Table 4.1: Applying the dependency based partitioning technique with varying min_size values. $nb\,modules$ corresponds to the number of modules in the resulting partition, $time\ (ms)$ to the time taken for executing the partitioning algorithm, $min\ entities$ to the size of the smallest module, $max\ entities$ to the size of the biggest module, and $graph$ to the structure of the resulting dependency graph (its depth).

4.4 Experiments on Applying the Technique

In order to evaluate the behavior of our technique for partitioning ontologies and of its implementation within the NeOn Toolkit, we applied it on a set of real-life ontologies in order to measure the characteristics of the tool and of the resulting modularizations.

Our first test intended to check the basic behavior of the algorithm, to verify that the four properties we defined where all validated, and to assess some measures on the results, as well as the influence of the parameter (min_size: the minimum size of a module). We applied the tool on a single ontology[2], chosen for its reasonably large size (439 entities, 2044 axioms) and complexity, with varying min_size values. The results are summarized in table 4.1.

Testing Rules 1, 2, 3 and 4. On each of the resulting modularization, we checked wether or not the rules defined above were properly enforced by the algorithm. In general, it was fairly straightforward to verify that Rules 1 and 2 were always respected, the resulting graph being always a simple tree. However, we discovered a small issue concerning Rules 3 and 4: in this and some other ontologies, there is a small number of entities that are not connected to any other. For each of these entities, the original algorithm generated a module (of size 1), completely disconnected from the others. In order to simplify the results, we gathered these singleton modules into one unique module. This module obviously does not enforce Rule 4 (it may contain several entities disconnected from each other) and might not respect Rule 3 (it may be smaller than the given minimal size). However, for all the other modules, both Rules are enforced properly. Note that in table 4.1, the size of the smallest module is measured without counting this special module (which is very often the smallest one).

Influence of min_size. A useful information that can be derived from the results is that acting on the parameter min_size seems to be a reasonably efficient way to influence the characteristics of the resulting modularization. Indeed, while the size of the smaller module obviously increases with min_size (following it very closely in most of the cases), we can also observe that there is a clear influence of the parameter on the number of modules in the results (which clearly decreases as min_size increases). However, surprisingly, the time taken to execute the partitioning algorithm is relatively constant and does not seem to be influenced by the parameter.

[2] `http://www.inrooh.net/ontologies/arabicitontology.owl`

Ontology	nb entities	nb axioms	time (ms)
FAO's Gears ontology	92	956	211
FAO's Vessels ontology	148	1,702	1,127
Arabic Ontology[4]	439	2,044	27,469
Sequence ontology[5]	3,509	12,225	239,414
FAO's Commodities ontology	1,394	111,361	3,414,307

Table 4.2: Influence of the size of the ontology on the execution time of the partitioning algorithm. $nb\ entities$ corresponds to the size in number of classes, properties and individuals of the original ontology, $nb\ axioms$ to the size in number of axioms, and $time(ms)$ to the time taken to execute the partitioning algorithm.

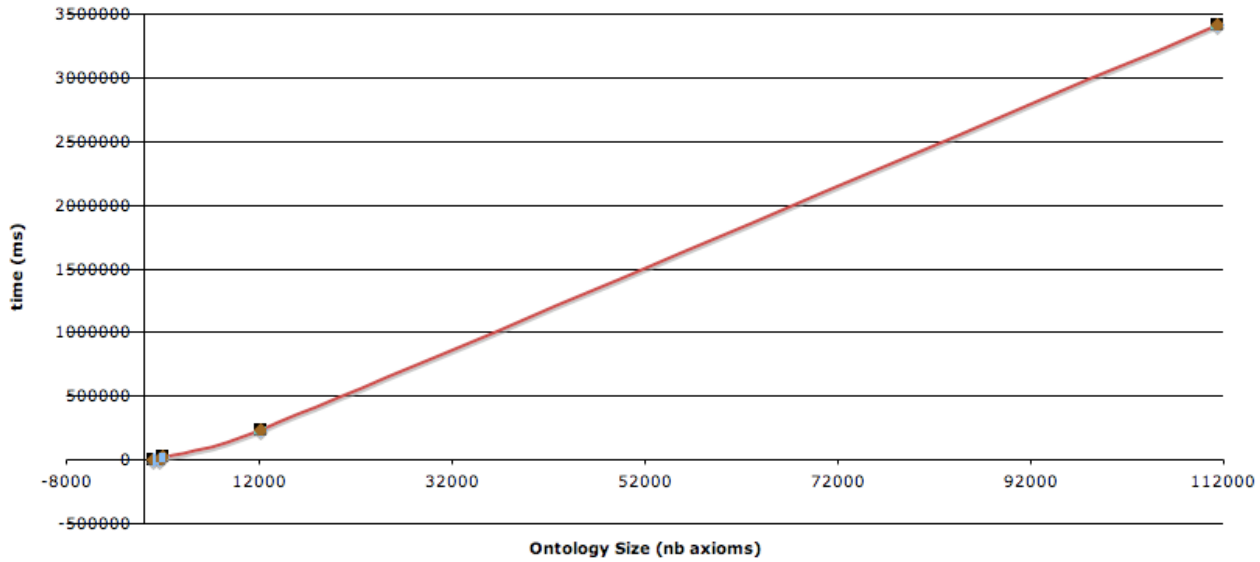

Figure 4.3: Execution time of the algorithm depending on the size of the original ontology.

Comparison with SWOOP. The SWOOP[3] ontology editor includes a feature for partitioning ontologies based on the ϵ-connection formal framework. We applied this feature on the same ontology as above. It is worth mentioning that SWOOP does not provide any way to act on the partitioning method, as there is no other input to the algorithm than the ontology. While the results are obtained faster with SWOOP (less than a second), they are actually quite disappointing as only one module is returned, which contains the same elements as the original ontology. It has been shown in [dSSS07] that this result is quite common and was also obtained with several other ontologies.

Response time. One important element in this evaluation concerns the execution time of the technique. We already established above that this time does not seem to be influenced by the min_size parameter. The other parameter that could influence the performance of the plugin is the ontology, or more precisely, its size. We ran our technique on five different ontologies with varying sizes (from a quite small one, to a very large one) with a min_size parameter at 5 (the default value). The results are summarized in Table 4.2. It can be easily observed from these results that the time taken to execute the algorithm varies linearly with the size of the ontology (in number of axioms, see Figure 4.3).

[3] `http://code.google.com/p/swoop/`

Chapter 5

Extracting Modules From Ontologies

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 ($Sig(M_{SV}) \supseteq SV$).

In [dHR+08], we proposed the definition of a *Reduce* operator for modules as follows:

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 A number of use cases explicitly rely on the feature provided by the reduce operator. In particular, this operation is often considered as a possible way to improve performance of reasoning with the ontology, by allowing the reasoner to focus only on the relevant part of the ontology in a given application.

On of the major problems related to this task of extracting modules from ontologies is that there is no clear definition of what should be in a module, i.e. what is relevant to a given sub-vocabulary of an ontology. Some techniques take a purely formal approach to this problem, defining logical criteria to find out what should be in a module. Some others rely on the traversal of the graph of relations that links ontological entities with each other. Each of these techniques is well suited only in a limited number of specific scenarios [dSSS07]. Also, while the fact that all these techniques take as input a sub-vocabulary of the ontology does not necessarily mean that the coverage of this vocabulary is the only criterion applied, other operators could consider other kinds of input to extract modules (based for example on the design properties of the modules or on the language applied in the labels of the entities). In this implementation, to avoid introducing even more heterogeneity, we choose to consider only operators with a set of entities of the ontology as input.

In this chapter, after a quick reminder about existing techniques for extracting modules, we present our implementation of the module extraction plugin for the NeOn Toolkit. With this plugin, considering the problem described above, we promote a different approach, where the ontology engineer can flexibly and interactively select, execute and combine a variety of techniques, until obtaining a satisfactory module, fulfilling the requirements of the given application scenario.

5.1 Ontology Module Extraction Techniques

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 [Stu06], 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.

5.2 A Plugin for Interactive Module Extraction based on Multiple Operators

In [dSSS07] we have shown through a number of experiments that extracting a module from an ontology is an ill-defined task: the criteria used to decide what should go in a module and what is a good, relevant module are highly dependent on the specificity of the application scenario. In other terms, there is no universal, generic module extraction approach. This appeared also very clearly in the different use cases described in [dHR$^+$08], where different users, in different contexts, provided completely different perspectives about what should go in a module. In general, what appeared from these use cases is that:

1. Users have different, more or less well defined ideas about what module extraction should do, varying from very elementary cases (e.g. extract a branch) to complex, abstract requirements (should extract everything that helps in interpreting a particular entity). Hence, each of the scenarios we encountered would require a different approach for module extraction.

2. Users want to keep in control of the way the module is created. It is required to support the parametrisation of the module extraction for the user to be able to really "chose" what goes into the module.

For these reasons, we implemented a plugin for module extraction that provides an interactive and iterative approach for this task. This plugin integrate a number of different "operators" for module extraction, most of them being relatively elementary (extract all the super/sub-classes of a given set of entities, all the other entities they depend on, etc.)

The interface for this plugin (see Figure 5.1) simply allows the user to easily combine these different elementary operators in an interactive way. A initial module can be created, using particular parameters (here the recursion level), obtaining an initial set of entities to be included. Then another operator can be used, on other entities and other parameters, to refine the module and extend it with other entities, until an appropriate module is created. At any point of the process, previous operations can be un-done and the module cleared.

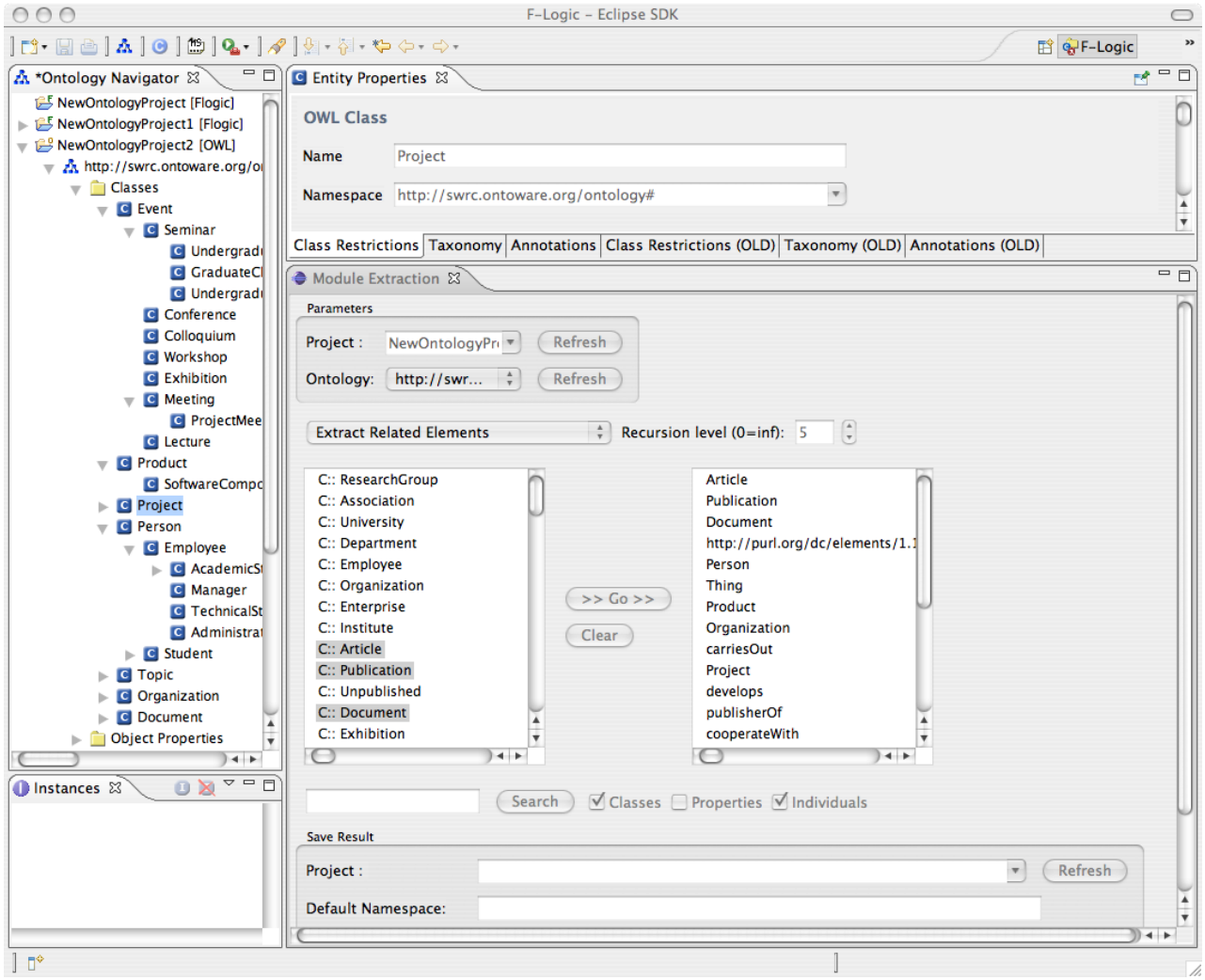

Figure 5.1: Screenshot of the ontology module extraction plugin.

In addition, the plugin provides straightforward functions to facilitate the selection of the entities to consider for module extraction. This includes restricting the visualization to classes, properties or individuals, and searching for entities matching a specific string. Once a module is created, it can simply be saved as part of the current ontology project and become itself processable as an ontology (module) to be composed or partitioned using the other modularization plugins.

5.3 A Plugin for Locality-based Modules

A completely different approach to specify what should be in a module is based on logical properties. Intuitively, a minimal module should contain exactly those axioms that are relevant for a particular interface (also called signature in this context), i.e. it should preserve all entailments, while at the same time no axioms should be included that are not needed to preserve the entailments.

For general description logics, it has been shown that is undecidable to compute minimal modules. But good approximations have been proposed. The approximations ensure that all entailments are preserved, but it may be that the modules contain some axioms that are not relevant.

One well established approximation is based on the notion of of *syntactic locality* and *locality-based module*, which have been first introduced in [GHKS07]. Syntactic locality is used to define the notion of module for a

Name	Syntax	Semantics
top	$\top$	$\Delta^{\mathcal{I}}$
concept name	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
nominal	$\{a\}$	$\{a^{\mathcal{I}}\}$
negation	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
exists restriction	$\exists r.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}} : (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
at-least restriction	$\geq n\, s.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y : (x,y) \in s^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq n\}$
role name	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
inverse role	r^-	$\{(x,y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (y,x) \in r^{\mathcal{I}}\}$
role hierarchy	$r \sqsubseteq s$	$r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
transitivity	$\mathsf{Trans}(r)$	$(x,y),(y,z) \in r^{\mathcal{I}}$ implies $(x,z) \in r^{\mathcal{I}}$
GCI	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$

Table 5.1: Syntax and semantics of $\mathcal{SHOIQ}$ concepts and axioms.

signature, i.e., a subset of the ontology that preserves the meaning of names in the signature.

To make the deliverable self-contained, we first need to introduce the Description Logic (DL) $\mathcal{SHOIQ}$ which is the underpinning DL formalism for OWL and used in our approach.

Starting with disjoint sets of concept names CN, role names RN and individuals Ind, a $\mathcal{SHOIQ}$-role is either a role name $r \in$ RN or an inverse role r^- with $r \in$ RN. We denote by Rol the set of all $\mathcal{SHOIQ}$-roles. $\mathcal{SHOIQ}$-concepts can be built using the constructors shown in the upper part of Table 5.1, where $a \in$ Ind, $r, s \in$ Rol with s a *simple role*[1], n is a positive integer, $A \in$ CN, and C, D are $\mathcal{SHOIQ}$-concepts.[2] We use the standard abbreviations: $\bot$ stands for $\neg\top$; $C \sqcup D$ stands for $\neg(\neg C \sqcap \neg D)$; $\forall r.C$ stands for $\neg(\exists r.\neg C)$; and $\leq ns.C$ stands for $\neg(\geq (n+1)s.C)$. We denote by Con the set of all $\mathcal{SHOIQ}$-concepts.

A $\mathcal{SHOIQ}$ ontology $\mathcal{O}$ is a finite set of *role hierarchy axioms* $r \sqsubseteq s$, *transitivity axioms* $\mathsf{Trans}(r)$, and a general concept inclusion axioms (GCIs) $C \sqsubseteq D$ with $r, s \in$ Rol and $C, D \in$ Con.[3] We use the notation $\mathrm{Sig}(\mathcal{O})$ to denote the signature of the $\mathcal{SHOIQ}$ ontology $\mathcal{O}$, i.e., the disjoint union of CN($\mathcal{O}$), RN($\mathcal{O}$) and Ind($\mathcal{O}$). Similarly, we write $\mathrm{Sig}(r)$, $\mathrm{Sig}(C)$ and $\mathrm{Sig}(\alpha)$ to denote the signature of a role, a concept and an axiom, respectively.

We can now define come back to the definition of syntactic locality.

Definition 3 (Syntactic locality for $\mathcal{SHOIQ}$) *Let* $\mathbf{S}$ *be a signature. The following grammar recursively defines two sets of concepts* $\mathsf{Con}^{\bot}(\mathbf{S})$ *and* $\mathsf{Con}^{\top}(\mathbf{S})$ *for a signature* $\mathbf{S}$*:*

$$\begin{aligned} \mathsf{Con}^{\bot}(\mathbf{S}) \quad ::= \quad & A^{\bot} \mid (\neg C^{\top}) \mid (C \sqcap C^{\bot}) \mid (\exists r^{\bot}.C) \mid (\exists r.C^{\bot}) \\ & \mid (\geq n\ r^{\bot}.C) \mid (\geq n\ r.C^{\bot}) \\ \mathsf{Con}^{\top}(\mathbf{S}) \quad ::= \quad & (\neg C^{\bot}) \mid (C_1^{\top} \sqcap C_2^{\top}) \end{aligned}$$

where $A^{\bot} \notin \mathbf{S}$ *is a concept name,* C *is a* $\mathcal{SHOIQ}$*-concept,* $C^{\bot} \in \mathsf{Con}^{\bot}(\mathbf{S})$*,* $C_i^{\top} \in \mathsf{Con}^{\top}(\mathbf{S})$ *(for* $i = 1, 2$*), and* $\mathrm{Sig}(r^{\bot}) \not\subseteq \mathbf{S}$*.*

An axiom α *is* syntactically local w.r.t. $\mathbf{S}$ *if it is of one of the following forms: (i)* $r^{\bot} \sqsubseteq r$*, (ii)* $\mathsf{Trans}(\mathsf{r}^{\bot})$*, (iii)* $C^{\bot} \sqsubseteq C$ *or (iv)* $C \sqsubseteq C^{\top}$*. The set of all* $\mathcal{SHOIQ}$*-axioms that are syntactically local w.r.t.* $\mathbf{S}$ *is denoted by* $\mathsf{s\_local}(\mathbf{S})$*. A* $\mathcal{SHOIQ}$*-ontology* $\mathcal{O}$ *is* syntactically local w.r.t. $\mathbf{S}$ *if* $\mathcal{O} \subseteq \mathsf{s\_local}(\mathbf{S})$*.*

Based on this notion, locality-based modules can be defined as follows: Let $\mathcal{O}$ be a $\mathcal{SHOIQ}$ ontology, $\mathcal{O}' \subseteq \mathcal{O}$ a subset of it, and $\mathbf{S}$ a signature. Then, $\mathcal{O}'$ is a *locality-based module for* $\mathbf{S}$ *in* $\mathcal{O}$ if every axiom $\alpha \in \mathcal{O} \backslash \mathcal{O}'$ is syntactically local w.r.t. $\mathbf{S} \cup \mathrm{Sig}(\mathcal{O}')$. Given an ontology $\mathcal{O}$ and a signature $\mathbf{S}$, there always exists a unique, minimal locality-based module [CHKS08], denoted by $\mathcal{O}^{\mathsf{loc}}_{\mathbf{S}}$.

[1] A simple role is neither transitive nor a superrole of a transitive role.

[2] Concepts and roles in DL correspond to classes and properties in OWL, respectively.

[3] A concept definition $A \equiv C$ is an abbreviation of two GCIs $A \sqsubseteq C$ and $C \sqsubseteq A$, while ABox assertions $C(a)$ and $r(a,b)$ can be expressed as the GCIs $\{a\} \sqsubseteq C$ and $\{a\} \sqsubseteq \exists r.\{b\}$, respectively.

Ontologies	♯Axioms	♯Concepts	♯Roles	Module size		Extraction time (sec)
				Average	Maximum	
GALEN	4 529	2 748	413	75	530	6
GO	28 897	20 465	1	16	125	40
NCI	46 940	27 652	70	29	436	65

Table 5.2: Benchmark ontologies and their characteristics.

We have implemented a plugin that computes these locality-based modules within the NeOn Toolkit. The interaction is very simple: The user simply specifies the signature (interface), the plugin computes the module as a result.

5.4 Example Uses and Evaluation

We now present an example of how modularization based on module extraction can be used to speed up reasoning tasks. We consider as specific reasoning task *finding the justifications for an entailment* (i.e., minimal sets of axioms responsible for it). Finding justifications is important in ontology engineering, as justifications facilitate important tasks like debugging inconsistencies or undesired subsumption. Though several algorithms for finding all justifications exist, issues concerning efficiency and scalability remain a challenge due to the sheer size of real-life ontologies. Based on the notion of locality-based modules, in [SQJH08] we proposed a method for finding all justifications in OWL DL ontologies by limiting the search space to smaller modules. In the paper, we showed that so-called locality-based modules cover all axioms in the justifications.

Intuitively, the (minimal) locality-based module for $\mathbf{S} = \{A\}$ in a OWL DL ontology $\mathcal{O}$ contains *all* the relevant axioms for any subsumption $\sigma = (A \sqsubseteq_{\mathcal{O}} B)$, in the sense that all responsible axioms for σ are included. In other words, in order to find all justifications for a certain entailment in an OWL ontology, it is sufficient to consider only axioms in the locality-based module. Since the *minimal* locality-based modules are relatively small (see, e.g., [GHKS07, Sun08]), our modularization-based approach proves promising.

For the details, theory and algorithms of how the locality-based modules are used to find entailments, we refer the reader to [SQJH08].

We here briefly summarize the empirical results that demonstrate an improvement of several orders of magnitude in efficiency and scalability of finding all justifications in OWL DL ontologies.

Our algorithm has been realized by using KAON2 (the default reasoner of the NeOn Toolkit) as black box reasoner. To fairly compare with the pinpointing algorithm in [KPHS07], we re-implemented it with KAON2 API (henceforth referred to as ALL_JUSTS algorithm). The experiments have been performed on a Linux server with an Intel(R) CPU Xeon(TM) 3.2GHz running Sun's Java 1.5.0 with allotted 2GB heap space.

Benchmark ontologies used in our experiments are the GALEN Medical Knowledge Base[4], the Gene Ontology (GO)[5] and the US National Cancer Institute thesaurus (NCI)[6]. The three biomedical ontologies are well-known to both the life science and Semantic Web communities since they are employed in real-world applications and often used as benchmarks for testing DL reasoners. Both GO and NCI are formulated in the lightweight DL $\mathcal{EL}$, while GALEN uses expressivity of the more complex DL $\mathcal{SHF}$. Some information concerning the size and characteristics of the benchmark ontologies are given in the left part of Table 5.2.

Modularization reveals structures and dependencies of concepts in the ontologies as argued in [CHKS08, Sun08]. We extracted the (minimal) locality-based module for $\mathbf{S} = \{A\}$ in $\mathcal{O}$, for every benchmark ontology $\mathcal{O}$ and each concept name $A \in \mathsf{CN}(\mathcal{O})$. The size of the modules and the time required to extract them are shown in the last three columns of Table 5.2. Observe that the modules in GALEN are larger than those in the other two ontologies although the ontology itself is smaller. This suggests that GALEN is more complex in the sense that more axioms in it are non-local (thus relevant) according to Definition 3.

[4] http://www.openclinical.org/prj_galen.html
[5] http://www.geneontology.org
[6] http://www.mindswap.org/2003/CancerOntology/nciOntology.owl

In the experiments, we consider three concept names in $\mathsf{CN}(\mathcal{O})$ for each benchmark ontology $\mathcal{O}$ such that one of them has the largest locality-based module[7]. For the sake of brevity, we denote by $\mathsf{subs}(\mathcal{O})$ the set of all tested subsumptions $A \sqsubseteq B$ in $\mathcal{O}$, with A one of the three concept names mentioned above and B an inferred subsumer of A. For each $\mathcal{O}$ of our benchmark ontologies, we compute *all* justifications for σ in $\mathcal{O}$, where $\sigma \in \mathsf{subs}(\mathcal{O})$. In order to compare with state-of-the-art, existing approaches (that are not based on modularization), we perform the following for each σ and $\mathcal{O}$ to compute all justifications:

1. ALL_JUSTS$(\sigma, \mathcal{O})$ (i.e., the algorithm in [KPHS07]).
2. REL_ALL_JUSTS$(\sigma, \mathcal{O}, s_{rel})$ (i.e., the algorithm in [JQH08]);
3. MODULE_ALL_JUSTS$(\sigma, \mathcal{O})$;

To visualize the time performances of the three algorithms, we randomly selected two subsumptions σ_1 and σ_2 from $\mathsf{subs}(\mathcal{O})$ for each ontology $\mathcal{O}$ and compared their computation time required by the three algorithms. These subsumptions are shown as follows:

GALEN:X_1	AcuteErosionOfStomach	$\sqsubseteq$	GastricPathology
GALEN:X_2	AppendicularArtery	$\sqsubseteq$	PhysicalStructure
GO:X_1	GO_0000024	$\sqsubseteq$	GO_0007582
GO:X_2	GO_0000027	$\sqsubseteq$	GO_0044238
NCI:X_1	CD97_Antigen	$\sqsubseteq$	Protein
NCI:X_2	APC_8024	$\sqsubseteq$	Drugs_and_Chemicals

The chart in Figure 5.2 depicts the overall computation time required for each algorithm to find all justifications for each tested subsumption. Unlike the time results reported in [KPHS07], which excluded the time for satisfiability checking, we report here the overall computation time, i.e. the total time of the algorithm including the time needed by the black-box reasoner for the standard reasoning tasks. Observe that both ALL_JUSTS and REL_ALL_JUSTS did not yield results within the time-out of two hours on three out of six tested subsumptions (marked by "TO" on the chart). Comparing these two algorithms (without modularization), REL_ALL_JUSTS performs noticeably better than ALL_JUSTS in most cases. For instance, on the subsumptions GALEN:X_2 and NCI:X_2, REL_ALL_JUSTS outperforms ALL_JUSTS by about 10 and 20 minutes, respectively. On the subsumption GO:X_2, both algorithms show a similar performance, i.e., time difference is less than a minute. More explanations on comparison between these two algorithms can be found in [JQH08].

Interestingly, MODULE_ALL_JUSTS outperforms all the other algorithms on all subsumptions, and the improvement is tremendous as can be seen in all cases in the chart. As an example, MODULE_ALL_JUSTS took *only* 0.6 seconds to find all the justifications for NCI:X_2, while REL_ALL_JUSTS needed 3 242 seconds. In this case, the locality-based module for APC_8024 in NCI consists of 9 axioms, whereas the whole ontology has some tens of thousands of axioms.

[7] The concept name with largest module is hand-picked in order to cover hard cases in our experiments, while the other two are randomly selected.

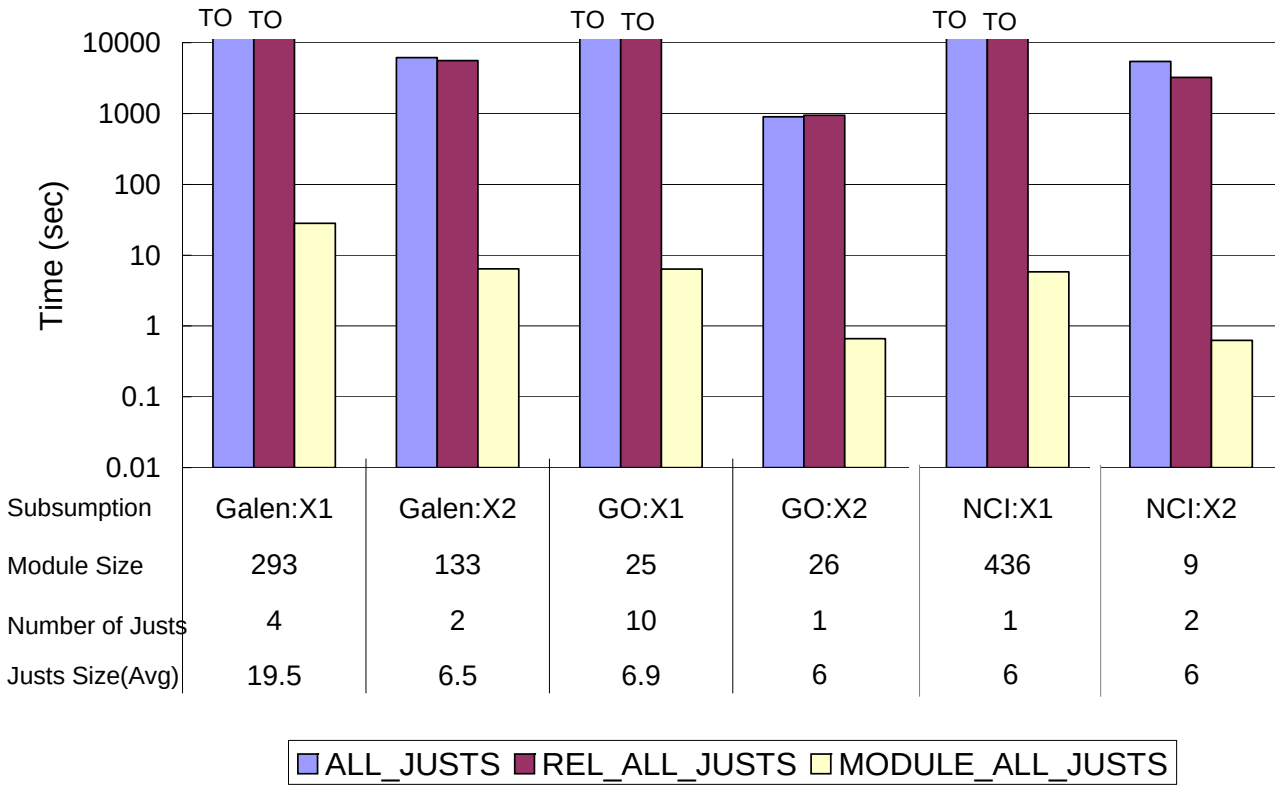

Figure 5.2: The time performance of three algorithms for finding all justifications.

Part III

Discussion

Chapter 6

Other Aspects of Modularization

In this chapter, we briefly summarize other ongoing work related to modularization and that will be considered in more details in other deliverables.

6.1 Methodological Guidelines for Modularization

Designing ontologies in a modular way is generally considered as a good practice. However, there are scenarios where an ontology designer need to reuse or exploit an ontology that has not been modularized at design time, or for which the criteria applied for distinguishing modules do not fit the particular requirements of his/her current application. This task of creating modules from an existing ontology is supported by several plugins described in the second part of this document.

To obtain a module or a set of modules from an ontology that suits the requirements of a particular application, there is a clear need for a guideline, a method, supporting developers in selecting and applying the appropriate techniques. For this purpose, we devise an ontology modularization guideline, to be part of the NeOn methodology and that will be described in deliverable 5.4.2.

The major issue faced by ontology engineers when applying ontology modularization techniques is that the task is relatively ill-defined. Indeed, the criteria to decide how modules should be identified, created and extracted my vary a lot from one scenario to another [dHR$^+$08]. For this reason, as depicted in figure 6.1, we promote an iterative and interactive approach to modularization, where the ontology engineer (with the help of domain experts) can apply a variety of techniques, evaluate the results and refine them, until obtaining a satisfactory, tailored made (set of) module(s). This approach is well supported by the modularization support implemented within the NeOn Toolkit, as described in this document.

As part of building this guideline, we will also study a complete experiment of modularizing an ontology, in a real-life scenario. For this experiment, we will use the plugins described in this deliverable and the guideline devised in workpackage 5. This will provide a complete evaluation of both the tool support and the methodology for modularization in NeOn, complementing the tests described in this document on individual components.

6.2 Reasoning with Modules

In [dHR$^+$08], we described the semantics of the NeOn formalism for ontology modules, which fixes the way logical consequences can be derived from modules linked by mappings. This semantics is based on the IDDL formalism. It establishes the formal foundation for a reasoner on interlinked ontology modules.

In this section, we briefly summarize an algorithm for reasoning on modular ontologies with the IDDL semantics and an IDDL reasoner based on the presented algorithm. These elements constitute the first step towards

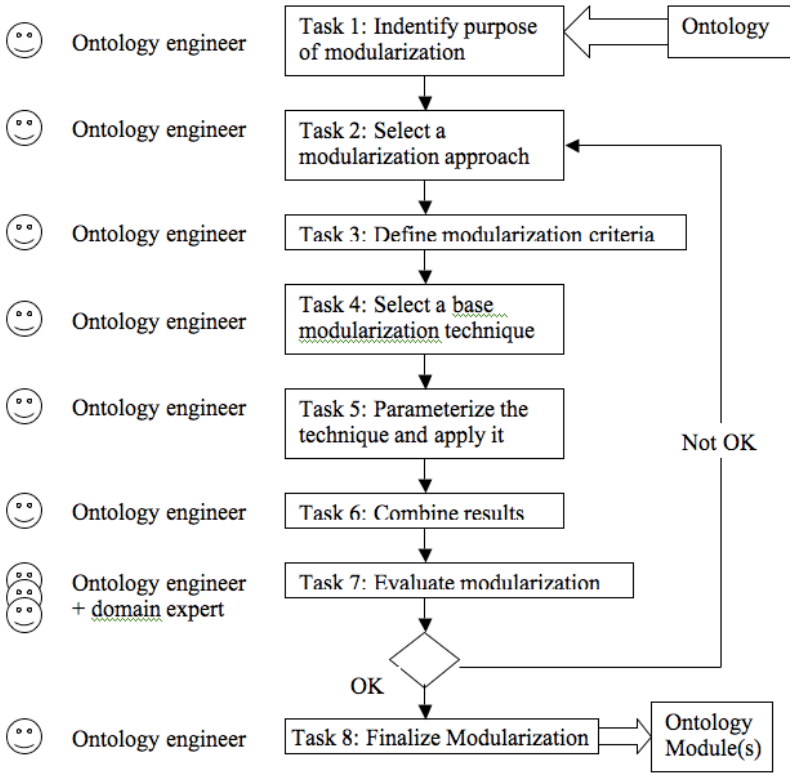

Figure 6.1: Workflow for the task of modularizing an existing ontology. This figure summarizes the activities realized as part of this task, as they will be described in deliverable 5.4.2.

a more general *reasoner for networked ontologies*, which will be described as part of deliverable D1.4.4. This is the reason why only a brief summary is presented here.

6.2.1 Reasoning with IDDL Modules

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

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 defined a notion of equalising function which helps making these domains commensurate (see [dHR+08]) by relating them to a common general domain.

The IDDL reasoner checks the consistency of a distributed modular ontology (see [**?**]). It works by having a module reasoner communicate with imported modules' reasoners. Each module reasoner acts as a *reasoning oracle* for the others, meaning that, considering an ontology O in a logic L (with $\mathcal{P}L$ the set of axioms in L), they implement a boolean function $F : \mathcal{P}L \rightarrow \mathrm{Bool}$ which returns $F(A) = \mathrm{Consistency}_L(O \cup A)$, for all sets of axioms $A \in \mathcal{P}L$.

The algorithm consists in chosing a set of concepts over the set of all concepts of the mappings between modules. The axioms associated with this set are then sent to the oracles to verify the consistency of the resulting ontologies. If they all return *true* (i.e., they are all consistent with these additional axioms) then the modular ontology is consistent. Otherwise, another set of concepts must be chosen. If all sets have been tested negatively, the modular ontology is inconsistent. Since there is a finite number of configurations, this

algorithm is correct and complete.

6.2.2 Implementation

The IDDL reasoner provides a basic interface to check consistency of an ontology module which consists of a set of imported ontologies in OWL and mappings between them. The current IDDL reasoner uses the Pellet reasoner as local reasoner for checking consistency of an imported ontology from an ontology module. The interface between the IDDL reasoner and local reasoners is designed so that any other reasoner, e.g. FaCT++, Racer, Drago, etc., can easily replace Pellet. Additionally, the IDDL reasoner uses the Alignment API [**?**] to manipulate correspondences during the reasoning process.
The reasoner also offers a possibility to get an explanation for the inconsistency of an IDDL system.

The IDDL reasoner plug-in for the NeOn Toolkit relies on the IDDL reasoner, and the core module API, which provides basic operations to manipulate ontology modules and mappings. It offers to users an interface to obtain an answer for consistency from the IDDL reasoner. In the case where the answer is negative, the plug-in can obtain an explanation indicating concepts and/or correspondences which are responsible for that inconsistency.

Chapter 7

Conclusion

In this document, we described the implementation of the NeOn toolkit support for ontology modularization, considering both aspects of designing modular ontologies and creating modules from existing ontologies. This area of ontology modularization has been attracting more and more attention in the last few years, as complex ontologies are being created and maintained in concrete applications. However, the results from the research community in this area are still very rarely applied concretely. This is due in particular to the difficulty of finding a common view on modularization. Many studies have been looking at ways to formalize the notion of modules in modular ontology languages, while other are considering the task of decomposing ontologies and extracting modules from them. Even within these two perspectives, different views appear, as the requirements from applications may vary a lot, providing many possible understanding of what ontology modularization could mean. As a consequence, until now, the research community has failed to propose proper tool support for ontology engineer in need of concrete solutions for ontology modularization.

In this deliverable, we have tackled this situation by implementing a comprehensive set of tools, supporting ontology modularization in its many aspects. Following the formalism proposed in [dHR$^+$08], we implemented a basic framework for specifying ontology modules within the NeOn Toolkit. Using this generic framework, we proposed a number of plugins, implementing various operators for manipulating modules. In a nutshell, we provide a complete, integrated environment for ontology modularization, which is made of a several, interoperable components each handling a different task. The added-value of such developments within the NeOn Toolkit goes beyond the availability of each individual tool. Indeed, the NeOn toolkit support for modularization makes it easy for ontology engineers to handle ontology modules in a controlled, interactive manner, by allowing them to create, edit, manipulate, extract, decompose, and combine modules within the same ontology editor and in an homogeneous way.

To complement this implementation work, we are currently looking at other aspects to support regarding ontology modularization. In particular, for the usage and deployment of ontology modules in applications, we are developing a reasoner for networked ontologies, able to handle distributed modules based on the IDDL semantics for mappings. Also, the interactive and iterative approach we promote for ontology modularization requires proper methodological guidelines, supporting ontology engineers in selecting, evaluating and refining ontology modules. Such guidelines will be provided as part of the NeOn methodology, on the basis of the tools described here.

Bibliography

[BGvH+03] 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.

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

[CHKS08] Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Modular reuse of ontologies: Theory and practice. *J. of Artificial Intelligence Research (JAIR)*, 31:273–318, 2008.

[dHR+08] Mathieu d'Aquin, Peter Haase, Sebastian Rudolph, JŐrŹme Euzenat, Antoine Zimmermann, Martin Dzbor, Marta Iglesias, Yves Jacques, Caterina Caracciolo, Carlos Buil Aranda, and Jose Manuel Gomez. D1.1.3 neon formalisms for modularization: Syntax, semantics, algebra. NeOn Deliverable 1.1.3, The Open University, March 2008.

[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.

[GHKS07] B. Cuenca Grau, I. Horrocks, Y. Kazakov, and U. Sattler. Just the right amount: Extracting modules from ontologies. In *Proceedings of WWW*, pages 717–726, Banff, Canada, 2007. ACM.

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

[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.

[HSH+05] 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.

[JQH08] Qiu Ji, Guilin Qi, and Peter Haase. A relevance-based algorithm for finding justifications of dl entailments. In *Technical report, University of Karlsruhe, http://www.aifb.uni-karlsruhe.de/WBS/gqi/papers/RelAlg.pdf*, 2008.

[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.

[KPHS07] Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding all justifications of OWL DL entailments. In *Proc. of ISWC/ASWC'07*, pages 267–280, 2007.

[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.

[Nab08] Janiko Naber. Entwicklung eines modularisierungskonzeptes von ontologien f"ur ontomodel. Master's thesis, Hochschule Karlsruhe, 2008.

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

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

[SQJH08] Boontawee Suntisrivaraporn, Guilin Qi, Qiu Ji, and Peter Haase. A modularization-based approach to finding all justiŕcations for owl dl entailments. In *Proceedings of the Asian Semantic Web Conference, ASCW 2008*, 2008.

[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.

[Stu06] H. Stuckenschmidt. Toward Multi-Viewpoint Reasoning with OWL Ontologies. In *Proc. of the European Semantic Web Conference (ESWC)*, 2006.

[Sun08] Boontawee Suntisrivaraporn. Module extraction and incremental classification: A pragmatic approach for ontologies. In *Proc. of ESWC'08*, pages 230–244, 2008.

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