NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

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

D4.2.3 Ontology customization prototype presentation

Deliverable Co-ordinator: **Noam Bercovici**

Deliverable Co-ordinating Institution: **University of Koblenz-Landau (UKO-LD)**

Other Authors: **Simon Schenk(UKO-LD)**

This deliverable presents a prototype for realizing ontology customization, which can be used to propose to the user a personalized view on an ontology for his/her current task or his/her job.

Document Identifier:	NEON/2009/D4.2.3/v1.1	Date due:	February 28, 2009
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	September 15, 2009
Project start date	March 1, 2006	Version:	v1.1
Project duration:	4 years	State:	Final
		Distribution:	Public

NeOn Consortium

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

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

Work package participants

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

- University of Koblenz-Landau (UKO-LD)
- Open university (OU)
- Laboratorios KIN, S.A. (KIN)

Change Log

Version	Date	Amended by	Changes
0.1	29-10-2008	Noam Bercovici	Initial customization description
0.2	08-01-2009	Noam Bercovici	Add the section Process
0.3	18-01-2009	Noam Bercovici	Modification of the section Customization
0.4	06-02-2009	Noam Bercovici	Add the section use case & architecture
0.5	21-02-2009	Noam Bercovici	Review of the section customization vs. modularization
0.6	25-02-2009	Noam Bercovici	Refine the chapter plugin
0.7	26-03-2009	Noam Bercovici	Add introduction and conclusion
1.0	06-04-2009	Noam Bercovici & Simon Schenk	Aswer to the QA review
1.1	15-08-2009	Noam Bercovici	Aswer to the reviewer's comments

Executive Summary

Real world ontologies nowadays become lager and more complex, therefore the user has to deal with irrelevant parts of the ontology. Customization can be one way to solve this problem by reducing the amount of information presented to the user and give to him/her a smaller and relevant ontology for his/her current task.

An important aspect of this deliverable is the presentation of customization in NeOn and the differences with modularization. Actually, customization and modularization are two notions which have similarities, so this deliverable will make clear the orthogonality of each one by showing their uses, advantages and disadvantages.

The goal of this deliverable is to present the customization plugin based on templates. A prototype of the plugin is implemented, we present here its description and its use.

Contents

List of Figures

Chapter 1

Introduction

This rapport is the third part of the work did on the ontology customization task. First, in [DDM+07] we describe the state of the art for customization techniques and we present several use cases where they could be apply. This state of the art highlights two main ways to custom ontology, at the graphical level to hide or to bring the focus of the user on a specific part of the ontology, this approach is preferred by [KS03] in the fish eye projection. Another way of customization is using algebra operators to change the structure of the ontology as [Wie94]. Afterward, in [BDS+08] we propose a method in between those two, using a query engine to evaluate customized view like the idea propose by J.F. Brinkley in [LTD06] and refine later in [LTDF07]. The previous deliverable presents the new query language for OWL: SAIQL. Now we will describe how the customization using view mechanism is working for that we will organize this deliverable in four parts. First we will position this work in the NeOn project by reminding the reader of a test case in which customization issues arise. Thereafter, we will present the customization in the perspective of NeOn. Thenceforth, the ontology customization process will be described. In fact, this process needs several inputs, the user preferences and the ontology which would be customized. This process involves a query engine to propose to the user a custom view. Afterward, we will explain the needs of a new query language next to SPARQL. Finally, we will propose the description of the architecture of the plug-in for customizing an ontology with a "quick start "user guide.

Obviously, since customization relates to such aspects as context or modularization, this work is related to other work packages in the NeOn project. In particular, we highlight a few points of overlap and where potential interactions can be found with the other work packages. First, one particular form of creating networked ontologies investigated in WP1 is their modularization. An ontology module is seen as a tuple of imported ontologies, certain import and export interfaces, and a set of mappings. Formal models of modularization techniques have been presented in [dHR+07]. By the way, we will devote a section to point out the differences and the overlap between those techniques.

The place of customization in WP4

We also mentioned in the previous works that a good visualization of ontologies is an important aspect of ontology construction tools. From the user studies made in D4.1.1 [DMG+06], visualization supported by the existing tools is rather poor, being too general and providing limited support for problem-specific needs. Concatenating visualization techniques with ontology customization operators (e.g., the one for pruning, described in the previous deliverable) may help to show only the relevant, more focused parts of ontologies, rather than showing the entire graphs with potential thousands of nodes. Thus, a good level of detail in visualizing can be offset by showing only a customized ontology view - an approach that clearly complements the techniques for context-sensitive visualization that strive to show large and networked ontologies by abstracting the level of visualized details.

Access control presented in [DKG+07] and [Dzb09] benefits clearly from the customization technique for creating customized view of an ontology. Those views represent the right amount of information that the user

is allow to access. In oder to specify the view definition the administrator can use the customization tools presented in this rapport with specific templates.

Relationship of this work in the other NeOn work package

Relationship to WP2 is less obvious, however, in the process of ontology customization we use templates, which can be filled in with a concrete input from the user or which can be bootstrapped and executed on a user profile. In other words, these defined queries may be seen as a specific type of "ontology patterns "; albeit we use those templates less as "ontology design pattern "and more in a role of a driver for simplifying views on an ontology. In WP2 the research is focusing on a more generic set of patterns; mainly on those used during ontology design phase [Gan05].

With respect to research done in WP3, the relationship is somewhat clearer. Customizing an ontology by adjusting what is shown to a particular user (e.g., by pruning unnecessary nodes and branches) can be pragmatically, seen as bringing an ontology in the context of a particular user. Even more importantly, the selection and use of particular templates by a given group of users may be seen as a valid contextual modifier for segmenting users into groups and for automatically constructing user profiles. The added value of our work is that SAIQL queries represent user preferences in terms of procedural knowledge, i.e., how they prefer to interact with a given ontology; whereas the majority of other user profiling techniques is focusing on what the user interacts with.

Chapter 2

Use cases for customization

Laboratory KIN has two departments involved in the invoices process depending on the type. Customer invoices are managed by the Sales Department and supplier invoices by the Financial Department. Those departments annotate the customer and supplier invoices. Afterwards the Sales Department can exploit this. Sales agents should be able to show up the need of the customer from his past invoice.

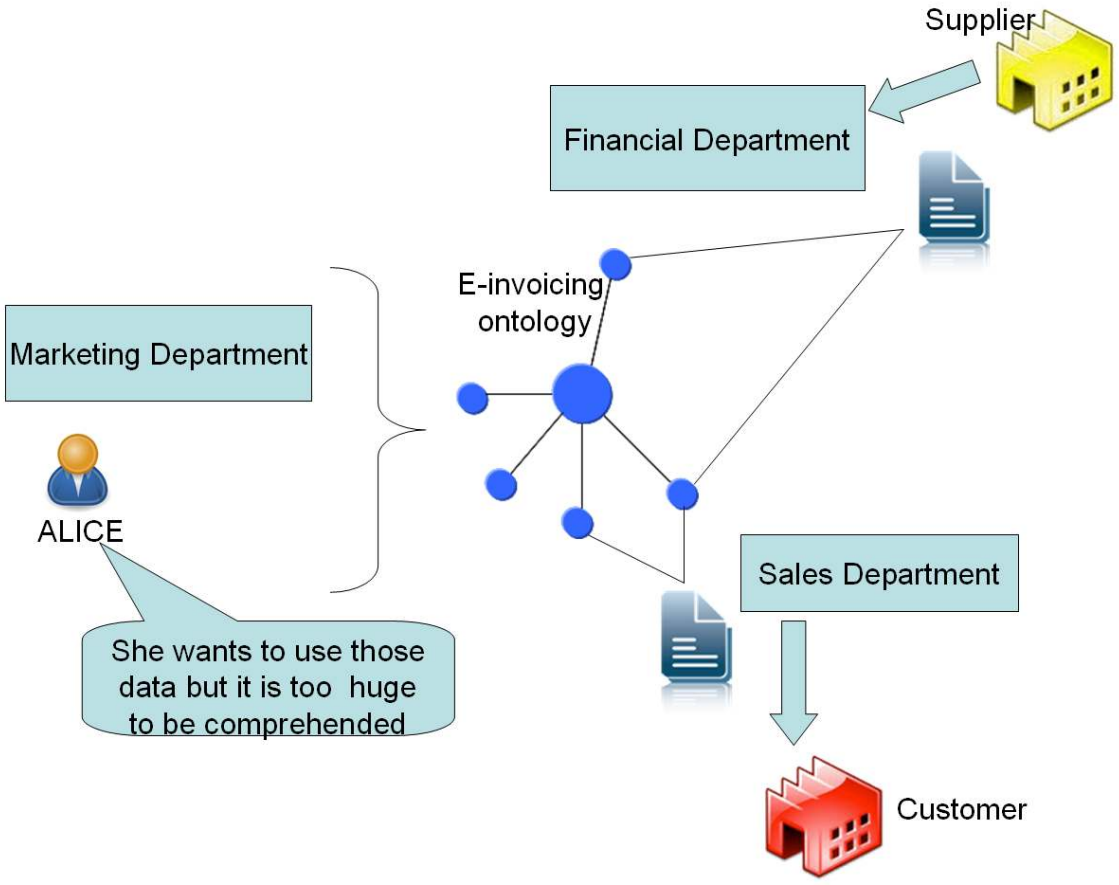


Figure 2.1: Use Case Description

2.1 Customization Use Case

Laboratory KIN has two departments involved in the invoices process depending on the type. Customer invoices are managed by the Sales Department and supplier invoices by the Financial Department. One

of the tasks involved in those two departments is to annotated each customer and supplier invoices with e-invoicing ontology [GPBH+07] from ISOCO. All the information content in the invoice is now represented in a DL-ontology. This allows for many possibilities to exploit those informations. Thus, a sales agent from the Sales department of KIN, named Alice, should be able to anticipate from their last invoices what the customer may need and propose it to him. Alice is very good in her domain but she is not a specialist of ontologies. Nevertheless most of this information she needs are content in e-invoicing ontology. She feels lost in this large ontology represented in figure 2.1.

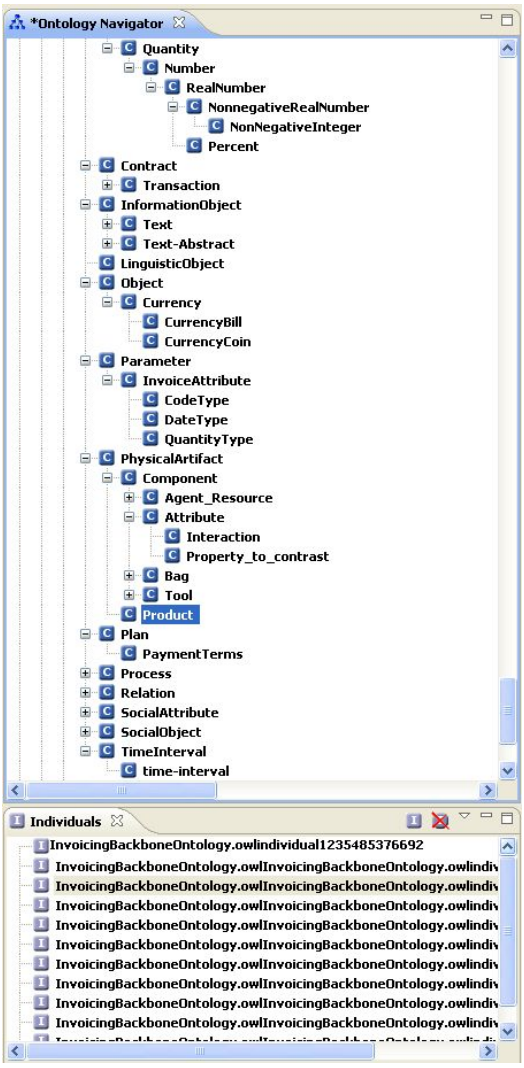


Figure 2.2: Partial view of the ontology

Alice knows what she would like to have as custom view. In order to extract relations between products, Alice might want to work with a view which represents all the products presented in the same invoice as the product X. She does not want to affect the original ontology because this view will be used by her and only for a couple of times.

2.2 Requirements

Some of the end users from our use case partner are not so familiar with ontologies or description logic. For those users they define several requirements to reduce or personalize the view on the ontology.

The two following requirements are extracted from [GPDM+06]. The first one is from the semantic nomen-

clature and the second from iSOCO:

1. "Some techniques should serve to generate consistent local views of the ontologies outside [...] of networked ontologies."
2. "Users of electronic invoicing solutions can typically have two roles, emitter or receiver, each with an associated context. Context information is useful in order to personalize visualization and edition of invoice models and data."

FAO in [ICJ+08], more precisely in the requirement UC30, argues for the necessity to filter ontologies. They need a tool for extracting every thing related to a concept or for reducing the amount of information presented to the user.

Chapter 3

Customization in NeOn

As ontologies become more and more complex and as they are integrated into networks of ontologies, it is reasonable to investigate the means, which would be capable of making a large network of complex ontologies more manageable. The customization and personalization of ontologies includes, in principle, two areas that are relevant to the NeOn project. First, there is a possibility to customize ontologies, e.g., during exploring a network of ontologies. This customization is more or less ad-hoc and the results of the customization may be discarded once the user proceeds with exploring the ontology. This customization during exploring an ontology tries to reduce the complexity of an ontology and only show parts which are relevant for the current user. Second, one can customize the ontology schema itself, for the purposes of reusing ontologies and integrating them into a network with other ontologies according to specific needs (e.g. during the ontology deployment, reasoning or design phases). Here the results of the customization will often be integrated into the edited ontology.

3.1 Motivation

We saw in section 2.2 requirements posted by the use case partner concerning the notion of customization. Even if the term customization is not use explicitly, we can clearly see from their description that it covers our definition of customization detailed in the paragraph 3.3.2. The notion of "filter ontology" [ICJ$^+$08], "local view" [GPDM$^+$06] or "personalized visualization of the model and data" [GPDM$^+$06] are use by our partner. Our conception of customization provides a solution for all of those notions.

3.2 Definition

3.2.1 View

The notion of *views* is prevalent in the relational database world. In database terminology, a view is a query that computes a new table from a pre-existing one. In the case of customization, we extend this notion to the ontologies. A view is defined as the result of a query expressed in a formal OWL ontology query language. To each view we associated a custom label, that allows the user to refer easier to the view.
$V =< Id, Q, L >$

1. Id is the URI of the view;
2. Q is a query which is required to build the view;
3. L is the custom label which represents the label of the view;

In practical, multiple queries can compose a view but theoretically all those queries can be merged in to one query, if those queries are evaluated on the same ontology and that is the case here. We will see in chapter 4

a technique to combine and create a sequence of views.

Example of a view:

The figure below shows an example of a view on an currency ontology. Let's have a look at Joe Bloggs, the financial department supervisor of a pharmaceutical laboratory. One of his missions is to ensure the smooth running of the strategic plan in every department of the laboratory. This job requires to check the ratio of how much each department earns and how much they sell. To cope this task, Joe does not need to know in detail what each department is selling (goods, drugs, services or/and patents). For making this job easier, he may want to create a shortcut like "Product" which is everything that brings money to them. In this context it means that "Product" is defined as the concept which is the union of the concepts: "Good", "Drug", "Services" and "Patent". Of course, Joe does not want to replace the original definition of "product" in the ontology of the company but wants to use this shortcut in this retrieval task. The views which are used for customization can allow the user to build such view. The view bellow represents the shortcut wanted by Joe.

```
<

Id of the View,

Q1:  CONSTRUCT SubClassOf(?X ?Z); Individual(?i type(?X))
      FROM OntologyURI
      LET  ClassName ?X; ClassDescription ?Z, ?Y; Individual ?i
      WHERE SubClassOf( ?X Union (Good Drug Service Patent) )
         AND Individual(?i type(?X))
   AND SubClassOf( ?X ?Z )
,

Product

>
```

The main idea of the use view in the process of customization is still to propose to the user some facilities to define a special meaning of a concept following his context. An user may want to have for the same concept several views attached to it, that means the user can link the concept with several definitions of it and use one instead of the other one following his task or his current context.

3.2.2 Template

In order to help the user to customize an ontology following is preferences, we introduce query design patterns, called template. Those templates are formulated in SAIQL patterns. A template is an abstraction of view which is not depending of a specific ontology.

$T =< N, AP >$

1. N: a name
2. AP: a set of axiom patterns

Example of a template: The Figure below represent a query design pattern which can extract a part of the ontology from an ontology. This query template represent the template use to generate the previous example of a view (cf. 3.2.1).

```
SubClassOf(?X ?Z); Individual(?i type(?X)); EquivalentClasses(?Z ?X)
```

3.3 Customization vs. Modularization

This section compares the techniques customization and modularization. We want to make clear the orthogonality of each one by showing their uses, advantages and disadvantages. To reach this goal we will first formally define those two notions. Thereafter we will explain the similarities and the differences at the technical and conceptual level.

3.3.1 Definition of Modularization

The following definition of modularization is extract from [dHR$^+$07]. *A modular ontology is made of smaller local ontologies that can be seen as self-contained and inter-related modules, combined together for covering a broader domain. Indeed, an ontology is not inherently a module, but rather plays the role of a module for other ontologies because of the way it is related to them in an ontology network. In other terms, an ontology module is a self-contained ontology, seen according to a particular perspective, namely reusability. The content of an ontology module does not differ from the one of an ontology, but a module should come with additional information about how to reuse it, and how it reuse other modules.*

For example, Alice, who is an ontology designer, would like to reuse for the ontology O_1, that she is designing, a specific design pattern DP_1 described in Dolce. In this example the use of modularization is required, indeed we want to reuse a specific part from the huge ontology and not import all Dolce in the new ontology O_1. In this case creating module contenting DP_1 is the best option.

The use cases described for the modularization in [dHR$^+$07] are:

1. modularization is helpful to design ontology;
2. re-usability;
3. the modularization improves the performance, by reducing the amount of knowledge which is manipulated by the reasoner and the editor;
4. The last use of modularization described how facilitating the exploration and the maintenance of the ontology.

Those use cases are clearly oriented on ontology design, a module is made to be shared and reused at a large scale.

3.3.2 Definition of Customization

A customized view is an self-contained ontology representing a personal "view/aspect" for an user of his/her domain for his/her current needs. However, this customized view does not affect the original ontology, the customization technique we propose is based on the technique used in the field of data base for decades. Indeed, the notion of view is already well investigated in this field so we use it as a base of our work. We extend the view to the ontology providing the possibility to "create" view on the instances and the schemas unlike in data base which allows only view on the instances. The main advantage to use view for customizing ontology is to have this dynamic aspect provided by the view. This ensures the user to have at all times an updated view of the original ontology.

The following example shows an use of the customization which can be solved only by customization techniques. Take a pharmaceutical laboratory as KIN which manages hundreds invoices per days. In this example we will have a look at the three specific employs: Alice, who works in the sells department, Bob, who is the supervisor of the production department and Carol who works in the financial department. Alice has the job of selling several new products to the customer, for her task she has an access to the part of the invoice concerning the customer e.g. address, name and so on. Bob needs to have access to the product and the quantities of the invoice to adjust the plan of the production. A solution which can be proposed is to create

two module of the ontology which represents the invoice. One module would contain the customer data and another one the rest of the invoices. The problem of this solution is that Carol needs to have a global view on the invoice. That means we have to merge the two modules each time Carol wants to access to a complete invoice. Thus, the best solution is to create for Alice and Bob views on the ontology representing the invoice.

3.3.3 Similarities

The modularization in NeOn which belongs to the work package 1, has several supports e.g. algebraic operation between module and module extraction. This last support is the one which seems to bring the most confusion between the notion of customization, as we are presenting in the WP4, and the modularization. The module extraction aims to create smaller chunks of ontology from a bigger one. In the customization, there is a techniques for extracting chunks from huge ontologies too. However, different techniques are used at the technical level to achieve the extraction. For example, the customized views are the result of queries and modules are achieved by graph manipulation. In the next paragraph we will show that other differences exist at the technical and the conceptual level between the both notions.

3.3.4 Differences

From the formal definitions described in the section 3.3.1 and 3.3.2 one thing comes up; Modularization and customization are two notions usable at different stage of the ontology life cycle. The first one is use while the ontology is in the design phase. The second one is used mainly during the "exploitation" phase of the ontology. Customization and modularization present other differences especially at the conceptual level. Actually, when the user customizes an ontology, he may want to define a special mean of something. The customization bring a tool to the user which allow this specialization following the context. The user may want to create shortcuts for his personnel information retrieval task or in own context; this is what the customization allows him to do.

At the technical level the differences still exist. Actually, the customized view is made with a query engine unlike the module which is obtained by using graph manipulation techniques.

As shown in this section, while similar in principles, customization and modularization are two complementary activities, having different goals and taking places at different stages of the ontology life-cycle. The modularization works at the design level whilst the customization works at the user level.

Chapter 4

Process for customization

The chapter 3 shows the need of customization. Indeed, the users need to see several views on an ontology without modifying it, according to their specific task. In this chapter, we will describe the process to get a customized view. The figure 4.1 shows the complete process to obtain this view. In this figure, data structures are illustrated through white boxes and process steps through colored boxes. The initial input of the customization process are ontology and user preferences, the final output is a customized view. First, the user must choose his preferences in term of customization. Afterward, this set of preferences is the input of the mechanism which will built a query and evaluate it to obtain the customized view, which is called "view generator ". Then, the user can choose to come back to step 1, "capturing user preferences", to make his choice more precise or/and to add more clauses. Finally, the user can manage those customized views. We will give more details all of this process steps respectively in the following sections User Preferences, Query Generator and in the View Manager.

4.1 User Preferences

Customization requires an ontology and user preferences as inputs to its processing. Either these are the preferences described in a view for which the further customization will be applied directly. Or, they exemplarily represent a type which requires a view generation method.

This step will give to an user the opportunity to choose how to customize the ontology. That choice corresponds to a specific template. As shown in the definition of a template, this mechanism gives to the user an easy way to generate a view. Several types of customization are possible, the following list is not exhaustive:

1. to prune $A \sqsubseteq B \bigcup | A, B \in classdescription$;
2. to extract every thing related to a description class;
3. to filter via labels or meta-information;

After choosing the kind of customization which influences how the view will be build, the user can focus on defining the content of it. Several choices are given to the user to fill the view as:

1. via an axiom ;
2. via a concept;
3. via an individual.

4.2 View Generator

The main goal of this step represented in the figure 4.1 by the number 2 is to generate a SAIQL query from the user preferences. To accomplish these two things are required as an input: the type of customization

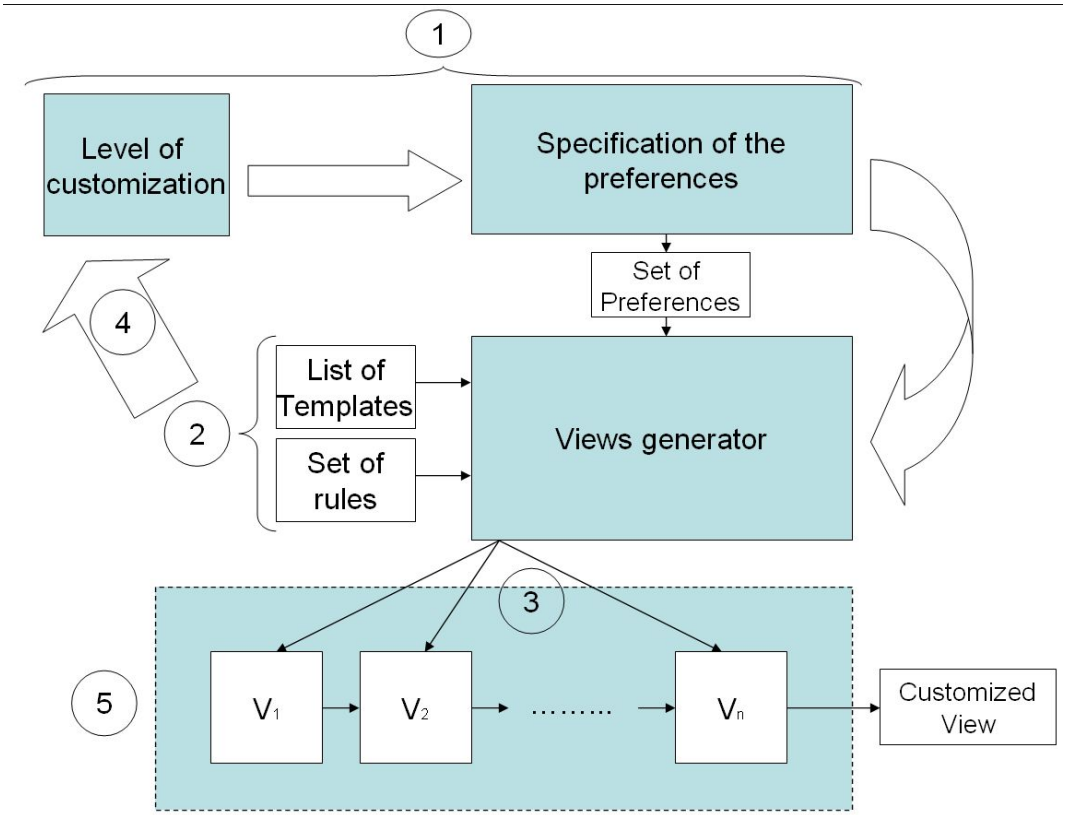


Figure 4.1: Customization Process

and the description of concept given by the user as explained in the section 4.1. The first input is needed to choose the right template to build the query. The second one represents the content of the query. From all this input given by the user, the system is able to generate a query which consists of several `WHERE` clause patterns. A `WHERE` clause correspond to the description of the customized view that the user would like to obtain. This clause is composed by several axiom pattern named `WHERE` clause patterns. Afterwards, the user can give a customized name to the set. This name is used whenever the user want to refer to the view in specific situations as querying for example.

4.3 Views Manager

It is this last step of the customization process which will extract the view from the original ontology. The current version of the plugin does not cope with the flexibility of the view. When the original ontology changes the view is not updated. The interactivity will be added in the future version of the plugin. Different ways are studied to do so, the most promising one will be to use the functionality of the workflow support plugin to track the changes in the original ontology. Afterward, we can check if those changes affect the view and how. Then two options are possible, the first will be to evaluate and extract the complete view. The second option will be to extract just the implication of the changes and replace it in the old version of the view. The second option is the more efficient, but it cannot be applied in every case.

The management of queries is the process, which allows the user to use as an input of another query, the result of a previous one. The stage number 4 represented in the figure 4.1, allows various other features such as sharing this view. The user may want to share the materialized view with a colleague . For the customization task, we think that it is crucial to have this kind of functionalities in the plugin. In addition, this step will give, to the user, the possibility to import/export the view definition.

Chapter 5

The Benefits of SAIQL

RDF-based query languages such as RQL[1], SeRQL[2] and the upcoming W3C standard SPARQL[PS], are defined on the notion of RDF triple pattern and their semantics are based on matching triple with RDF graph. From its conception SPARQL is not aware of the OWL semantic. The elaboration of a query for OWL DL ontologies becomes very cumbersome and sometime even impossible e.g. when you query a concept define as a intersection of cardinality restriction. We will show it using the simply use case of pharma-innova ontology presented in the section 5.1. Afterward we will explain how SAIQL can retrieve all the concepts which satisfy a query using OWL-DL constructors when in the same time SPARQL failed.

5.1 Simplify use case

To Illustrate the differences between those tow query languages we will use a sub set of the pharma-innova ontology. Below the reader can find a version of this sub ontology describe in the OWL abstract syntax describe in [PSHH]

```
EquivalentClasses(Invoice IntersectionOf( restriction (hasHeader cardinality(1))
 restriction (hasBody min(1))
 restriction(hasSumary cardinality(1)) ) )

Class(InvoiceFromProvider partial Invoice restriction(hasEmiter someValuesFrom(owl:Thing)))
Class(InvoiceForCustomer partial IntersectionOf( restriction (hasHeader cardinality(1))
 restriction (hasBody min(1))
 restriction(hasSumary cardinality(1))
      restriction(hasReceiver someValuesFrom(owl:Thing))) )
Class(Provider partial Thing)

Individual(i0 type(Invoice))
Individual(i1 type(InvoiceFromProvider))
Individual(i2 type(InvoiceForCustomer))
Individual(p type (Provider))
```

Query in natural language: “Retrieve all concept names and their descriptions and their instances, which are sub-concepts of *Invoice*!”

5.2 The limitation of querying using SPARQL

For instance, using SPARQL in our example use case from the above section, the class name *InvoiceForCustomer* could not be retrieved because it is not explicitly stated as a subclass of the class named *Invoice*. Even if we use an OWL reasoner such as Pellet to infer such a relation, there is no standard way to explicitly

[1] http://www.w3.org/Submission/RDQL/
[2] http://wwww.openrdf.org/doc/sesame/users/ch06.html

store the results of the inferencing in RDF. Additionally, there are ambiguous serializations e.g. for qualified number restrictions. Thus, in our use case the definition of the class named *Invoice* which is expressed as intersection of cardinality restriction is impossible to reach for any RDF query language like SPARQL.

5.3 The opportunities brought by SAIQL

As shown in the previous section 5.2 SPARQL syntax is build on the top of the RDF syntax which limits its usability to retrieve OWL-DI statement. From this constatation and to address this limitation we have build SAIQL on the top of the OWL-DL syntax. The main characteristic of SAIQL is its ability to retrieve and to allow query at the class description level. That is made possible because of its evaluation process which its explain bellow. Given the sub-ontology of pharma-innova presented in the section 5.1, the query in natural language is formulated. In the first step of the query evaluation, the `LET` clause is evaluated. Thereby, we extract:

- $N_C = \{$Invoice, InvoiceForCustomer, InvoiceFromProvider, Provider$\}$
- $N_I = \{i_1, i_2, i_3, p\}$
- $N_{CD} = N_C \cup \{$ restriction(hasHeader cardinality(1)), restriction(hasSummary cardinality(1)), restriction(hasBody min(1)), intersectionOf(restriction(hasHeader cardinality(1)), restriction(hasSummary cardinality(1)), restriction(hasBody min(1))), intersectionOf(restriction(hasHeader cardinality(1)), restriction(hasSummary cardinality(1)), restriction(hasBody min(1)), restriction(hasEmiter someValuesFrom(
owl:Thing)), restriction(hasReceiver someValuesFrom(
owl:Thing)), Provider, Invoice, InvoiceForCustomer, InvoiceFromProvider$\}$

Afterwards, the set of all possible solutions S_{all} is tested. As the `LET` clause contains a variable `?i` representing individual names, a variable `?X` representing class names and a variable `?Z` representing class descriptions, to compute this set we use the following formula $|S_{all}| = |N_I| \times |N_C| \times |N_{CD}|$ then the set of all possible solution look like the following set :
$S_{all} = \{$

[?i | i_1][?X | Invoice][?Z | restriction(hasBody min(1))]

[?i | i_2][?X | Provider][?Z | restriction(hasHeader cardinality(1))]

...

[?i | i_0][?X | InvoiceProvider][?Z | intersectionOf(restriction(hasHeader cardinality(1)), restriction(hasSummary cardinality(1)), restriction(hasBody min(1)))]

$\}$.
In the second step, the conjunction of the axiom patterns in the `WHERE` clause is evaluated. In our example, the first and the third single axiom pattern is a class axiom pattern and the second single axiom pattern is an individual axiom pattern. After checking the axiom patterns, $S_v \subseteq S_{all}$ is retrieved. A reasoner helps to check each axiom pattern. Each instantiated pattern is added to the original ontology afterward the reasoner control if the axiom keep the ontology consistent then it kept as a possible solution.
In the third and last step, the CONSTRUCT clause is evaluated. Each single axiom pattern of the CONSTRUCT clause is instantiated with each valid solution $s \in S_v$. Thus, the classes Invoice, InvoiceForCustomer and InvoiceFromProvider, their descriptions and the individuals i_0, i_1, and i_2 are inserted as axioms into a new OWL ontology that is shown in Figure 5.1.
We have shown in this section two main characteristics of SAIQL which make it unique. SAIQL syntax is build on the top OWL-DL, it allow you to query every concepts and individuals define by a intersection, restriction

```
Class(Invoice partial Invoice)
Class(InvoiceFromProvider partial Invoice restriction(hasEmiter someValuesFrom(Provider))
Class(InvoiceForCustomer partial Invoice restriction(hasEmiter someValuesFrom(Customer))

Class(Invoice partial IntersectionOf( restriction (hasHeader cardinality(1))
 restriction (hasBody min(1))
 restriction(hasSumary cardinality(1)) ) )

Class(InvoiceFromProvider partial IntersectionOf( restriction (hasHeader cardinality(1))
 restriction (hasBody min(1))
 restriction(hasSumary cardinality(1)) )
  restriction(hasEmiter someValuesFrom(Provider)))

Class(InvoiceForCustomer partial IntersectionOf( restriction (hasHeader cardinality(1))
 restriction (hasBody min(1))
 restriction(hasSumary cardinality(1)) )
 restriction(hasEmiter someValuesFrom(Customer)))

Class()

Individual(i0 type(Invoice))
Individual(i1 type(Invoice))
Individual(i2 type(Invoice))
Individual(i1 type(InvoiceFromProvider))
Individual(i2 type(InvoiceForCustomer))
```

Figure 5.1: Resulting OWL Ontology for the Given Query Example

or an enumeration. The second particularity come from the fact that SAIQL query and retrieve all instances of the class description from the OWL meta model. In other word SAIQL is able to retrieve the definition of class named where other query language ignore their existence.

Chapter 6

Plugin Architecture

The last chapter shows the process of using the ontology customization tool. From this process, a set of customization statements, called axiom patterns are defined. Profiling or "preferring" are two ways to make this set of axiom patterns. This set of axiom patterns represents all the content the user needs for his current task. It may be a laborious task to express by hand all SAIQL queries covering all syntactic expressions needed for generating such queries. Thus, we present an approach how the generation of customized views can be automated via several templates. This specific part of our framework is called "query generator". The query generator uses templates to generate the queries which are needed for the creation of customized views. The customization process is decomposed into two plugins as presented in the fig. 6.1. The first plugin integrated all the necessary components to interact with the user and generated the view definition, this is the customization plugin which will be presented in the section 6.1. The second plugin implements all the mechanisms to evaluate the view definition, this is the SAIQL plugin which can be use directly by an user via a the NeOn toolkit if the user wishes it.

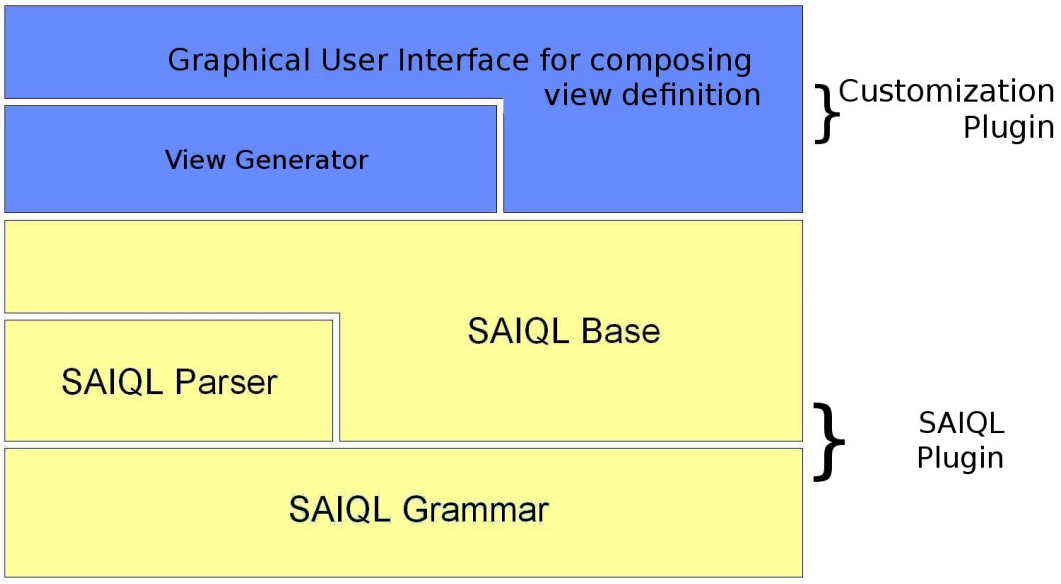


Figure 6.1: Relation between the customization plugin and the SAIQL plugin

6.1 Customization plugin, two components: Compositor of View definition and View Generator

The customization plugin is there to help the user of defining his customized view we develop a plugin on the top of the SAIQL plugin to guide the user throw the several step presented in the chapter 4. An important requirement was that the user of the plugin do not have to know the syntax of SAIQL. To fulfill this requirement and to make the user as comfortable as possible throw the customization process we propose as user interface a wizard with a little amount of information on each step (some screen-shot of this interface are shown at the end of deliverable in the user guide). Two other ways are currently exploring to extend this interface, in particular, for improving the sharing and the re-usability of query templates which take part of the customization process,. We are making available, directly from the customization plugin, the possibility of sharing and downloading template by using the ODP portal. We are also implementing a more user-friendly interface, base on browsing ontologies for defining the view.

Now we will explain how that was implemented in the following of this section. Important classes are described with their UML class diagram which are provided in support. The diagram is mainly there to show the hierarchy between the different classes and packages constituent the core of this customization plug-in. When the user is defining his view, no matters the way how he does it, by the templates or not, he will need two main components: the variables and the axioms pattern. The first one are describe in the diagram of the fig. 6.2 and the second one in the diagram of the fig. 6.3.

There is three kind of variables for ClassName, class description and individual. The user may want to describe expression which may contain concrete values for OWL-classnames and also for variable, representing OWL-classnames. From this class the classes ClassName and ClassNameVar are derived. The class ClassName is used for the names of OWL-classnames and accordingly the class ClassNameVar is used for variables.

Axioms pattern are axioms which content variables. This class and its specializations are use also to describe a OWL-Axiom thus for each Axiom-pattern there exists an corresponding OWL-axiom, where all variables are substituted by concrete values, i.e. class names, instances, etc.

6.2 Query Evaluation using SAIQL Engine

In the previous section we explain how is implemented the different steps of helping the user to define a view. While the view is define, the plugin have to evaluate this view to make it usable by the user. The evaluation is made by the second plugin: SAIQL. We will first explain the architecture of this plugin and then we will present the query evaluation which was introduced previously in 5.3

6.2.1 Architecture of the SAIQL Engine

The following section describes the architecture of the SAIQL query evaluation engine. The engine is composed of two main components :

1. the base which includes the query evaluating processes;
2. the grammar which represents all the components needed by SAIQL (cf. [BDS+08]);

The base implements all the classes to represent and to evaluate all the clauses of the query. The `WHERE` clause is represented by the class "WhereClause" which contents a set of axiom patterns. The "construct clause" is represented in the similar way by the class "ConstructClause". A class "SAIQLQueryEvaluation" evaluates all the query including an optimization round by sorting the "where clause" with the most discriminant first. The package contains classes to read a query from a text format and transform it to the SAIQL internal representation.

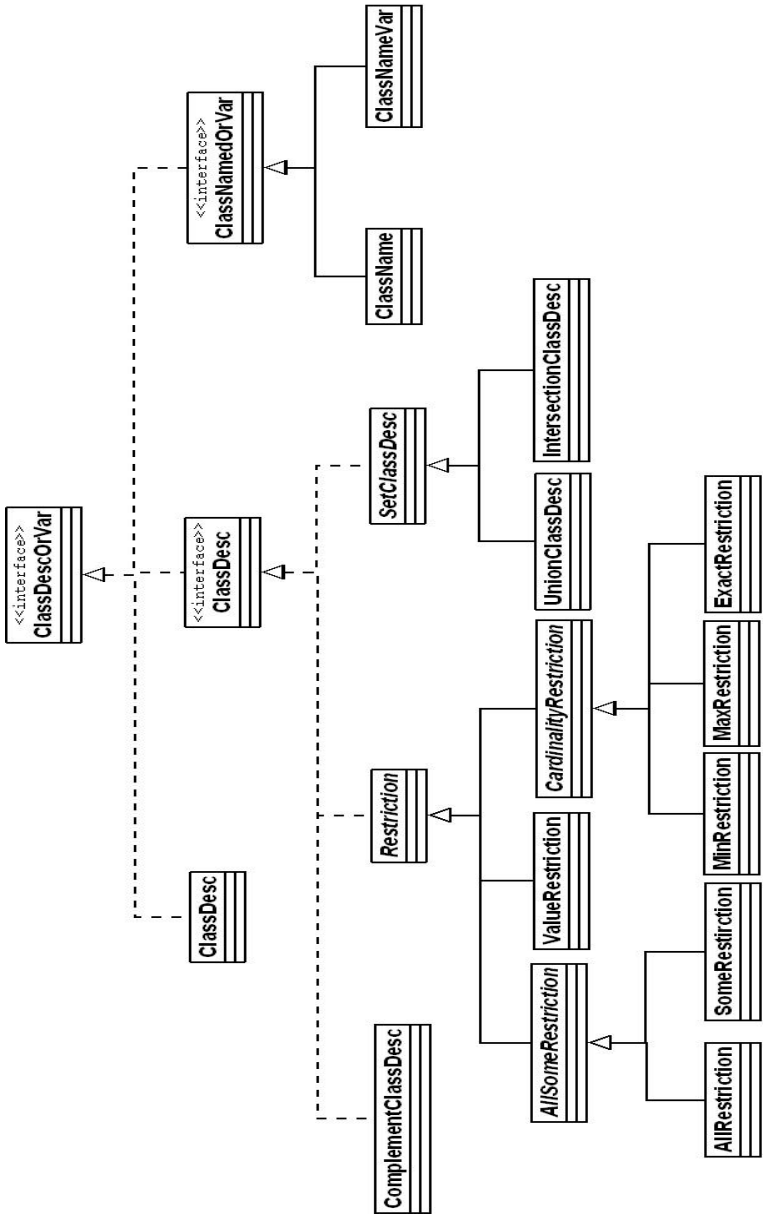


Figure 6.2: Diagram of classes which represent an "Description or Variable"

The grammar implements all the classes and interfaces to represent all the components of a SAIQL query: clauses, axiom pattern, axioms, ... The internal representation is close to the representation we use for the customization plugin (cf.6.3)

6.2.2 Query Evaluation Strategy

In the following we describe the evaluation strategy for OWL-SAIQL queries. The query evaluation consists of three steps. In the first step the `LET` clause is evaluated: From the ontology O declared in the `FROM` clause we retrieve three sets of ontology elements by syntactically parsing the ontology, namely the finite set of class named N_C, the finite set of class descriptions N_{CD}, the finite set of individual-valued property named N_{IP} and the finite set of individual named N_I. The finite set of class names N_C contains the names of all atomic classes and the names of all named complex classes. Additionally, the finite set of class descriptions N_{CD} contains all class descriptions that appear in the concrete syntactic notation of O (note that this includes all class names as well). Thus, $N_C \subseteq N_{CD}$.

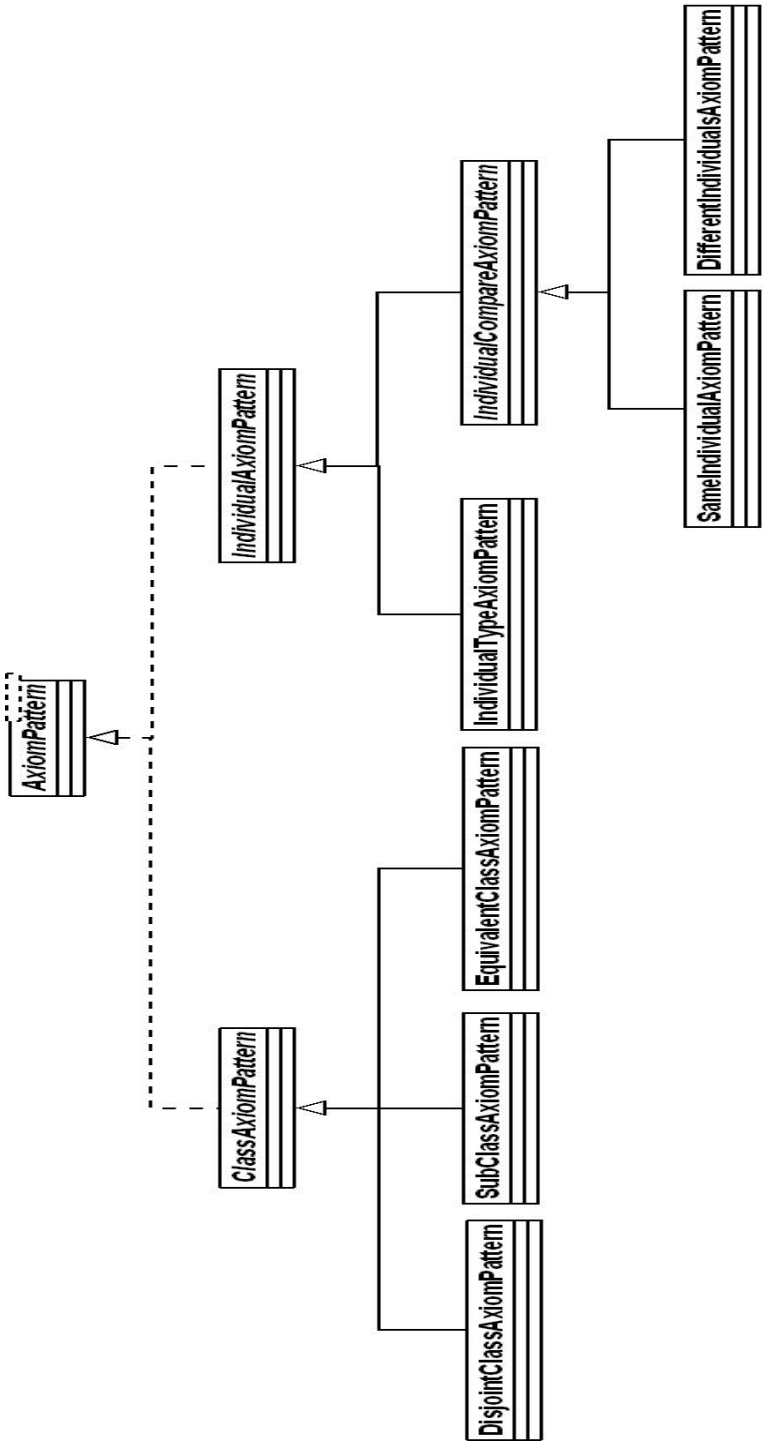


Figure 6.3: Diagram of classes which represent an "Axiom Pattern"

The range of every variable is bound to one of these sets, as declared in the `LET` clause.

In the second step the `WHERE` clause is evaluated. Each axiom patterns from the `WHERE` clause are sorted in terms of "evaluation cost" from the lower to the higher time consumer. At this stage, the sorting only takes into account the number of variable content in the axiom pattern. Afterwards, the set of all syntactically possible solutions S_{all} is created by building the Cartesian product involved in the first pattern. The conjunction of the axiom patterns in the `WHERE` clause is instantiated with each solution $s \in S_{all}$ and, then, it is decided for each solution s if it is valid or not. Each valid solution s is added to the set of valid solutions S_v. After each evaluation of a pattern S_{all} is replace with S_v. This step is suitable for further optimizations like tree index

access [d'A07].

In the third and last step, the `CONSTRUCT` clause is evaluated and the result of the query is generated. Therefore, the axiom patterns in the `CONSTRUCT` clause are instantiated with each valid solution $s \in S_v$ and, thus, new axioms are created. The result of the query evaluation is a new set of axioms, i.e. a new ontology.

6.3 Experimentation and Evaluation

At this stage, the prototype was tested on three platforms supported by the NeOn toolkit (Windows, Mac OS, Linux). Those tests were made to ensure the compatibility of the plugin with all version of the toolkit. This plugin was tested on the version 1.2.0 and the latest version 1.2.2 of the Neon toolkit. Another review of the plugin was made by two of our partners Joan Candini form Kin laboratorios which represents the final user and Dr. Martin Dzbor form Open University which has an expertise in user expectation and even more so he is a developer himself. It is important to have two different visions for the review, one from the user perspective and the other one from the developer. The first one focuses on the global functionality aspect and the visual aspect unlike the developer which will focus on finding bugs and paying less attention to the way of getting the result.

Another round of experiments with end users from Kin is in plan. We will give the plugin to several users from different departments of Kin laboratorios, peoples form the sales department and from the financial department. Each user will get a short overview of the plugin, how to use it, what its capabilities are and some examples of use related to the task of the tester. Then the user will have few weeks to use the plugin in his current task. Afterwards, he will have to answer to an questionnaire to get his feed back.

Another experiment with FAO is in discussion. This phase of test will target another aspect of the customization tool; the possibility to customize an ontology via the language of the user. For example, this functionality allows the user to display only the label of concept and property in his own language or combine languages. This adaptation of the tool has two advantages, first, the user sees only the right amount of information needed. Second, the ontology is getting smaller in size, so easier to manipulated (like querying). AGROVOC with 7 languages allocates 130MB whereas the same ontology with 2 languages English and French represents only 53MB.

It is not easy to measure the quality of the customization process because it is closely related to the user expectation. One way to evaluate the result of the customization process is to get the feed back directly from the user.

Chapter 7

Conclusion

In this deliverable we have presented two plugins: the SAIQL plugin and the ontology customization plugin. The SAIQL plugin proposes a novel manner to query an ontology to the user of the NeOn toolkit. This new language aims at solving queries from the TBox and the ABox at one time. The ontology customization plugin gives the possibility to generate a customized view following the preferences of the user.

In this deliverable we pointed out the orthogonality of the notion of ontology customization compared to the notion of ontology modularization. We showed that those notions can share similar methods to generate a customized view or a module, but that each of them is used at different stages of the ontology life cycle.

This initial prototyping has enabled us to gain valuable lessons to shape further research on using query-based ontology customization techniques. Further functionalities have to be implemented such as the customization via labels or annotation properties. This technique will bring the possibility of customizing following the natural language.

For our future work, we are studying the eventuality to use the workflow support developed in the WP3 to track the ontology changes to update all or parts of the customized view whenever a change is made in the original ontology.

As a second step, we aim to extend this work on customization to the notion of trust and access right. The main goal of this work will be to generate customized views following the access right of the user.

Appendix A

Customization Plugin Manual

A.1 Functional Description

The main functionality of this plugin is to give a tool to the user which has an user friendly interface to generate customized view.

A.2 How to install it?

For manual installation, download the latest version of the plugin with all the dependencies (cf. the list bellow) at the following address (http://www.uni-koblenz.de/ bercovici/Plugin/). After downloading, you have to copy the java archive (.jar) in the folder "Plugins" located in the NeOn toolkit's directory. After manually installing the plugin you have to restart the NeOn-Toolkit.

A.2.1 Dependencies

For the customization plugin v0.3

org.neontoolkit.saiql0.4

org.neontoolkit.owlbridge.1.0.1

Test for the toolkit version:

windows 1.2.1 & 1.2.2

linux 1.2.1 & 1.2.2

A.3 User Documentation

The plugin is visible at one location in the toolkit as a new view. To make the view appears you have to go to the menu "Windows>Show View>OtherĚ" then a pop up window show up. The view is located in the section "Other>Customization".

A.3.1 Select an Ontology to custom

The selection of an ontology is the first step to custom. First you have to refresh the list of the project as well as the list of the ontology if the project or the ontology wanted does not appear in those lists.

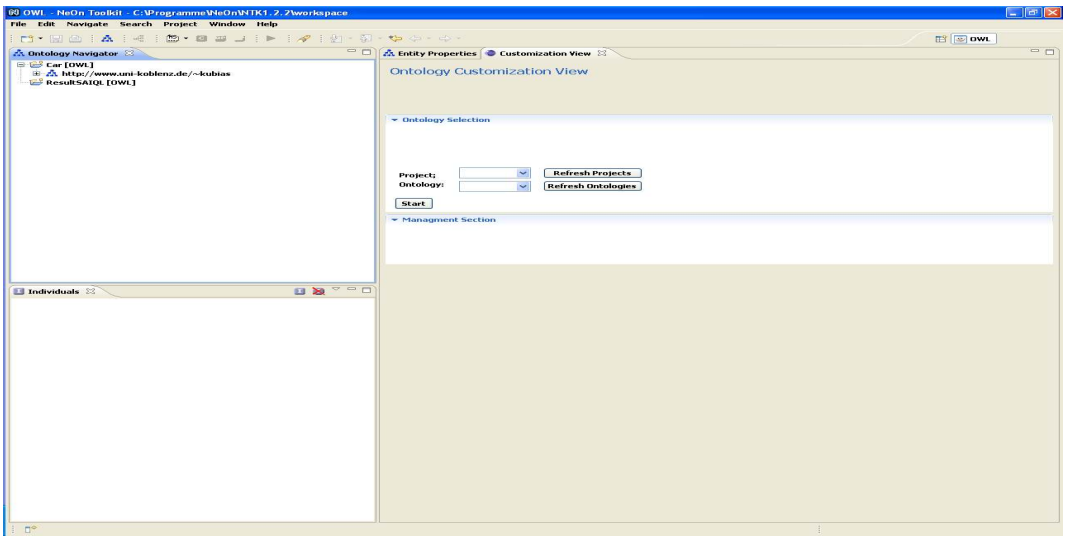


Figure A.1: Choose Ontology

A.3.2 Start the capture of the preferences

After the selection of ontology you can press the button "start" to start the capture of your preferences. Then a wizard appears as in the figure 2. It gives you the choice between three ways of customizing an ontology:

1. via a class named
2. via a description
3. via an instance

A.3.3 Manage the view

In the section "Management" you can do several actions:

1. generate the view
2. share the view
3. modify the view through the wizard or not
4. delete the view

A.3.4 Generate a view

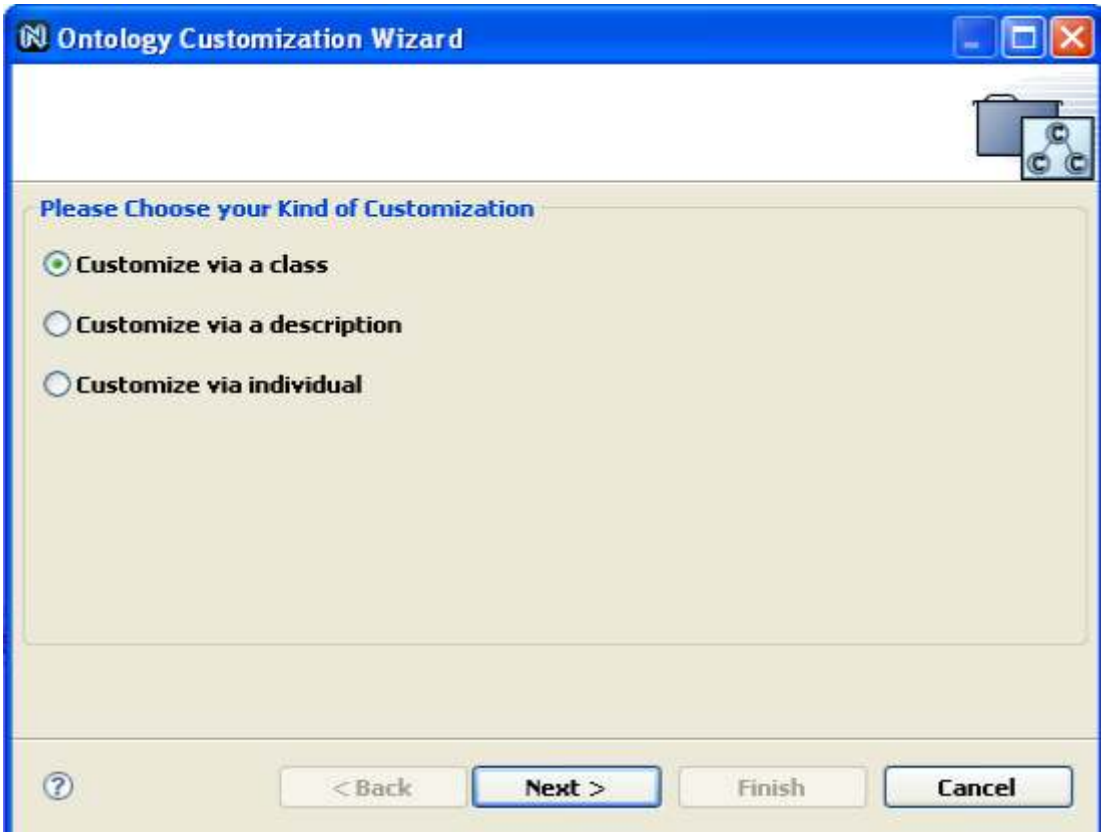


Figure A.2: the first wizard for customization

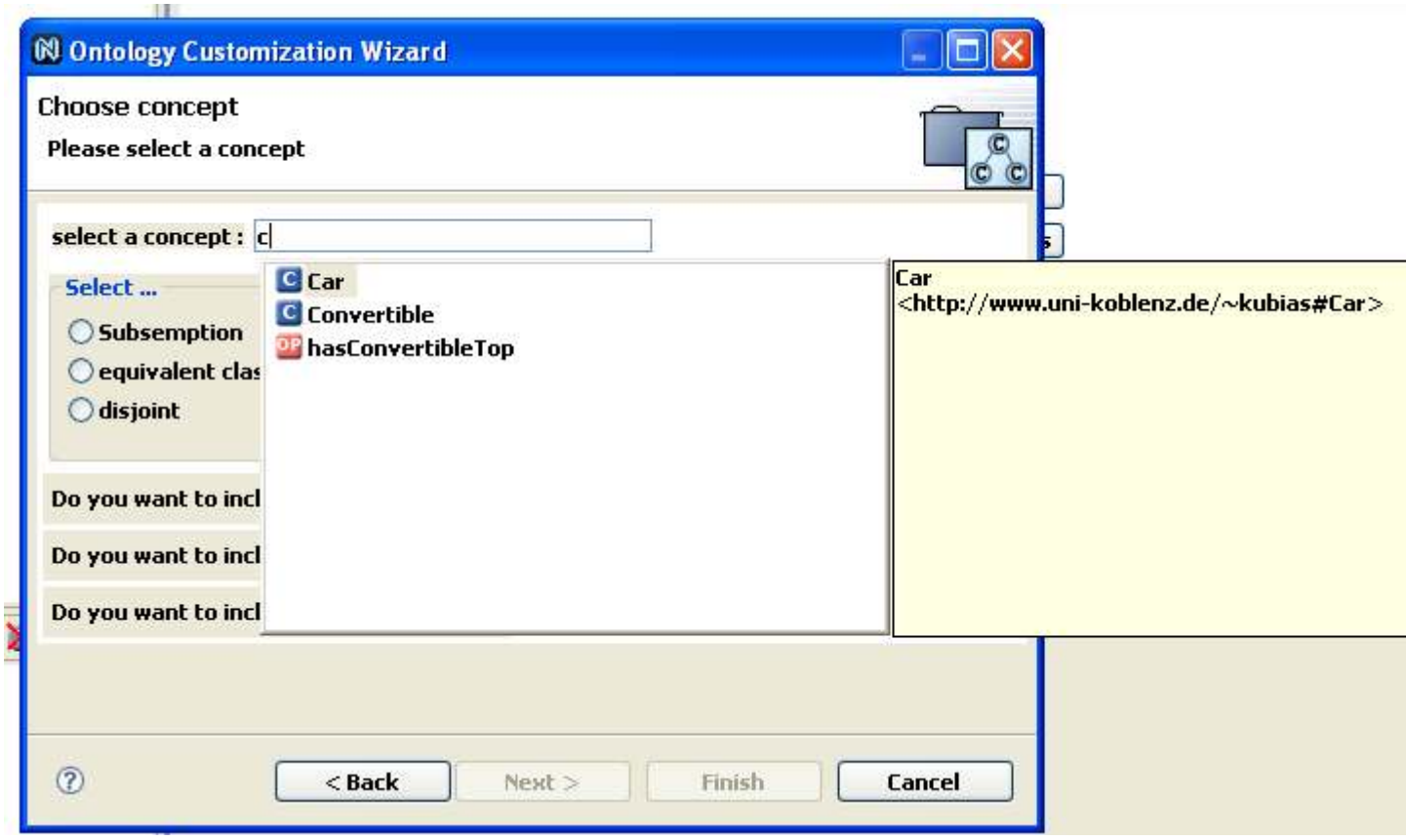


Figure A.3: Customization via a class named

Figure A.4: Customization via an instance

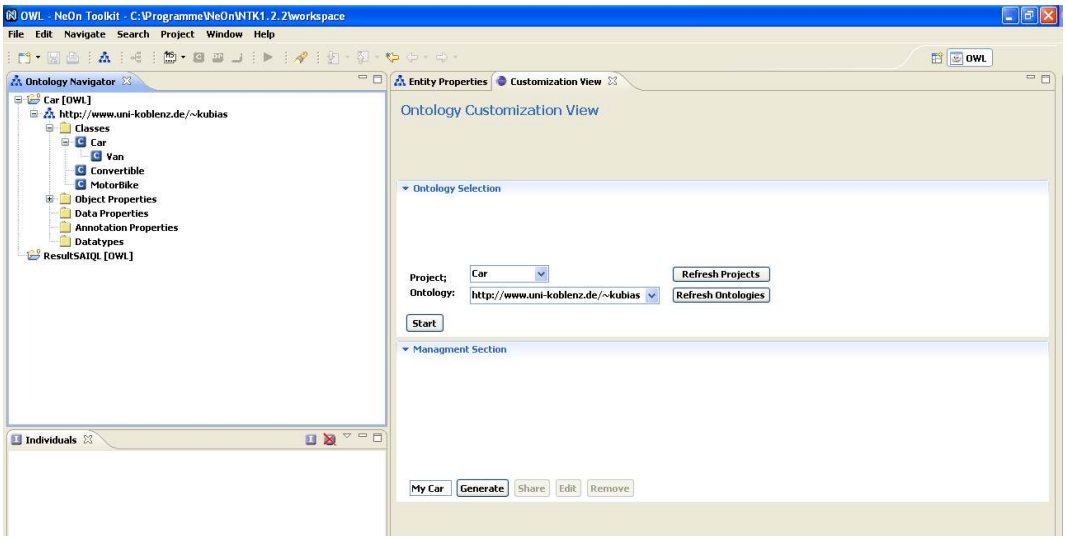


Figure A.5: Management Section

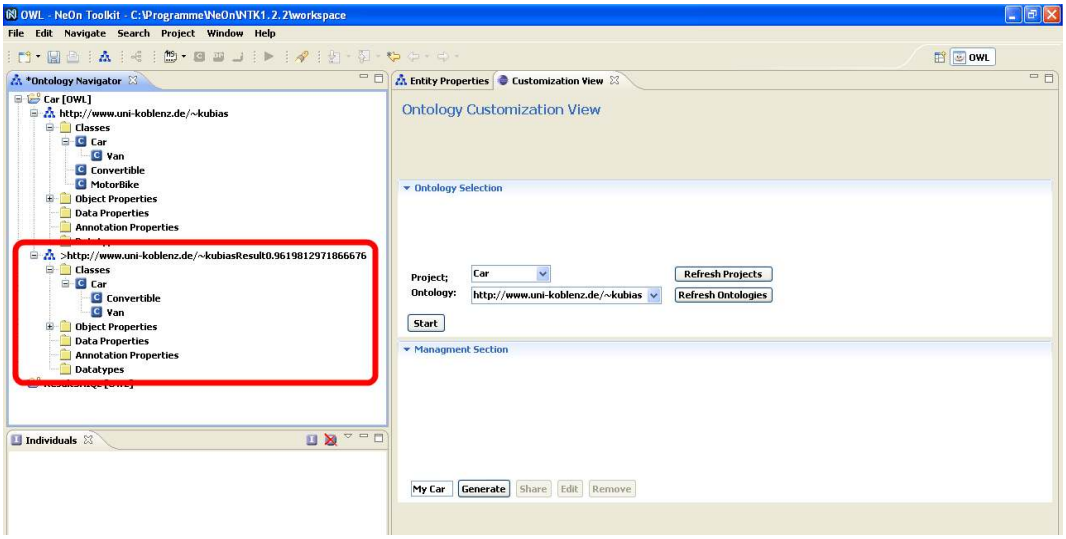


Figure A.6: Generate a customized view

Appendix B

SAIQL Plugin Manual

B.1 Start of the Application

After starting the NeOn toolkit, it should look like the following figure. B.1.

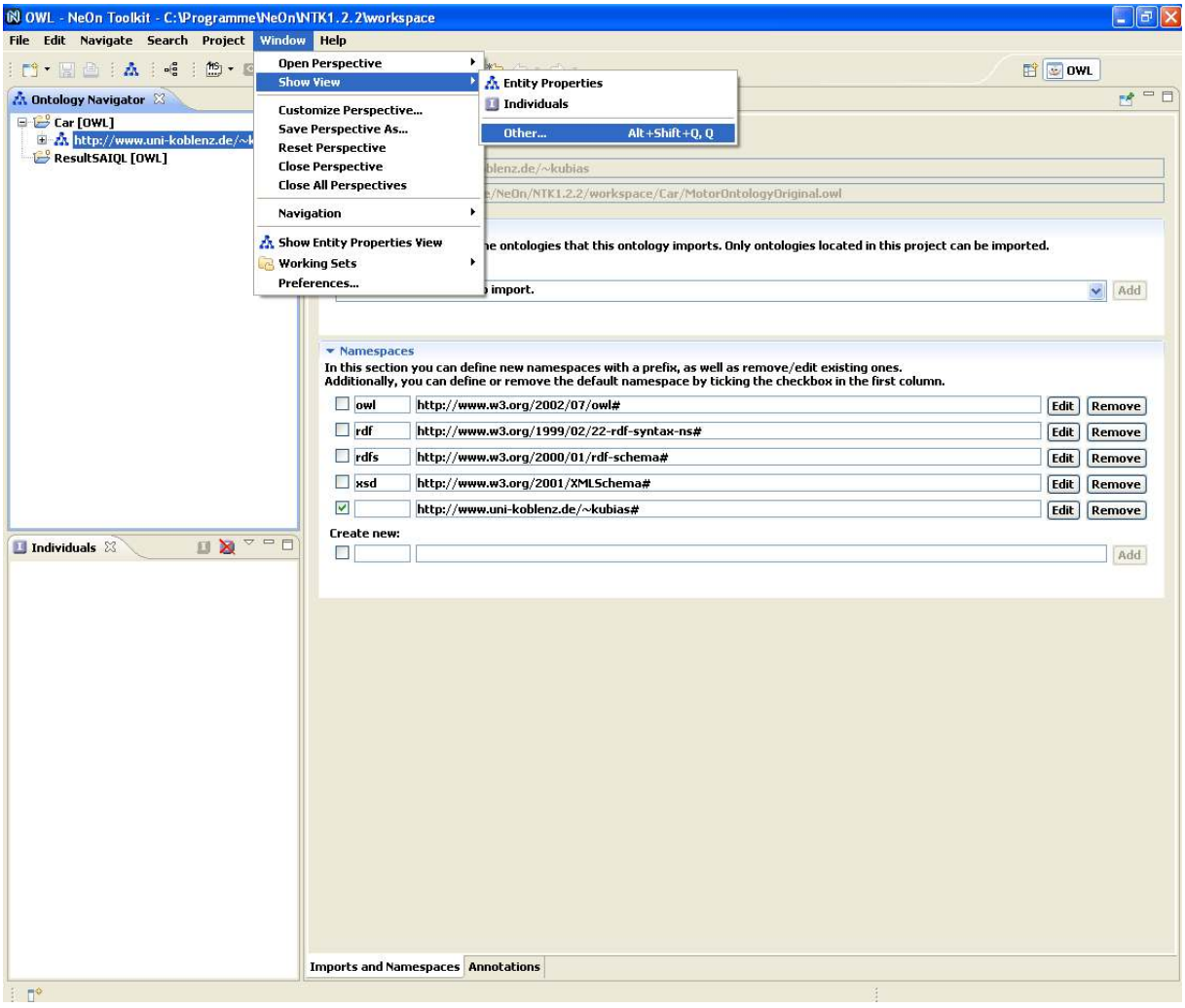


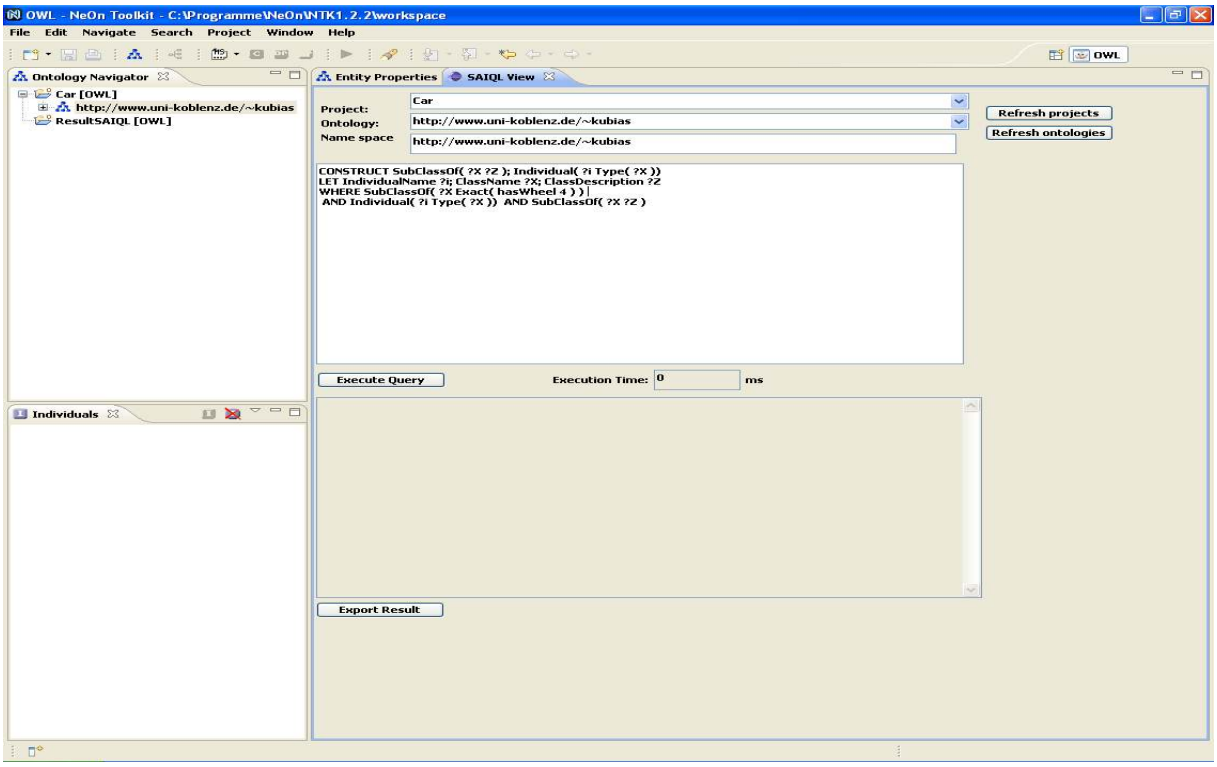
Figure B.1: The main window of the NeOn toolkit

B.2 How display the SAIQL view in the toolkit

Click on window>Show view>Other... as shown in the figure.After the figure has appeared, select, then you should select the "SAIQL View"in the section query engine.

B.3 Choose Project and Ontology to Query

First you should select the project which contains the ontology you want to query. For that, you can use the list which contents all the OWL projects of the toolkit as show in the figure B.3, if the list is empty please click on the refresh project. After, you have to select the ontology using the same process. Note: the field namespace is automatically completed with the namespace of the ontology but it is editable if you need to change it.



B.4 Entering the SAIQL Query

The next step is the input of the SAIQL Query in the second Memo-Textfield, it is located just under the field of the namespace.

Note: a basic query is already pre written to help you to deal with the syntax. This query correspond to our example presented in the paper. This query **HAS TO BE CHANGED**.

```
OWL-SAIQL-query ::= 'CONSTRUCT' constructClause
                    'LET' letClause
                    'WHERE' whereClause

constructClause ::= axiomPattern {';' axiomPattern}
letClause ::= variableBinding {';' variableBinding}
whereClause ::= axiomPattern {'AND' axiomPattern}

axiomPattern ::= classAxiomPattern | individualAxiomPattern

className ::= URIreference individualName ::= URIreference
ontologyID ::= URIreference indProperty ::= URIreference

variableBinding ::= classNameBinding
   | individualNameBinding
   | classDescriptionBinding
   | indPropertyBinding

classNameBinding ::= 'ClassName' classNameVar {',' classNameVar}
individualNameBinding ::= 'IndividualName' individualNameVar
   {',' individualNameVar}
classDescriptionBinding ::= 'ClassDescription' classDescriptionVar
   {',' classDescriptionVar}
individualPropertyBinding ::= 'IndividualProperty' indPropertyVar
   {',' indPropertyVar}

lexicalForm ::= a unicode string in normal form C
classNameVar ::= '?'lexicalForm
individualNameVar ::= '?'lexicalForm
classDescriptionVar ::= '?'lexicalForm
indPropertyVar ::= '?'lexicalForm

classNameOrVar ::= classNameVar | className
indNameOrVar ::= individualNameVar | individualName
classDescOrVar ::= classDescVar | classDesc

classAxiomPattern ::=
   'SubClassOf(' classDescOrVar classDescOrVar ')'
   | 'DisjointClasses(' classDescOrVar classDescOrVar ')'
   | 'EquivalentClasses(' classDescOrVar classDescOrVar ')'

classDesc ::= classNameOrVar
   | restriction
   | 'UnionOf(' {classDescOrVar } ')'
   | 'IntersectionOf(' { classDescOrVar } ')'
   | 'ComplementOf(' classDescOrVar ')'

restriction ::= 'All(' indProperty classDescOrVar ')'
   | 'Some(' indProperty classDescOrVar ')'
   | 'Value(' indProperty indNameOrVar ')'
   | 'Min(' indProperty non-negative-integer ')'
   | 'Max(' indProperty non-negative-integer ')'
   | 'Exact(' indProperty non-negative-integer ')'

individualAxiomPattern ::=
   'Individual(' indNameOrVar 'type(' classDescOrVar ')' ')'
   | 'SameIndividual(' indNameOrVar indNameOrVar ')'
   | 'DifferentIndividuals(' indNameOrVar indNameOrVar ')'
```

B.5 Evaluation of the SAIQL Query

The next step is the evaluation of the SAIQL Query. Use the button “Evaluate Query”. The evaluation process can take some time.

B.6 Display of the extracted ontology

The extracted ontologyŞ is shown in OWL Abstract Syntax in the Memo-Textfield which is not editable.

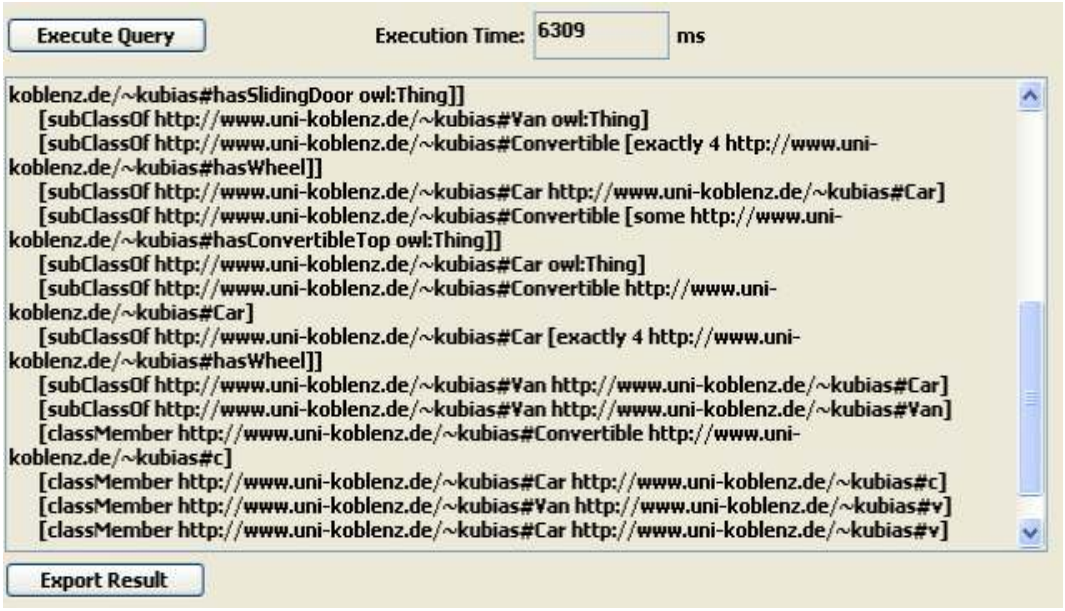


Figure B.2: The window with the result of SAIQL

B.7 Export extracted Ontology

The extracted ontology could be extracted to a Owl project which already exists in the toolkit, using the "Export Result "button.

Bibliography

[BDS+08] Noam Bercovici, Martin Dzbor, Simon Schenk, Alexander Kubias, and Gerd GrÃűner. Ontology customization and module creation: query-based customization operators and model. Deliverable D4.2.2, NeOn Project, 2008.

[d'A07] C. d'Amato. *Similarity-based Learning Methods for the Semantic Web*. PhD thesis, University of Bari, 2007.

[DDM+07] Klaas Dellschaft, Martin Dzbor, Dunja Mladenic, Alexander Kubias, Carlos Buil Aranda, and Jose Manuel Gomez. Review of methodds and models for customizing/personalizing ontologies. Deliverable D4.2.1, NeOn Project, 2007.

[dHR+07] Matthieu d'Aquin, Peter Haase, Sebastian Rudolph, Jérôme Euzenat, Antoine Zimmermann, Martin Dzbor, Marta Iglesias, Yves Jacques, Caterina Caracciolo, Carlos Buil Aranda, and Jose Manuel Gomez. Neon formalisms for modularization: Syntax, semantics, algebra. Technical report, The NeOn Project, 02 2007.

[DKG+07] Martin Dzbor, Alexander Kubias, Laurian Gridinoc, Angel Lopez-Cima, and Carlos Buil Aranda. The role of access rights in ontology customization. Deliverable D4.4.1, NeOn Project, 2007.

[DMG+06] M. Dzbor, E. Motta, J. M. Gomez, C. Buil Aranda, K. Dellschaft, O. Grlitz, and H. Lewen. Analysis of user needs, behaviours & requirements wrt. user interfaces for ontology engineering. Deliverable D4.1.1, NeOn Project, 2006.

[Dzb09] Martin Dzbor. Realization of a prototype extension for access control in neon infrastructure. Deliverable D4.4.2, NeOn Project, 2009.

[Gan05] Aldo Gangemi. Ontology design patterns for semantic web content. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *Proceedings of ISWC'05: the 4th International Semantic Web Conference, Galway, Ireland, November 6–10, 2005*, volume 3729 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2005.

[GPBH+07] Jose Manuel Gomez-Perez, Carlos Buil, German Herrero, Tomas Pariente, Angel Baena, Joan Candini, and Juan Carlos Dalmacio. Ontologies for the pharmaceutical case studies. Deliverable, Intelligent Software Components (iSOCO), 08 2007.

[GPDM+06] Jose Manuel Gomez-Perez, Claire Daviaud, Berta Morera, Richard Benjamins, Tomas Pariente Lobo, German Herrero, and Gloria Tort. Analysis of the pharma domain and requirements. Deliverable, Intelligent Software Components (iSOCO), 09 2006.

[ICJ+08] Marta Iglesias, Caterina Caracciolo, Yves Jaques, Margherita Sini, Francesco Calderini, Johannes Keizer, Fynvola Le Hunte Ward, Malvina Nissim, and Aldo Gangemi. Wp7 user requirements. Deliverable, FAO, 09 2008.

[KS03] J. Komzak and P. Slavik. Scaleable GIS data transmission and visualisation. In *Proceedings of the International Conference on Information Visualization (IV)*, 2003.

[LTD06] James F Brinkley Landon T Detwiler. Custom views of reference ontologies. In PubMed, editor, *American Medical Informatics Association Fall Symposium*, volume 2006; 2006: 909, 2006.

[LTDF07] James F Brinkley Landon T Detwiler and James F. Querying non-materialized ontology views. In PubMed, editor, *American Medical Informatics Association Fall Symposium*, 2007.

[PS] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C. http://www.w3.org/TR/rdf-sparql-query/, October 2006.

[PSHH] P. Patel-Schneider, P. Hayes, and I. Horrocks. Web Ontology Language (OWL) Abstract Syntax and Semantics. http://www.w3.org/TR/owl-semantics, February 2003.

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