UiO : Department of Informatics
University of Oslo

Investigating performance of complex data in Semantic Web

Comparing two different ontology approaches

Magnús Dæhlen

Master's Thesis Spring 2014

Acknowledgement

I would like to thank my supervisors Kjetil Kjernsmo and Martin Giese for their support and guidance during my thesis. I also want to thank Daimler, especially Kai Holzweissig, for contributing with valuable data. Without these data the thesis would have been difficult to realize. In the end I want to thank my family and friends for motivation and help along the way.

Abstract

The internet is becoming a marketplace for more and more products. With this in mind there will soon be possible to buy and sell any kind of product with component constraints, in the same manner we order a personalized computer. Today there is not done enough research within this area. The problem is how users and manufacturers should share their data to increase the information flow on the web. This will be even more important when more complex data are uploaded to the web. In this thesis we study how semantic web can be used to represent complex data with component constraints on the web today. This will be done by testing a product specific ontology against a generic ontology. To do this we have used an application which represented a real life scenario to help us test the two different ontology approaches. We have used a domain about car models from both Renault and Daimler to be able to experiment the two approaches. The experiments were conducted using the approach *Design of Experiments* (DoE) to investigate what factors would influence the performance of the two approaches.

We will discuss several aspects of the different approaches, emphasizing the definition of the ontologies and the representation of the complex data. This discussion will be based on the experiments done against the application. In the end we will present the most important elements when representing and structuring complex data in semantic web.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

In society today it is more and more common to browse and buy products over the internet. Until now the products available for sale are often defined products with none or a small amount of specifiable content. With the internet taking over more and more of our shopping habits it is natural that soon more complex products can be bought or ordered online. This increases the complexity when users get the option of ordering new products with several component constraints. In order to describe such products, there is a need to represent the models in a structured way.

A complex product which can be bought online today are computers. Several vendors offer the possibility of specifying the computer down to the smallest detail. One can say that the personal computer is getting even more personal. Even automobiles can be specified and ordered online from the web pages of each individual car manufacturer, but what if one wants to compare cars from different manufacturers? Today one would have to check each site and do a manual comparison. A solution for a third party site is to make a car comparing tool, but to do this effectively they need data.

This increases the demand for trustworthy information and the optimal source of such information is the manufacturers. Semantic web can be an easy and comprehensive way of exchanging valid data between sources. The problem today is that there are no best practice on how to represent and store complex data with semantic web.

1.2 Approach

An approach to this problem is using semantic web and creating structured ontologies. With semantic web the manufactures can post their data and the vendors can easily extract the data needed. Today there are several internet vendors which utilize semantic technologies to enhance their data,

for instance Rakuten.de. [1] They are using GoodRelations, a web vocabulary for E-commerce, to represent and enhance the information around their products. This also opens up for several ways of representing data about more complex products. Products like this often consists of more than one component and the components can have constraints between them. The use of semantic technologies on these kinds of products are a quite new notion, but a field with endless possibilities.

This is why we propose a web application which utilizes complex data from different data sources. Our application will be the base of a recommendation on how data about complex products should be represented and handled.

1.3 Goal

The goal of this thesis is to present a proposal on how to represent complex data in semantic web. Today there are several ways to represent data with semantic web. In this thesis we will present two different approaches and compare them to find their strengths and weaknesses. Each approach contains an ontology and has its own way of representing data. To make this comparison we have used data about the same domain, data about car models and their component constraints. The first approach is a generic ontology. By generic it means that it can be used to represent any product model with component constraints. The second approach is a domain specific ontology which as the name implies, is only applicable with one particular domain. In this thesis that domain is about car models.

We will also show how to create a viable web application which utilizes such complex data.

1.4 How to evaluate

We will in this thesis do several performance tests on the prototype to determine the weaknesses of each ontology approach. This will be done with complex data found on the web today from different car manufacturers. With performance we mean the response time between a HTTP[2] post against the application and when the application presents the user with an answer. The application will contain the possibility to do HTTP posts against both approaches.

There will be several options on how to query the ontologies because of all the different specifications. That is why we have chosen to use the testing approach *Design of Experiments* (DoE). This approach will be further explained alongside the results in Chapter 5. The evaluation will be based on four experiments testing several aspects of the approaches. They will help us determine what kind of factors are significant to the performance.

[1] Rakuten.de is a leading German online mall, aggregates 6500 merchants with more than 16 million item pages [1]

[2] The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. [19]

1.5 Previous work

Complex products and specifying configurations has been a research topic for over a decade. The possibility to personalize more and more products is why several research articles have proposed different approaches on how to handle these configurations. Most of the research are around finding the ultimate solution with a specification system. In 1999, M. Aldanondo et. al proposed how to structure a system to handle configurations and their constraints. This included proposed definitions for products, configurations and configurators. The paper focused on making a generic solution to fit several manufacturers. [35] In a newer article, H. Afsarmanesh and M. Shafahi (2013) proposed a complex product specification system. [10] This include object modelling and a user interface. They have focused on supporting stakeholders in the specification process.

Unfortunately these articles do not present any research done with semantic technologies. The use of semantic technologies on complex products is a young field of research. The only thing done here is what Renault and Volkswagen have presented. Both of these car manufacturers have presented the public with two different solutions. They have also shown how to present the data on the web and the complexity around their solutions.

Chapter 2

Background

2.1 History of semantic web technologies

2.1.1 The beginning

The definition of semantic "is the study of meaning". The science of studying semantics has existed longer than computer science. Still semantics have played an important role in the history of computer science, which is still a young science.[1]
When semantic web was first suggested, the World Wide Web (WWW) was mostly designed so that humans could read and understand the content. The semantic web was then described as an extension of the WWW so that machines could more effectively and intelligently search the web for data. It was not supposed to clean up and restructure the WWW in an instant, but it was suggested as a new technology which would improve communication and globalization of the web. It was going to be so easy that almost every web page with any user communication would benefit from it. In "The Semantic Web" Hendler et al. even said that even an office manager with no prior computer science background could easily encode the semantics into a web page by using "off-the-shelf software for writing Semantic Web pages". [41] They visioned it to be so easy that even a doctor's office could implement it, but they also knew there would be challenges. Even from the beginning Tim Berners-Lee and the other people working on "The Semantic Web" project, knew that the biggest challenge would be to make everybody communicate in the same understanding of the world.

2.1.2 Today

Today the semantic web technologies have evolved quite much from the start, but maybe not as much as Tim Berners-Lee had hoped. Figure 2.1 on the following page shows some of the steps, frameworks and publication done since the beginning. It is even a little outdated because there have been published a lot of new research the last few years, like SPARQL 1.1 which became a recommandation 21th of March 2013. Many companies

[1] Compare it to philosophy which can be dated as far back as 2880 BC. [17]

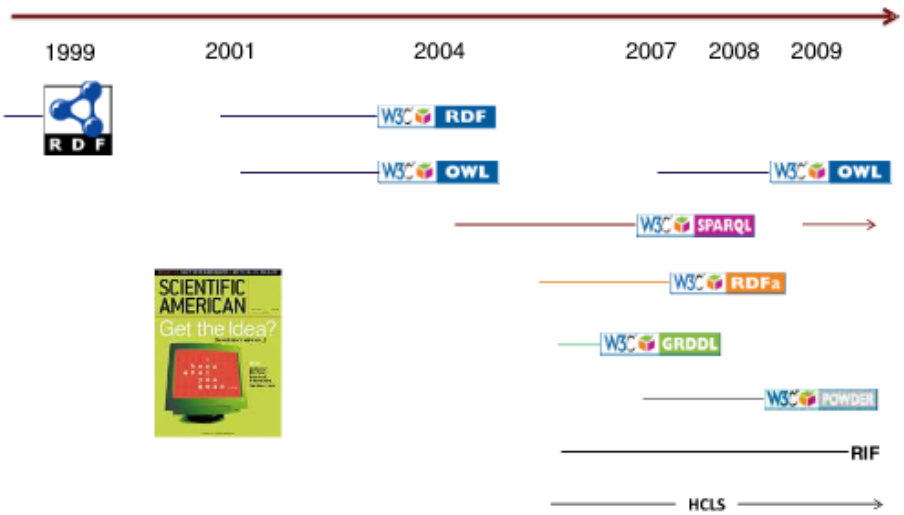

Figure 2.1: The semantic web timeline

and organizations have adapted the semantic model and are using it for different purposes. For instance is Google using semantic web for additional information to a search besides the regular word search. An example would be to do a search on the word 'Reagan'. Google then presents the regular search results with links to different sites, giving us a Wikipedia page about Ronald Reagan as the top result. The additional information that Google gives us is a small information tab about Ronald Reagan. This information tab included links to other American presidents and several of the movies Ronald Reagan starred in. To acquire this information Google uses their *Knowledge Graph*. It uses semantic web to enrich data and be able to give the user links to useful information linked to that particular entity.

It is only in recent years that large companies, such as Google, have begun using Semantic Web actively to improve the user experience and their information exchange. There are still some of the same issues today as it was in the beginning of Semantic Web. One of the major issues with the semantic web is trust. Even though there are plenty of standards and frameworks for semantic web, which will be explained in further detail later, there will always be a little uncertainty whether one can trust one another on the web. It is the same issue Wikipedia face every day. Anyone can say anything about everything, there is no information control except manual labour. This means that if a company decides to use semantic web, they have to know which sites are reliable. Even if a site is reliable how do one know that the semantics on that site has the same meaning as your semantics and how can one link seemingly disjoint information together. That is why ontologies have become an important part of sharing data.

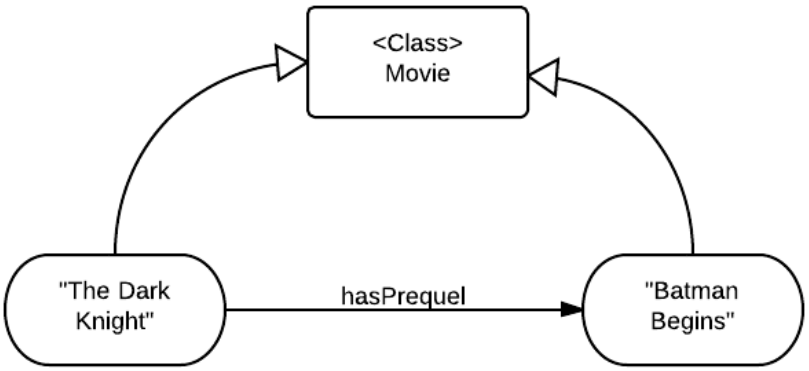

Figure 2.2: Small ontology example

2.2 What is an Ontology?

There are several definitions of what an ontology is and some of them contradicts each other. In this thesis we will use Stanford's definition of an ontology. An ontology is a formal explicit description of concepts in a domain of classes. [37] An ontology also contains properties describing various features, attributes and relations concerning the classes. The purpose is to represent a set of individual instances with the ontology and to enrich the meaning of the data. The ontology together with the set of data is often referred to as a *knowledge base*. These steps are done when developing an ontology:

- Defining classes in the ontology
- Arranging the classes in a taxonomic (subclass - superclass) hierarchy
- Defining relations and describing allowed values for these relations

In the end when one got a fully functional ontology the last step is to fill it with individual instances. In Figure 2.2 there is an example of how a small ontology is used. The ontology has one class, *Movie*, and one property, *hasPrequel*. In the example it is also added two instances.

Why do we need ontologies?

There are several reasons for why we need ontologies. Beneath some of them are presented:

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

- To analyze domain knowledge

The first two reasons are key concepts of semantic web, to share data and information on the web. This helps with identifying information and defines the meaning behind this information. Making explicit domain assumptions means that there is no ambiguity about what the data states. This also helps when there has to be done alterations in the domain specifications. Analyzing domain knowledge is possible once a declarative specification of the terms is available. Formal analysis of terms is extremely valuable when both attempting to reuse existing ontologies and extending them (McGuinness et al. 2000). In this thesis it was very valuable for analysing the ontologies and use them correctly in the application.

2.3 Frameworks and technologies

It has been done a lot of work around the semantic web since the beginning. In this chapter we will present frameworks and technologies which are useful for this thesis.

2.3.1 Resource Description Framework

Resource description framework (RDF) was introduced in 1999 as a foundation for processing metadata by the World Wide Web Consortium. [34] It had the intention to improve the communication and information exchange between applications on the web. In the beginning one could only write RDF in XML syntax. Later it has been developed several syntaxes for the RDF format. Up until 24th of February 2014, RDF/XML was the only standardized syntax for writing RDF. Now several serializations are standardized, among them is TURTLE which will be used in most of the examples in this thesis. Beneath are some of the serializations used for representing RDF.

- RDF/XML
- N3
  - TURTLE
  - N-Triples

RDF was the first step on the way to make the semantic web a reality. In 1999 when RDF was revealed and W3C proposed that it could be used for resource discovery to improve search engines, structuring content of a web page which normally would have been unstructured, helping intelligent software agents to easily access content on open web pages and collect useful data. RDF could also be used for expressing privacy settings of either a user or a whole web page. In 1999 W3C launched their own interpretation of the concept “the web of trust” which are one of the main keys to semantic web success.

Subject	Predicate	Object
Norway	hasCapital	Oslo
Norway	population	≈ 5000000
Oslo	population	≈ 620000

Table 2.1: Example of triples

An RDF graph may contain an infinite amount of triples. A triple is a sentence which contains a subject, a predicate and an object. This defines some logical dependency between the subject and the object. An example is shown in Table 2.1. There are three ways to represent the information in an RDF graph and that is either with uniform resource identifiers (URI), blank nodes or literals. URIs can be used in the subject, predicate and the object position, blank nodes can be used in subject or object position while literals can only describe the object. The document described in Table 2.1 may be written in TURTLE syntax shown in Listing 2.1.

```
@prefix dbpedia: <http://dbpedia.org/resource/>
@prefix dbp-ont: <http://dbpedia.org/ontology/>

dbpedia:Norway  dbp-ont:capital          dbp:Oslo .
dbpedia:Norway  dbp-ont:populationTotal "5000000" .
dbpedia:Oslo    dbp-ont:populationTotal "620000" .
```

Listing 2.1: An RDF document in TURTLE syntax

On 24th of February 2014, W3C published the new RDF 1.1 recommendation. [26] There they added more serializations as standards and changed from using Uniform Resource Identifiers (URIs) to Internationalized Resource Identifiers (IRIs). The difference between them is that IRIs are international and allows non-latin alphabet characters in them. An URI is almost the same as an URL except the URI might not link to an actual web page. Unlike URL which should provide a location of the resource, hence the name Uniform Resource Locator (URL). URIs are therefore a great way to separate different resources since they will always be unique. This is also quite effective when one wants to merge two RDF graphs. It will not arise any merge conflicts since the URIs are unique and then eliminates the possibility for name clashes. Often the content of an RDF graph may be referred to a graph where the subjects and the objects are nodes, and the predicates are the edges. Still an URI from a predicate position can be a subject in another triple because a predicate can be assigned a property. This means that an edge can also be a node in the graph which is in a normal notion of a graph is a little odd.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix fam: <http://www.ifi.uio.no/INF3580/v13/family#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

fam:hasSibling rdfs:range foaf:Person ;
               rdfs:domain foaf:Person.
```

Listing 2.2: Adding domain and range to a property

2.3.2 Resourse Description Framework Schema & Reasoning

RDF Schema is a vocabulary defined by W3C and it was originally thought to be a "schema language" for RDF just like XML Schema. In an earlier publication, RDFS contained different kinds of constraints which would specify what properties classes and types may have. [15] Today RDFS is not used for validating RDF syntax, but for enriching property and resource description in RDF. RDFS introduces the concepts of classes to RDF which only has *rdf:type* to assign subjects and objects to different categories. The RDFS vocabulary adds some axioms about resources and properties, these are the most important.

- Resource
  - *rdfs:Class*
  - *rdfs:Resource*
- Property
  - *rdfs:domain*
  - *rdfs:range*
  - *rdfs:subClassOf*
  - *rdfs:subPropertyOf*

The first axiom added is *rdfs:Class* which is according to rdfs documentation the superclass of everything. [13] The second is *rdfs:Resource* and is a subclass of *rdfs:Class*. It states what is a resource which is what one can have in a subject and an object position in RDF triples. This alone does not add much functionality to RDF, but the property statements do. In RDFS one can define what domain and range a property should have. In Listing 2.2 it is defined a domain and a range for the relation *fam:hasSibling*. This means that the subject and the object resource when using this relation has to be an instance of *foaf:Person*. Two other important statements added with RDFS are *rdfs:subClassOf* and *rdfs:subPropertyOf*. The first is used to declare a subclass of another class. This means that an instance of the subclass will also be an instance of the superclass. This shows its use when one wants to apply reasoning on an RDF graph. The second statement is the same as subclass just for properties. In Listing 2.3 there is an example on how subClassOf and subPropertyOf can be used. In the example, *fam:familyMember* is set to be a subClass of foaf:Person, which means that every family member has to be a person.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix fam: <http://www.ifi.uio.no/INF3580/v13/family#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

fam:familyMember   rdfs:subClassOf      foaf:Person .
fam:hasFamilyName  rdfs:subPropertyOf   foaf:name .
```

Listing 2.3: Adding domain and range to a property

Reasoning is used to derive RDF triples from existing triples in an RDF graph. This is done by using a set of entailment rules and a set of premises. RDFS has a set of entailment rules which say something about how to derive certain triples. For instance in the set of RDFS entailment rules there is a rule for transitivity. This means that if AAA is a subclass of XXX and XXX is a subclass of YYY, then AAA is also a subclass of YYY. This is entailment rule rdfs11. Listing 2.4 shows an example showing the use of another RDFS entailment rule.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex:   <http://www.example.org/> .

# Premises
ex:Man  rdfs:subClassOf foaf:Person .
_:Adam  rdf:type        ex:Man .

# Derived from entailment rule rdfs9
_:Adam  rdf:type        foaf:Person
```

Listing 2.4: RDFS entailment rule 9

RDFS reasoning is quite simple since it only allows certain kind of derivations. For instance it does not include relation symmetry which is quite useful in several cases. In the next chapter OWL will be described which adds a lot more functionality both to the RDF graph and to the reasoning. Often there will also be useful to describe your own ontology and do your own reasoning.

2.3.3 Web Ontology Language

The Web Ontology language, also known as OWL, is the standard ontology language to this date. It became a W3C recommendation in 2004. Today a lot of people who are modeling ontologies are using OWL 2 which is just an extension of the former OWL. Today people just use the term OWL 1 for OWL 2 which is backwards compatible so that everything that were allowed in OWL 1 is allowed in OWL 2. Like RDF, OWL is a concept of how to define sets of things as classes, properties and instances. This means that OWL syntax can be written in different ways. With RDF we had syntaxes like RDF/XML, TURTLE, N3 and so on. The same is for OWL. There are several ways to write OWL syntax and the most common one is OWL/RDF.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix :    <http://www.example.org/> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

:hasNickname     rdf:type          owl:DatatypeProperty .
:hasViewer       rdf:type          owl:ObjectProperty

:Adam            :hasNickname      "Ellohime"^^xsd:string .
:Adam            :hasViewer        :Magnus .
:Magnus          :hasNickname      "Primstav"^^xsd:string .
```

Listing 2.5: OWL properties

Here OWL is written in RDF syntax where one states the triples and their relations except adding the OWL vocabulary. Another way of writing OWL is in OWL/XML, which is a non-RDF XML format.

OWL overwrites some relations from RDF and RDFS, an example is that *rdfs:Class* is replaced by *owl:Class*. Also *rdf:Property* is not used in OWL. This is because OWL introduced different types of properties, *owl:DatatypeProperty*, *owl:ObjectProperty* and *owl:AnnotationProperty*. These different property relations describes and dictates the object in a relation. DatatypeProperty links resources to data values like *xsd:string*, ObjectProperty links resources to other resources and AnnotationProperty is used to annotate a resource. Annotation properties will be ignored by reasoners since it is not important for deriving new information. It is mostly used for labeling resources so that it can be more readable by humans. In Listing 2.5 there is an example to illustrate the different properties in OWL. An important thing to make notice about these three property relations is that they are mutually disjoint which means that a property can not be more declared to be both an ObjectProperty and a DatatypeProperty.

Most of the RDFS relations are carried on in OWL, like *rdfs:subClassOf*, *rdfs:subPropertyOf* etc. This allows one to use some of the familiar syntax to describe class hierarchy, range/domain to properties and property hierarchy. With OWL one can now express things that were not possible with RDFS. In OWL one can for instance describe symmetric and inverse relations without having to construct specified relations like one would have to do in RDFS. This is also present when reasoning on RDFS and OWL. RDFS is sound, but not complete while OWL are both. This is why OWL is used to model new ontologies. One can, with a correctly modelled ontology, derive exactly the triples one would want with reasoning. There is also no trouble to extend the vocabulary with ones own relations like seen in basic RDF.

2.3.4 SPARQL

SPARQL, SPARQL Protocol And RDF Query Language, is as the name implies a language for querying RDF graphs. SPARQL first emerged as a working draft in 2004, but in 2008 it became an official W3C recommendation. It resembles SQL (Structured Query Language), which is

the main query language for working on relational databases, and TURTLE syntax. Some similar terms in SPARQL and SQL are *SELECT*, *WHERE* and *ORDER BY*. The big difference between the two query languages is the querying. The way SPARQL queries an RDF graph is by matching triples in a query with triples in the RDF. This means that it is necessary to get a complete match if one wants to get a value in return. A SPARQL query is written in the same way one writes triples in TURTLE syntax except in a query one is allowed to add variables. A variable in SPARQL is represented by a question mark(?) before a name. An example is shown in Listing 2.6 where the subject and the predicate must match to a subject and a predicate in the RDF, but the object can be whatever which matches with the whole triple.

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbp-ont: <http://dbpedia.org/ontology/>

SELECT ?capital
WHERE {
        dbpedia:Norway  dbp-ont:capital ?capital .
        }
```

Listing 2.6: A SPARQL query to find the capital of Norway

In SPARQL there are other functionalities which are important for making good queries. Filter is one of those functionalities. It is used for filtering the result by checking a filter-clause. Here one can check if the capital of Norway starts with an "O", or it contains over 600 000 people. In Listing 2.7 one can see a query which filters on the population of Norway. That query will return the literal “50000000” since it fulfills the filter-clause.

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbp-ont: <http://dbpedia.org/ontology/>

SELECT ?population
WHERE {
        dbpedia:Norway  dbp-ont:populationTotal ?population .
        FILTER (?population > 4500000)
        }
```

Listing 2.7: A SPARQL FILTER example

Other important functionalities in SPARQL would be UNION and OPTIONAL. UNION is used to take the union of graph patterns. This is done by writing the wanted triples in the first query then writing "UNION" and then write the triples in the second query. Just remember to surround the first and the second query with curly brackets. OPTIONAL is used to specify optional information. An example would be if one is searching for a person in a FOAF(Friend of a friend) file. The example is shown in Listing 2.8.

2.3.5 GoodRelations

GoodRelations is a web vocabulary for e-commerce and was launched in 2008. Over 10000 businesses and web sites are using GoodRelations. Some

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?person ?name
WHERE {
        ?person rdf:type foaf:Person .
        OPTIONAL{
                ?person foaf:name ?name .
        }
}
```

Listing 2.8: A SPARQL OPTIONAL example

of them are Google, Yahoo! and BestBuy. [2] GoodRelations can be used for detailed information about products to be sold online. It uses an Agent-Promise-Object principle which means than an agent or a user gets a promise about an object. A promise might be a service or transfer of ownership while the object can be anything that can be sold or done for the agent. For instance if a user wants to buy a car. Then the promise is that the ownership of the car, the object, will be transferred from the store to the user. This allows one to use the same vocabulary for offering services and items. Under one can see some of the classes found in GoodRelations.

- *gr:BusinessEntity*
- *gr:Offering*
- *gr:ProductOrService*
- *gr:Location*
- *gr:QualitativeValue*
- *gr:QuantitativeValue*

These four classes can describe the Agent-Promise-Object principle. The *gr:Offering* class represents an offer to sell, repair, lease or to express interest in a service or an item. This functions as the promise. The *gr:ProductOrService* is as the name implies the service or the product the offering describes, the object. The agent is just the user which access this information and is in need of a service or an item. The next two classes are represent the information about the seller. Here *gr:BusinessEntity* describes a company or an individual while *gr:Location* describes where the offering can be obtained.

The two last classes represent values in GR. A qualitative value is a predefined value for a product characteristic. Product color is a qualitative value. If we want to represent that our coffee mug is green, we should wrap it in a *gr:QualitativeValue*. A quantitative value can be a numerical value or interval that represents the range of a certain product or feature. Number of grams our coffee mug weights is another example. This should then be wrapped in a class *gr:QuantitativeValue*. [29]

GoodRelations uses the three different owl properties, object property, datatype property and annotation property. Under there is a small snippet

```
:CoolComp a gr:BusinessEntity;
        gr:legalName "The Cool Company";
        :address [ :streetAddress "Cool Street 5a";
                :postalCode "12345";
                :addressLocality "Oslo, Norway" ];
        :telephone "004712345678";
        :email "contact@awesome.org";

        gr:Offers :Offer .

:Offer a gr:Offering;
        gr:name "SuperBall 2000";
        gr:description """The ultimate football""";
        gr:hasBusinessFunction gr:Sell;
        gr:condition "new" .
```

Listing 2.9: GoodRelations example

of the properties used in GoodRelations and which property they are a subPropertyOf.

- Datatype Properties
  - *gr:name*
  - *gr:serialNumber*
- Object Properties
  - *gr:owns*
  - *gr:acceptedPaymentMethods*
- Annotation Properties
  - *gr:displayPosition*
  - *gr:relatedWebService*

In addition to all the properties and classes there are some individuals GoodRelations has default added. Those are mostly individuals describing payment companies, delivery companies and weekdays. An example is *gr:MasterCard* which is an individual describing *"Payment by credit or debit cards issued by the MasterCard network"*. [3]

There is an example on how GoodRelations is used in Listing 2.9. It shows how a business entity is represented with additional information. Also with a business entity there can also be offers which will link to an individual offer. The *"SuperBall 2000"* is an offer. It also has additional information like name, description, condition and that it is for sale. There is a lot of additional information which could have been added, for instance if it was a book it would have been relevant to add the ISBN number.

This may be why some major companies, which are dealing with online merchandise, is using GoodRelations.

2.3.6 Linked data

Linked data is about using the Web to create links between data from different sources. [27] The data may be as diverse as heterogeneous databases. Linked data refers to data published on the web and that are machine readable, which means that its meaning is defined. Tim Berners-Lee proposed in an article (2006) four rules for publishing data on the web, a ruleset for linking your data. [12]

- Use URIs as names for things
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)
- Include links to other URIs. so that they can discover more things.

These rules were proposed to define a basic recipe for publishing data on the web. The rules are meant as guidelines and breaking those rules will have no consequence other than making the data less useful to others. It would make it easier for several actors to share and enhance their data in conjunction with others. Linked data is also reliant on the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers, URIs. The linked data term can be split into two sub groups, Linked Open Data and Linked Enterprise data.

Linked open data

Linked open data (LOD) can be described as a part of linked data. LOD is the amount of data which is open to the public. Tim Berners-Lee made a 5 star rating system for LOD. [12] One would get 1 star for each of the criterias for LOD he presented. The rating is incremental so one would need to satisfy the first in order to move on to criteria 2. Under we see the 5 criterias.

- 1. Available on the web (whatever format) but with an open license, to be Open Data
- 2. Available as machine-readable structured data (e.g. excel instead of image scan of a table)
- 3. As (2) plus non-proprietary format (e.g. CSV instead of excel)
- 4. All the above plus, Use open standards from W3C (RDF and SPARQL) to identify things, so that people can refer to your data
- 5. All the above, plus: Link your data to other people's data to provide context

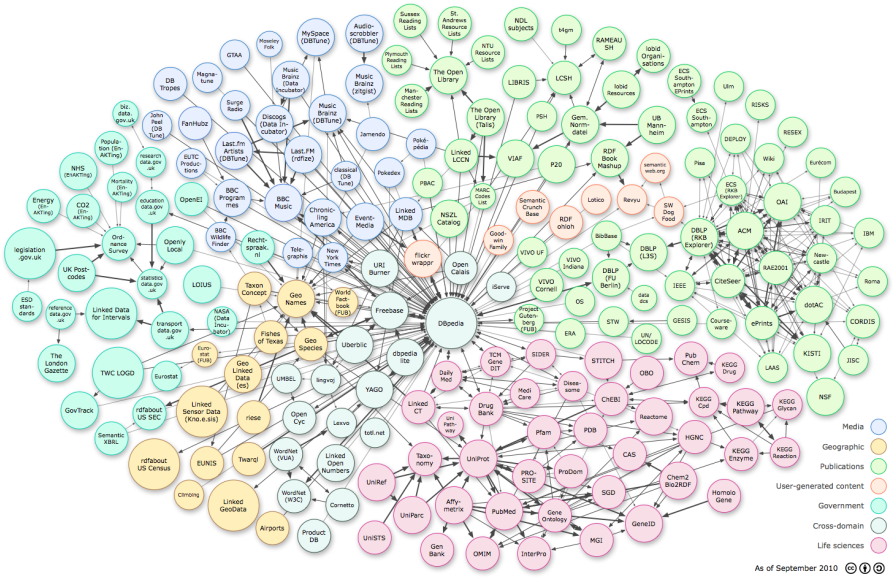

Figure 2.3: Illustration of how the linked open data cloud looks like

Today the data is often published via SPARQL endpoints for third party applications or persons to access. Dbpedia is an example of linked open data with an SPARQL endpoint. Dbpedia is everything in Wikipedia stored as RDF triples. There are other companies and organizations which publishes their data on different formats. Renault has published their data with an API to extract information about the range of cars that Renault manufactures. This way of publishing data is done because Renault has a huge amount of possibilities of extracting their data, so that they have hidden the reasoning on their side. This is to relieve the pressure of performance and speed on the client. In Figure 2.3 we can see an illustration of how the LOD cloud interacts.

Linked enterprise data

Linked enterprise data (LED) is the same as LOD except where it is published and where it is visible. LED is data published within a community, company or organization. Often one talks about LED when opening data within a company over department borders. For instance could LED at the University of Oslo be that the department of informatics would publish their data for the whole of Univerity to benefit from, but no one else. This could be used to maybe track computer activity over the whole University etc.

2.3.7 Ontology alignment

Ontology alignment, or ontology matching, is in semantic web a process to determine equality between concepts and is mostly used when linking data from multiple LOD clouds. As described in the section about LOD, LOD

clouds are often used to enrich data on the web. This is often done with a link between an entity used by an application to an entity in the cloud. To do this one often needs ontology alignment, but some LOD clouds may offer an easier solution for instance freebase.com. They offer an API for querying their cloud where they can suggests similar entities and concepts to make it easier to enrich the data. [4]

Manual and automatic alignment

There are two main ways of aligning ontologies. One can do it manually or automatically. Manually its most often used when resources in two or more ontologies are hard to identify by general knowledge. The automatic aligners often use public databases for their alignment, like freebase and dbpedia. In OWL there are different properties for linking equivalent classes, properties and resources. We will focus on manual alignment since it is the approach we chose to use later on in this thesis.

OWL axioms are a regular approach to align ontologies or parts of ontologies. There are three different alignment properties. With these three properties one can align classes, properties and resources.

- *owl:sameAs*
- *owl:equivalentClass*
- *owl:equivalentProperty*

The equivalent properties are used for aligning classes and properties defined in the ontology, while *owl:sameAs* is used for stating equivalence between resources. For instance might one want to say that petrol is the same as gasoline. One often uses these three in conjunction with reasoning to obtain wanted result.

SPARQL Construct is another way of aligning ontologies. This is a more practical way of obtaining the result. With a SPARQL contruct one identifies the wanted triples and then one can change the current RDF graph by either altering the triples or adding new triples. In Listing 2.10 we see a contruct which alters the name property from "foaf:name" to a new name property. SPARQL contructs are more powerful than OWL axioms, but are not so easy to make generic for more than one case.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/#term_>
PREFIX p: <http://www.newNameBank.com/term#>
CONSTRUCT {
        ?somePerson p:name ?name .
} WHERE {
        ?somePerson foaf:name ?name.
}
```

Listing 2.10: SPARQL Construct example

2.3.8 Jena

Apache Jena is a free and open source java framework for building Semantic Web and Linked Data applications. It was created by the Apache Software Foundation which also have other known projects like Maven, Hadoop etc. Apache Jena is made for the programming language Java. It can easily be added to a regular java program either just as a jar-file or through Maven.

Jena is a large framework which consists of several components which can be divided into three different categories.

- RDF
  - RDF API
  - ARQ
- Triple Store
  - TDB
  - Fuseki
- OWL
  - Ontology API
  - Inference API

RDF

The first category, RDF, contains the RDF API which can build the foundation of almost any semantic web application. This is because it is the core API for creating and reading RDF graphs. It has support for reading the most common RDF serialization like TURTLE, RDF/XML, N3 etc. [43] With the RDF API one represent the wanted RDF graph in a Model. It can be done by using classes and functions within Jena to extract an RDF graph and contain it in a model as we see in the example below.

```
Model model =
      FileManager.get().loadModel("http://uk.co.rplug.renault.com/docs#this");
```

The second big part of the RDF category is ARQ, A SPARQL Processor for Jena. ARQ is a query engine for Jena that supports SPARQL 1.1. This means that Jena has full SPARQL support and that one can do any type of SPARQL query against a given model. [11]

Triple Store

The second category, as the name implies, is about storing RDF triples in different ways. TDB is Jena's own triple store, which means that one can store RDF on a single machine with high performance. This also opens for querying triples in the DB. [42]

If one does not want to store their triples locally one can use Fuseki. It is Jena's own SPARQL server. This means that one can set up their own

SPARQL endpoint so that everybody can access their data through HTTP. This provides REST-style [2] interaction with RDF data. [23]

OWL

In the last category there are the Ontology API and the Inference API. The Ontology API one can say is an extension of the RDF API. It adds RDFS and OWL semantics to the RDF data. This opens for a lot more triples and more ways of connecting data with the new semantics. When representing complex data it is not sufficient enough to only use the RDF semantics, which means that RDFS and especially OWL comes in handy. [32]

The last part of Jena is the Inference API. This adds reasoning to Jena. The Inference API has several in-built reasoners like RDFS reasoners and other OWL reasoners. The use depends on what kind of reasoning task one wants done. It also allows for adding custom reasoners if there is some reasoning need that is not met with the in-built ones or that one wants to optimize a reasoner for faster results. [39]

Structure

Figure 2.4 on the next page shows how the different parts of Jena interact with both an application and their internal structure. It can be broken down to almost 4 independent parts. The RDF API is the basic structure for creating or extracting an RDF graph. The ontology API and ARQ depends on functionality within the RDF API. Both the storage units can be used outside the RDF API. This is because one only need an RDF graph to persist in order to use them. The Inference API is also seen as an independent unit. This API is not used for anything unless one have an application using the RDF API, but it has no dependencies in the RDF API. It only applies different rules to a model and infer new triples from them.

[2] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

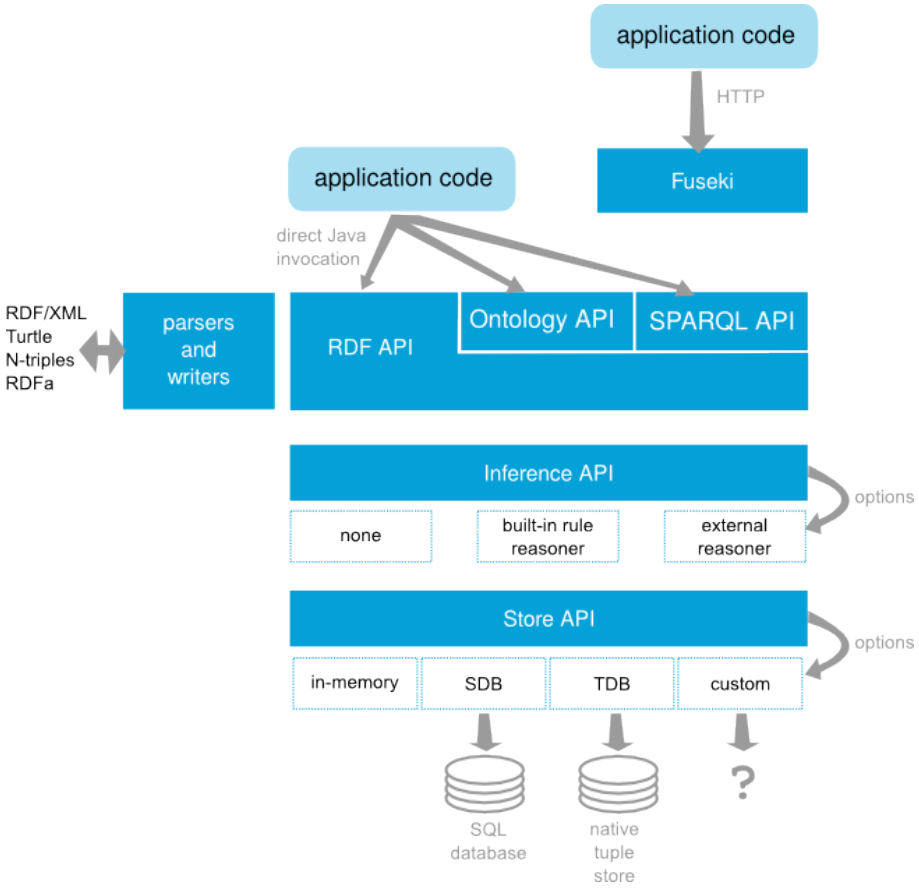

Figure 2.4: The interaction between the different APIs in Jena

Chapter 3

Ontologies

In this chapter the structure of the three ontologies used in this thesis will be described. All of the ontologies have a large number of properties and classes which means that only the most important ones will be described. For the rest, we will refer to their documentation.

3.1 Vehicle Sales Ontology - VSO

The vehicle sales ontology is a web vocabulary for describing cars, boats and other vehicles on the web. This is particularly meant for e-commerce to use in sales. The vocabulary is written by Martin Hepp, who also is the founder of GoodRelations. VSO is designed to be used in combination with GR. It is a way to describe the vast amount of specifications a vehicle can have, and to increase the visibility of a sale on the web. The vocabulary supports both new and used vehicle for sale. The ontology is meant to only represent instances of vehicles, for instance if one wants to sell their old *Volvo V40*. The classes and properties are described based on the definitions found in the VSO vocabulary as of 10th February 2014. [30]

3.1.1 Classes

The vehicle sales ontology got 28 classes from Automobile to Watercraft. [31] Here some of the most important ones will be presented.

Motorized road vehicle

> *A motorized road vehicle is a wheeled land vehicle whose main propulsion is provided by an engine or motor.*[1]

This is the super class to all land driven motorized vehicle like motorcycles, cars, etc. This is a subclass of *vso:Vehicle* which in turn is a subclass of *gr:ProductOrService*. This means that every vehicle is considered a product which in GoodRelations terms can be offered in an e-commerce environment.

[1] Definition taken from the Vehicle Sales Ontology vocabulary [30]

Automobile

> *An automobile, motor car, or car is a wheeled motor vehicle used for transporting passengers, which also carries its own engine or motor.*

This class is the most important one for this use case. It is a subclass of *vso:MotorizedRoadVehicle*. Also it is the domain of a large amount of properties which will be described in further detail later.

3.1.2 Properties

VSO has a lot of properties, most of them are object properties and here a small subset of the properties relevant for this thesis. The properties described below are all object properties and can be divided in two groups, properties defining qualitative values and properties defining quantitative values. More about these types of values can be found in Section 2.3.5 on page 13.

Acceleration

This property describes the acceleration of a vehicle. A common measurement for acceleration is how many seconds a vehicle uses from 0 to 100 kilometres per hour. To define the acceleration one uses the annotation *vso:acceleration*. It is a sub-property of *gr:quantitativeProductOrServiceProperty* which implies that this object should have specified quantitative characteristics, which were described in Section 2.3.5 on page 13. The *vso:Acceleration* has a domain which includes all vehicles found in the VSO vocabulary.

Fuel type

Fuel type is represented as *vso:fuelType*. It describes what type of fuel is suitable for the engine of the vehicle. The range of this object property is the class *vso:FuelTypeValue*, but one can also use dbpedia resources.

Total amount of gears

With the *vso:gearsTotal* one can specify the number of gears a vehicle has. This is also a sub-property of *gr:quantitativeProductOrServiceProperty*.

Transmission

Almost every vehicle has a transmission. That can be represented with *vso:transmission*. It is quite similar to fuel type because the standard values are dpedia resources which describes manual and automatic transmission, but one can also define their own transmission type with the class *vso:TransmissionTypeValue*.

Example

All of these properties and classes plus many more are used to describe sales of used and new vehicles. In Listing 3.1 one can see them used in a minimalistic example. In the example it is used the unit of measurement *C62*. It is the ISO standard for describing that a value is unitless, that it is only a number. [38]

```
<http://folk.uio.no/magnudae/daimler#31806> a vso:Automobile ;
  rdfs:label "A 250 Sport" ;
  gr:isVariantOf daim:AClassBase ;
  vso:acceleration [ a gr:QuantitativeValueFloat ;
          rdfs:label "Acceleration 0-100 km/h" ;
          gr:hasUnitOfMeasurement "s"^^xsd:string ;
          gr:hasValueFloat "6.6"^^xsd:float ] ;

  vso:fuelType [ a gr:QualitativeValue ;
                 gr:valueReference dbpedia:Manual_transmission ;
                 gr:name "Manual Gearbox"@en    ] ;
  vso:gearsTotal [ a gr:QuantitativeValueInteger ;
          rdfs:label "Forward gears (total number)" ;
          gr:hasUnitOfMeasurement "C62"^^xsd:string ;
          gr:hasValueInteger "7"^^xsd:int ] ;
  vso:transmission dbpedia:Automatic_transmission .
```

Listing 3.1: Example data about VSO

3.2 Car Options Ontology - COO

With the vehicle sales ontology in mind, there was also made another ontology for e-commerce. The car options ontology (COO) is a vocabulary which describes available configuration options for car models. It was constructed by Martin Hepp in collaboration with Volkswagen. The ontology was made to help navigate through all the configurations a car may have, to help the user define a partially defined product (PDP). COO has the option of setting compatibility, dependency and inclusion information. Like VSO, COO also extends the GoodRelations ontology and it was made to be combined with VSO. COO is meant to describe car models, not instances of them. The classes and properties are described based on the definitions found in the COO vocabulary as of 10th of February 2014. [28]

3.2.1 Classes

COO has three important classes which defines different types of car models.

Base model

As the name implies it is the basic model. It can be defined with *coo:BaseModel* and can contain standard information about that particular car model. This can be height, weight, number of wheels etc.

```
ex:GolfBaseModel a coo:BaseModel, vso:Automobile ;
                 rdfs:label "Golf"@en ;
                 vso:fuelType ex:Diesel .

ex:GolfS a coo:Trim, vso:Automobile ;
        rdfs:label "Golf S"@en ;
        gr:variantOf ex:GolfBaseModel ;
        vso:gearsTotal   [ a gr:QuantitativeValue ;
                           gr:hasValue "6"^^xsd:integer  ] .

ex:GolfSRacingEdition a coo:Derivative, vso:Automobile ;
        rdfs:label "Golf S Racing Edition"@en ;
        gr:variantOf ex:GolfS ;
        vso:speed    [ a gr:QuantitativeValue ;
                           gr:hasValue "280"^^xsd:integer  ] .
```

Listing 3.2: Larger example with base model, trim and derivative

Trim

A trim is an entry configuration of a base model. Here one might specify all the features this car may have and other features, like transmission, fuel, etc. An example would be that a Volkswagen Golf S is a trim of the regular Golf. To specify that a model is a trim one uses the *coo:Trim* class. It should inherit the features from a base model. This is done with *gr:isVariantOf*.

Derivative

A derivative model should have specified configurations and compatibilities. Trim and derivative is very similar except that the derivative model should specify compatibilities and constraints between configurations. For instance it could describe that getting the model in silver is not compatible with getting an AM/FM radio installed. One specifies that a model is a derivative with the class *coo:Derivative*. The derivative also inherits all the features specified from its trim with the *gr:isVariantOf* as seen in diagram 3.1 on page 32.

In Listing 3.2, we see a larger example of how the three model classes interact. Here, the base model only contains the information about the fuel type. The trim inherits the information from the base model through *gr:isVariantOf*, which means that it also has the fuel type Diesel even though it does not have a specific triple stating it. It is the same with the derivative. It inherits all information from the trim, which in our case is number of gears and the fuel type from base model.

3.2.2 Properties

The properties in COO are mostly about representing different choices one can have on a car.

Compatibility

There are several ways of representing compatibility with COO. One way is to pair compatible or incompatible resources. This can be done with either *coo:compatibleWith* or *coo:incompatibleWith*.

```
ex:CarModel1 vso:fuelType ex:Manual_Transmission ,
                          ex:Automatic_Transmission .
ex:Manual_Transmission coo:incompatibleWith ex:Automatic_Transmission .
```

Both properties are defined as *owl:SymmetricalProperty*[2], which means that in this example *Automatic transmission* is also incompatible with *Manual tranmission*. The incompatibility property has nothing to do with *owl:incompatibleWith* which defines ontology incompatibility.

Another way of representing compatibility is with the property *coo:compatibility*. With this property one need a ConfigurationInfo resource to state that a sequence of specifications are either valid or invalid.

```
ex:CarModel1 coo:compatibility ex:TransmissionCompatibility .
ex:TransmissionCompatibility a coo:ConfigurationInfo ;
        coo:includesChoices ex:Manual_Transmission,
                            ex:Automatic_Transmission ;
        coo:valid "false"^^xsd:boolean .
```

Available - & Default choice

Each derivative of a car can have a set of choices or components. The *coo:availableChoice* and *coo:default* are used within a item collection. These properties state which specifications comes as default and which can be chosen to a car which contains this item collection.

```
ex:ColorCollection a coo:SpecItemCollection ;
        coo:default ex:Red ;
        coo:availableChoice ex:Blue, ex:Green .
```

3.3 COO & VSO

Both these ontologies can easily be used in conjunction with each other. This is because COO is designed to fit with both GoodRelations and Vehicle Sales Ontology. It is COO which brings the complexity to the ontologies since it is only COO which handles the compatibility issues. This is one thing that make VSO alone insufficient to represent the amount of specification possibilities. One big difference is when VSO is used alone it represent instances of cars, but when it is used in combination with COO it helps enrich data about car models. This is because with COO there are often several ways to define a car. In Figure 3.1 on page 32 we have a diagram showing how the two ontologies are used in conjunction. The previous section described the different types of models. In the diagram we see that the three types form a subclass hierarchy. Even though a Derivative

[2] http://www.w3.org/TR/owl-ref/#SymmetricProperty-def

```
:Diesel a    co:Specification ;
      rdfs:label "Diesel"@en ;
```

Listing 3.3: Specification example

is not a subclass of Trim, it will inherit all the features added to the trim which in reality makes it an extension of the Trim. The same is true about the relationship between Trim and BaseModel. The diagram also shows that there are two different types of values to represent, QuantitativeValue and QualitativeValue. Each of the VSO properties has the range of either one of those values.

3.4 Configuration Ontology

Here we will present a quick overview of the Configuration Ontology (CO). It is a generic ontology for describing different configurations of a product model. It is supposed to be able to represent any kind of PDPs or customizable product models. The configuration ontology can be enriched with the GoodRelations ontology framework or other ontologies if it is applicable to the data. The classes and properties are described based on the definitions found in the CO vocabulary as of 25th January 2014. [20]

3.4.1 Classes

In the CO there are several classes and they can be separated into two main groups. The first three classes in this section describe the product model itself, components and other features. The two last classes describe compatiblity and how each product model can be created.

Specification

A specification represents a real world object which can be a Diesel fuel, the color red, etc. Instances of this class contains the actual value of the specification. It can also be a subclass of either *gr:QualitativeValue* or *gr:QuantitativeValue* which makes it easier to integrate with GoodRelations. In Listing 3.3 we can see an example of how a specification resource could look like.

Configuration Variable

A configuration variable is a class which should describe a type of specification, like fuel type or transmission. It should contain a link to one or more specifications and a label stating what type it is describing. In Listing 3.4 we can see an example on how it can be used. We can see that it contains a label describing the specification type and it has a property linking to a specification instance with this type.

```
:FuelType a  co:ConfigurationVariable ;
      rdfs:label "Fuel Type"@en ;
      co:hasValue :Diesel .
```

Listing 3.4: Configuration variable example

Lexicon

A lexicon is a recipe for a product. It is as the name implies, used as a lookup table for finding specifications a product may have. Each product model will have their own unique lexicon. As the documentation states today, the lexicon should be a URI linking to an RDF graph containing several configuration variables and even more specifications. A lexicon may look like Listing 3.3 and 3.4 combined.

Configuration & ConfigurationLink

The next two classes are used to add compatiblity to the ontology. They are interdepenent which is why we will present them together. When using both classes they form a tree structure. The configurations is simlar to the nodes while the configuration links are similar to the edges.

A configuration can be defined by *co:Configuration*. It represent a valid partially defined product and can contain links to several previously defined configurations, alternative configurations and possible new configurations. It should also have a link to a unique lexicon. This means that a configuration has access to its own recipe. When creating the knowledge base for CO one need a starting point which means that there is a need for a list of starting components to choose from. In Listing 11 we can see that the starting component is fuel types. The starting configuration can be seen as the root of the configuration tree.

To move to a new configuration one need a configuration link and it can be defined by *co:ConfigurationLink*. It is used to represent the edge between configurations. A configuration link should contain a link to a configuration aswell as a link to a choosable specification. In Listing 16 we have chosen the specification Diesel and have then moved to a new configuration through a configuration link. In this next configuration we have the opportunity to choose the transmission of the car model, but also if we regret our previously chosen specification, we have an alternative. We can choose Petrol instead, but then we have to move to a new configuration which has Diesel removed as a specification. In Figure 3.2 on page 33 we can see a visual approximation of the interaction between configurations and configuration links.

```
ex:StartingConfiguration a co:Configuration ;
      co:lexicon ex:VolveV40Lexicon ;
      co:possible [ a       co:ConfigurationLink ;
                  co:linkedConf ex:DieselConfiguration ;
                  co:specToBeAdded ex:Diesel
                  ] ;
      co:possible [ a       co:ConfigurationLink ;
                  co:linkedConf ex:PetrolConfiguration ;
                  co:specToBeAdded ex:Petrol
                  ] .
```

Listing 3.5: Starting configuration

```
ex:DieselConfiguration a co:Configuration ;
      co:lexicon ex:VolveV40Lexicon ;
      co:alternative [ a       co:ConfigurationLink ;
                  co:linkedConf ex:PetrolConfiguration ;
                  co:specToBeAdded ex:Petrol
                  co:specToBeDiesel ex:Diesel
                  ] ;
      co:possible [ a       co:ConfigurationLink ;
                  co:linkedConf ex:AutomaticConfiguration ;
                  co:specToBeAdded ex:Automatic_Transmission
                  ] ;
      co:possible [ a       co:ConfigurationLink ;
                  co:linkedConf ex:ManualConfiguration ;
                  co:specToBeAdded ex:Manuel_Transmission
                  ] .
```

Listing 3.6: The next configuration

3.4.2 Properties

Values

Configuration variables should link to its specifications with the *co:hasValue* property. There are no specific property in CO to represent actual qualitative or quantitative values. One can either use another ontology or use *rdfs:label*.

Properties around configurations and specification

There are several properties to define the way through the configuration tree. *co:possible* and *co:alternative* are used to refer to a configuration link which represents the edge to a new configuration. Both of them have the domain *co:Configuration* and the range *co:ConfigurationLink*. Inside the configuration link we find the property *co:linkedConf* which describes the new configuration.

co:chosenSpec and *co:impliedSpec* both links to a specification. As the name implies they describe either specifications that are already chosen or specifications that are implied at this configuration. They have the domain *co:Configuration* and the range *co:Specification*.

The last two important properties are *co:specToBeAdded* and *co:specToBeRemoved*. They are both used in a configuration link to describe the specification that has to be added or removed in order to move

```
ex:DieselConfiguration a co:Configuration ;
        co:chosenSpec ex:Diesel ;
        co:impliedSpec ex:4x4Drive ;
        co:impossible ex:2Seats ;
        co:alternative [ a co:ConfigurationLink ;
                         co:linkedConf ex:PetrolConfiguration ;
                         co:specToBeAdded ex:Petrol ;
                         co:specToBeRemoved ex:Diesel ] ;
        co:possible [ a co:ConfigurationLink ;
                      co:linkedConf ex:AutomaticConfiguration ;
                      co:specToBeAdded ex:Automatic_Transmission ] ;
        co:possible [ a co:ConfigurationLink ;
                      co:linkedConf ex:ManualConfiguration ;
                      co:specToBeAdded ex:Manuel_Transmission ] .
```

Listing 3.7: The next step in the configuration process

to the configuration. They have the domain *co:ConfigurationLink* and the range *co:Specification*.

In Listing 3.7 we can see a full example on how all properties are used. Here we have chosen Diesel as a specification and now have the possibility to choose transmission. The same process has to be done for every specification we want to add to our product model. This is similar to traversing a tree. We move from node to node until we are satisfied with the results. This is how compatibility is solved in CO. The configurations we can get to with the configuration links are only valid configurations. This means that we will not have the possibility to choose a specification that are not valid with something we already have chosen. The *co:impossible* is used as an extra insurance.

Each configuration is unique and is linked to one product model. This means that the process of choosing specifications has to be done for each product model if the goal is to search through all models. In Figure 3.3 on page 34 we have a diagram describing the classes and properties of CO that are relevant to the prototype. It is smaller than the diagram about COO and VSO. That is because there you see the whole graph just by looking at the classes and properties. Some of the properties were left out of the diagram because they have the same range and domain, like *co:possible* and *co:alternative*. With CO the representation will produce a large graph with resources for every possible way to a fully defined product.

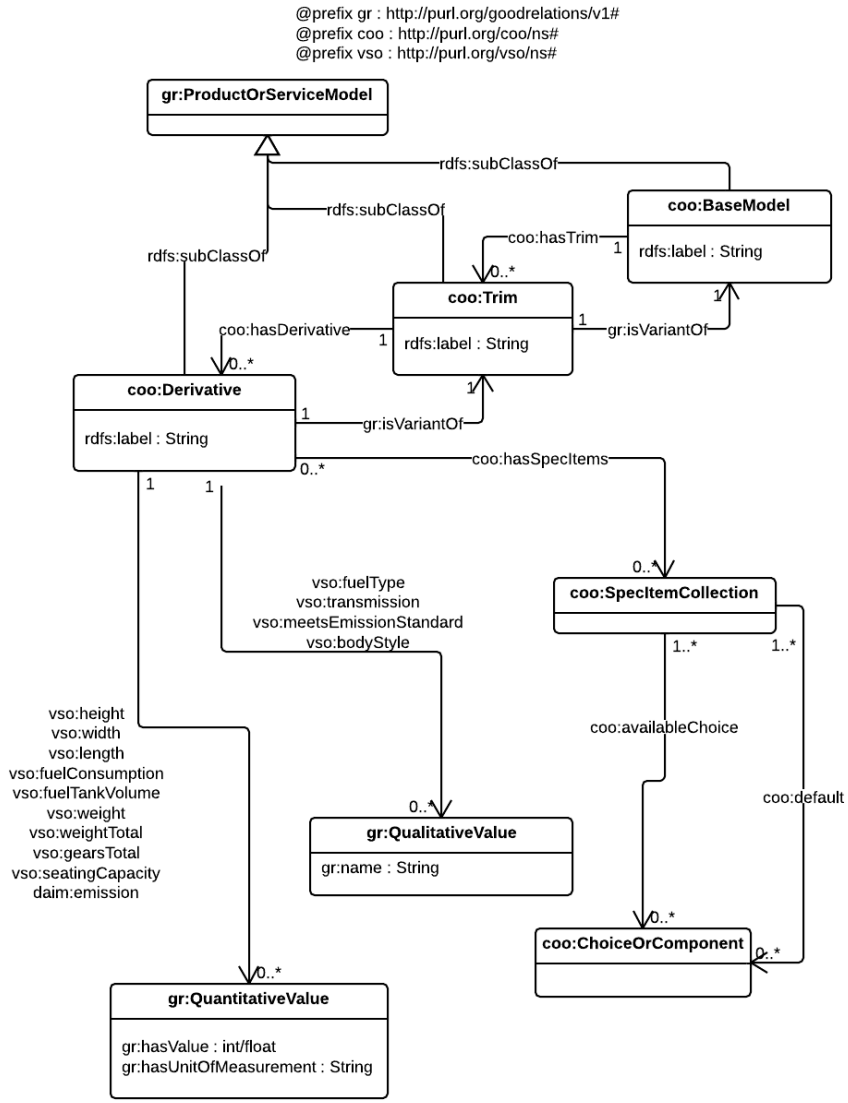

Figure 3.1: Diagram showing how COO and VSO interact

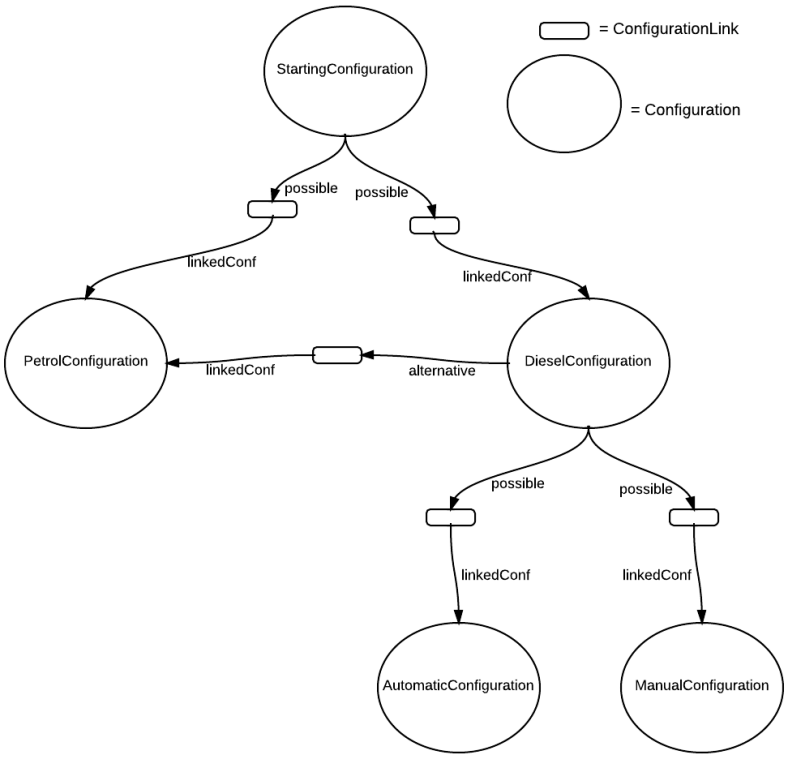

Figure 3.2: Visual approximation of the CO representation

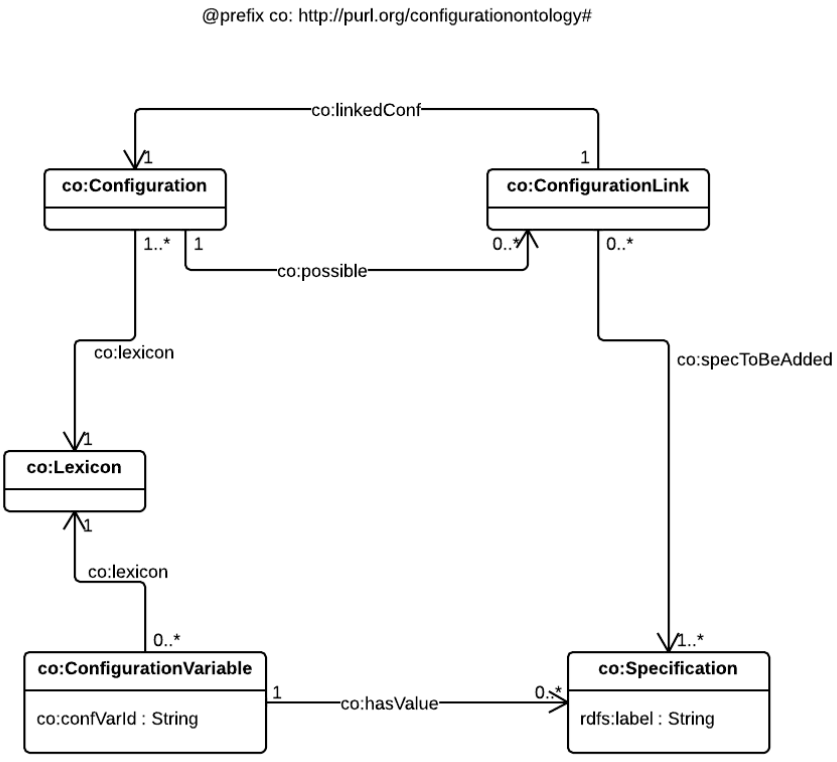

Figure 3.3: Chart describing the Configuration ontology

Chapter 4

Prototype

Ultimately one wants to use semantic technologies in various applications and tools. In this chapter we will take a look at a prototype that has been designed and built in order to make a comparison of the ontology approaches presented earlier. This chapter will also take a look at how the different ontologies represent their data, interact with each other and are presented. Since the purpose of this thesis were to test the two ontology approaches, it has not been taken into account the scalability and robustness of the application because that would be solely a programming issue.

To create a test environment for the thesis we needed data from different sources. Today there is only Renault, of all the car manufacturers, which has opened their data using semantic technologies. Volkswagen had their data published until the start of 2013, but unfortunately they canceled their semantic web project and the data were removed. However, their ontologies are still open for the community to use. After some time searching for data and contacting other car manufacturers, Daimler came through and offered data about their A- and B-class cars.

In this chapter there will be a lot of information around models and lexicons. A *lexicon* refers to the configuration ontology definition of a lexicon, that was described in the Section 3.4.1 on page 29. This is only used to describe car models from the Renault RDF graph.

4.1 Overview

There were several ways to make a functional application using the data we were provided. One could use the data to make a auto-complete system. One would choose a car model or a specification and the application would slowly fill out the form on behalf of the user. Often the auto-complete functionality would not finish before the user had added several specifications due to the possibility that several car models have the same specifications.

Another possible application would be a car chooser application. With this application there are two main approaches:

- Choosing several or all specifications and then search for model

- Choose one after one specification until one have a smaller set of wanted models

The first option demands a lot more from the ontology structure than the latter. This is because a car can contain a lot of different specifications and searching through a graph of specifications can be time-consuming. This is if the graph is huge or if there is not enough information to determine which way to traverse in the graph. We chose to create an application where a user choose several or all specifications, the first option. We chose this because it allowed us to test the performance of big operations on the two ontology approaches. This would most likely give us a more significant result and revealing weak parts in both ontologies.

4.1.1 Programming languages

The first issue we faced before making the application was what kind of programming language we would use. There are a vast amount of possible languages to chose from, but it came down to JavaScript or Java in the end. This is mainly because we had previous experience dealing with both these languages, and both had good support for several semantic frameworks.

JavaScript

Our first thoughts were to make the application with NodeJS which is a back-end JavaScript solution. [5] This would let us make a fast web application without needing to set up any large back-end structure.

After working on the data provided, we quickly understood that JavaScript was not the suitable language for our task. There are a lot of RDF frameworks for JavaScript like rdfstore-js[1] and rdfQuery[2], but the problem was that JavaScript is not a suitable language for handling big operations. [3] Another reason for not using JavaScript were that none of the frameworks we wanted to use had full SPARQL support which meant that we would have to use another language for some tasks.

Java

Another language for programming semantic technologies is Java. It is maybe the most used programming language for semantic technologies and a common framework around this is Apache Jena [7]. Other frameworks similar to Apache Jena are Sesame [4] and OWL API[5]
Jena has full SPARQL support which was needed for our use case. It also has support for using different kinds of RDF serialization and working

[1] https://github.com/antoniogarrote/rdfstore-js
[2] https://code.google.com/p/rdfquery/
[3] Several tests done by Debian shows that JavaScript is slower than java on big operations [6]
[4] http://www.openrdf.org/
[5] http://owlapi.sourceforge.net/

with different ontology models. In the list below one can see which other frameworks were used in the prototype.

- Apache Maven
- Spring
- Apache Tomcat

In the end, Java were chosen as the main language because we had previous experience with both the language itself and the semantic web framework Jena.

4.1.2 Code

Beneath we can see the most important Java classes from the prototype. All code can be found in appendix D

- CarData - The search module
- IndexController - Handling HTTP request to the application
- FormBean - The form that contains the incoming data
- Settings - The class for doing all pre-computations
- Quantitative- and QualitativeValue - To represent the different values from the form.
- PartialCar - To represent every partially defined product

The application present users with a web interface where they can input values for any specification they might want to search for. In Figure 4.1 on the following page we can see how the application initially looks for a user. Here the user can input values for the different specifications, for instance fuel type or CO_2 emission. After a user is done choosing specifications he can execute the actual car model search. The application extracts the data from the form and comprises it into another format that can easily be used later on. The next step begins in the search module. Here it starts a thread for each possible car model to execute each individual search. In the end the user is presented with a set of valid models to choose from, which contain the chosen specifications. In Figure 4.2 on the next page, we can see what the user is presented with after a successful query execution, in this case seventeen different models. In this run, the user have searched for a model with diesel fuel, automatic transmission, four or more gears and a maximum of six seats. We also have the values and URIs for each of these valid models, which means that the application could present the user with more information than it does today.

There are done some pre-computations to make the application a little more effective. This includes extracting all the model names from both ontologies, initiate all possible partially defined products as a PartialCar

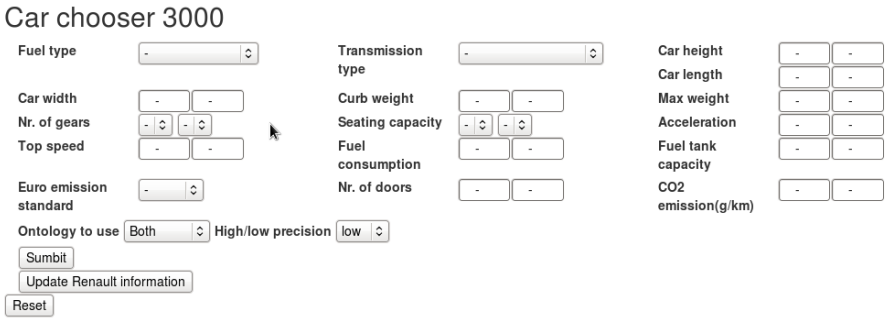

Figure 4.1: Initial view for a user

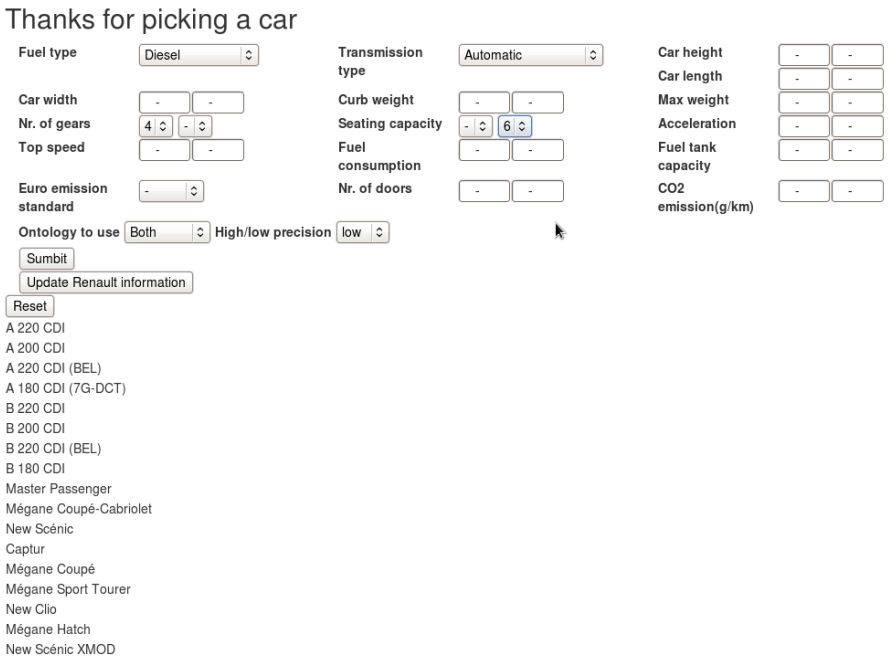

Figure 4.2: The view after a search

object and creating some internal structure. These pre-computations were done so that it would be possible to present the user with the name of each car model, not just the URI. We will also describe more in depth how the car model search is done later on.

4.2 Daimler

Daimler is a German multinational automotive corporation[6]. In this thesis only a small part of the car domain at Daimler has been used.

4.2.1 Choosing ontology

Since the data we received from Daimler came on a non RDF serialization, we had to do semantic lifting [7]. This meant that we also needed an ontology to structure the data. As shown in Listing 4.2 on page 41, we chose to use the ontology Volkswagen had made for this purpose, using both VSO and COO to represent the data. VSO, as stated earlier, was made by Martin Hepp as a car extension on top of GoodRelations. COO on the other hand was made in collaboration between Martin Hepp and Volkswagen to make a specific car specification ontology. The previous technical lead for Volkswagen's web page[8], William Greenly, said in an interview in 2011 "This car options ontology is relevant to any car manufacturer". [48] Since Renault and Volkswagen are the only car manufactures who have published their data by using semantic web at some point, it was easy to choose the work of those car manufactures for comparison. Unfortunately Volkswagen canceled their project and removed their data which lead us to parsing the data from Daimler into Volkswagen's ontologies.

4.2.2 Lifting the data

Daimler provided the thesis with data about their A-Class and B-Class cars. The format of the data was on Extensible Markup Language(XML) in two different files, one for A-Class and one for B-Class. The information in the XML files can be divided into these main groups.

- Technical data
- Dimension drawings
- Color materials
- Tourque curves
- Equipment lists

[6] http://en.wikipedia.org/wiki/Daimler_AG

[7] Semantic lifting refers to the process of associating content items with suitable semantic objects as metadata to turn "unstructured" content items into semantic knowledge resources. [14]

[8] volkswagen.co.uk

- Equipment and validity
- Pictures and text descriptions

The most important groups from these are *Technical data* and *Equipment and validity*. These two groups contain the information about each model and what kind of specifications these models may have. Unfortunately there were no distinct information about compatibility between specifications in their data. These sections also contained information about what extra equipment came as standard and what was optional. It also described what extra equipment that was not valid.

In this thesis extracting the technical data and specifications were emphasized. This means that from the list of groups, *Technical data* and *Equipment and validity* were the sections that were used for our investigations.

XML parsing tools

To continue our investigation, we needed an XML parsing tool to parse the Daimler XML. In almost every programming language there is a module to parse XML. In Java there is *dom4j*, in C there is *Expat* and in python there is *ElementTree*. There are of course many more than just one XML parser in each of those languages, but these three are the most renown ones.

W3C also has their own XML transformation language. In 1999, Extensible Stylesheet Language Transformations (XSLT) became a W3C recommendation. XSLT is a functional programming language used to specify to convert an input XML document into another text document. This can be used to convert an XML document to another XML document, but it can also be used to transform XML documents into any other format. [18] It is easy to use and one do not need any prior programming language knowledge to use it. One only need knowledge about how an XML file is structured and some XSLT functions. XSLT it very useful for making a general converting document for a large amount of documents on the same format.

Daimler XML

Daimler delivered two XML files with around 100k lines each. One about A-Class cars and one about B-Class cars.

The task at hand was to use semantic lifting to add semantics to the data from Daimler. We needed to get the data on an RDF serialization. We chose Turtle as the RDF serialization because it can contain more information on less lines than RDF/XML. This would prove to be useful with the large amount of data from Daimler. This made it more human readable which helped understanding the data while we were parsing them. In appendix A you can find a link to both RDF graphs which were parsed out of the XML files about A-Class and B-Class car models. In Listing 4.1 and 4.2 one can see a small snippet from the XML and what it is equivalent to in the RDF

```
<td_model pit_id="33795">
    <td_model_series uniquename="t161500f030_2837">
        <![CDATA[176011]]>
    </td_model_series>
    <td_model_designation uniquename="t161500f030_2837_1">
        <![CDATA[A 160 CDI]]>
    </td_model_designation>
</td_model>
<td_item line_title="Fuel" line_id="70050">
    <term uniquename="t163600f033_2837_12">
        <![CDATA[Fuel]]>
    </term>
    <value uniquename="t163600f021_2837_12">
        <![CDATA[Diesel fuel]]>
    </value>
    <unit uniquename="t163600f034_2837_12">
        <![CDATA[]]>
    </unit>
</td_item>
```

Listing 4.1: Small snippet from the Daimler XML file about A-Class cars

```
daim:A160CDI a coo:Derivative, vso:Automobile .
daim:A160CDI rdfs:label "A 160 CDI" .
daim:A160CDI vso:fuelType dbpedia:Diesel .
```

Listing 4.2: The RDF equivalent to the Daimler XML snippet

serialization. As we see in these listings the amount of lines gets shortened quite a bit without losing any relevant information.

As presented earlier, there were a lot of options when it came to parsing the XML data from Daimler. The deciding factor was that the parsing was only needed once and it had to be done fast without many lines of code. This is why we did not chose to use XSLT because there was no need to parse more than two documents once. There were also a lot of redundant information in the XML files that we did not need. It was because of these reasons we chose to use python and the ElementTree package. [44] It was easy to use and allowed us to parse the XML fast with a small amount of code. Our prior knowledge with python was also a deciding factor since we did not need to use any time learning new semantics. The downside was that the python script became hard coded which meant that it would only work for an XML file on this particular format. This means that if there were any changes in the XML structure, we would need to change the script. The upside was that both XML files provided by Daimler came on the same format, which meant that the script worked for both.

Added functionality

We chose not to represent all the information found in the XML file. This was because there were a lot of information which did not have any corresponding information in the Renault data set and would not be useful for our application with this use case. An example would be what kind of engine oil cooler a car model could have. This is just an assumption since

```
daim:emission a owl:ObjectProperty  ;
              rdfs:label "CO2 emission"@en  ;
              rdfs:comment "The amount of CO2 a
                            motorized vehicle emits.
                            Typical unit code(s):
                            g/km grams per kilometre"@en ;
              rdfs:isDefinedBy <http://folk.uio.no/magnudae/daimler#> ;
              rdfs:range gr:QuantitativeValueFloat ;
              rdfs:subPropertyOf gr:quantitativeProductOrServiceProperty
.

daim:A180CDI daim:emission
             [ a gr:QuantitativeValueFloat ;
               rdfs:label "CO2 emission" ;
               gr:hasValueFloat "105"^^xsd:float ;
               gr:hasUnitOfMeasurement "g/km"^^xsd:string
             ] .
```

Listing 4.3: OWL definition of daim:emission and example of use.

it is probably some users which find this relevant and it can be easily added to the data set at any time in the future.

Even though COO and VSO covers most of the functionality a car may have, it did not cover all the information found in the Daimler XML. This is why we chose to create and add some functionality. VSO offers some classes for creating resources to cover alternative information. In our case this was used for representing emission standard. Here we used *vso:EmissionStandardValue* to create a resource for representing the emission standard, for instance that a car model has emission standard *Euro 5*. Furthermore, there was no way of representing the amount CO_2 emission for each car model. Listing 4.3 shows the OWL property which we had to create to represent these values and an example of how it is used.

The data from Daimler also lacked compatibility between features which meant that before we could make the prototype and test it, we would have to add compatibility between specifications in the Daimler data. In Listing 4.4 on the next page we can see the compatibility triples for the car model "B 180". We chose the pairwise way to add compatibility with COO, because that was the most human readable representation and was easy to query against. A minus with this way is that it demands reasoning to be effective because without reasoning the *coo:incompatibleWith* will not yield all invalid triples.

Data representation

The data from Daimler was represented in an RDF graph for each base model. This means with the current data there is one RDF graph for all A-class car models and one for all B-class car models. The compatibility triples were created in a separate RDF graph because they were created manually. In appendix A one can find all three RDF graphs.

```
daim:B180_transmission_automatique coo:incompatibleWith daim:B180_emission_144_0 ,
                                                         daim:B180_emission_139_0 ,
                                                         daim:B180_emission_134_0 .
daim:B180_transmission_mÃ©canique coo:incompatibleWith daim:B200_emission_144_0,
                                                       daim:B180_emission_149_0 ,
                                                       daim:B180_emission_154_0 .
daim:B180_speed_190_0 coo:incompatibleWith  daim:B180_gearsTotal_6 ,
                                            daim:B180_gearsTotal_5 .
daim:B180_speed_160_0 coo:incompatibleWith daim:B180_gearsTotal_7 .
daim:B180_weightTotal_1950_0 coo:incompatibleWith daim:B180_fuelConsumption_7_2 ,
                                                  daim:B180_fuelConsumption_6_9 ,
                                                  daim:B180_fuelConsumption_6_2.
daim:B180_weightTotal_2010_0 coo:incompatibleWith daim:B180_fuelConsumption_5_4 ,
                                                  daim:B180_fuelConsumption_5_7 .
daim:B180_meetsEmissionStandard_Euro_4 coo:incompatibleWith daim:B180_transmission_mÃ©canique .
daim:B180_acceleration_10_4 coo:incompatibleWith  daim:B180_fuelConsumption_5_4 ,
                                                  daim:B180_fuelConsumption_5_7 ,
                                                  daim:B180_speed_160_0.
daim:B180_acceleration_8_4 coo:incompatibleWith daim:B180_fuelConsumption_7_2 ,
                                                daim:B180_fuelConsumption_6_9 ,
                                                daim:B180_speed_190_0 .
daim:B180_acceleration_6_2 coo:incompatibleWith daim:B180_fuelConsumption_5_4 ,
                                                daim:B180_speed_202_0.
daim:B180_acceleration_5_4 coo:incompatibleWith daim:B180_speed_202_0 .
            ] .
```

Listing 4.4: Added compatibility triples.

4.2.3 Issues

Data issues

During the semantic lifting we found some issues and weakness with the data. The XML provided by Daimler had a good structure for setting up the models. However, all the properties like weight, height and fuel consumption used a line ID which were not used anywhere else. The only way to identify the important properties within each model was to read the XML and manually note every place which corresponded to a property in COO or VSO. Another weakness was that there were no compatibility information linked to any specifications. This meant that we had to construct all compatibility triples manually.

Ontology issues

One issue was that the structure of the Daimler data did not fully align with COO. This was because every model from Daimler did not have any clear base model or trim. Every model had a set of possible specifications and features, but they were no derivations of another model or set. This meant that *coo:BaseModel* and *coo:Trim* were only needed to keep the data valid in the COO ontology, not to keep any extra information. In the COO definitions they only refer to a derivative that it should have a trim and a trim should have a base model as a guideline. [28] This would affect the distribution of the data and the possibility to let the derivatives inherit data. This could have been used to let models share data and could help to limit the search for a faster result. This will be further discussed in Chapter 6 on page 73.

4.3 Renault

Renault have data about their car models published on the web today. This is done via an API on the web. They have decided to represent their data as a graph which their API helps third party application traverse. [22] It is because of the vast number of specifications a car model may contain. By Renault's calculations they have around 10^{20} different combinations of specifications which yield a valid car model. [21] This is why they have chosen to represent their data as an API.

The API is a way to traverse the RDF graph of possibilities and find a smaller result set of cars. This also allows Renault to handle the reasoning on their side, which can be a big operation with 10^{20} of combinations. They also state that hiding the reasoning in their API could relieve a lot of performance pressure from applications wanting to use their data. This decision was built upon observations that the environments wanting to use their data are still young and do not have the reasoning capabilities which would be expected.

They have represented their data with the Configuration Ontology (CO). The ontology focuses on building an RDF graph to represent all valid configurations for a product model with component constraints. It is also made so that it is not needed to do any reasoning to get a valid result set. They call it a traversal of *Configurations* which will end up with a *Partially Defined Products* (PDP)[9]. Their thoughts of how it is meant to be used is for a user to choose configurations on the fly. This means that for each selection the user gets one step closer to a fully defined product.

All possible specifications for a car model has a finite set of options. For instance with fuel type where the options to choose from are gasoline, diesel, electric and gasoline-electric hybrid. [20] With this it is possible to create a GUI where one iteratively choose the specifications one wants in a car. Renault has a prototype on how this user interface could look like and act on their Configuration Ontologies site[10], but it has been non-operative since December 2013. When it was operative it first presented the user with an option to choose which car model he or she desired. After that there were several options to choose from, like fuel type, transmission and other specifications. The data would then exclude or include new data as the user chose the specifications. Renault are also working on allowing to query their API with a list of specifications as seen below.

$$configService?chosenSpec = spec1\&chosenSpec = spec2$$

Here two specifications are chosen. This query will return the list of specifications which are valid after these two were chosen.

[9] Partially Defined Product is a way of defining a product without using all of the features. For instance a car which may have many features, but we describe it as a car with mp3 and sunroof.

[10] http://www.semanlink.net/2012/cold/configurator.html

4.3.1 Data representation

Here we will take a look at how Renault have chosen to represent their data about car models and their specifications. Since there is a underlying graph through all the configuration possibilities, there is also a default starting point.

`http://uk.co.rplug.renault.com/docs#this`

The road to a fully defined car

At the default starting point one will be presented with all the car models Renault has present in their API, one can for instance find the *Renault Captur* here. There are also links to each individual car model's lexicon. In each lexicon one can choose between different specifications which may lead to a valid configuration. Here one can choose a specification and then check in the current configuration if it is a valid specification for the particular model. In the current configuration there are several possible specifications to choose from which will link to new configurations. There are also stated if some specifications are impossible in the current configuration. These are represented by either the *co:possible* or *co:impossible* property. In our prototype the user can choose all specifications at once, which mean that we can end up with a composition of specifications that a lot of car models do not support. To validate this, we first check the lexicon if the car has the possibility for that specification value, then we check if that particular value can lead to a valid configuration. This is an iterative process until all the wishes of the user are tested against all the models.

Resource values

To choose a specification one will be presented with some options of values. For instance that one can choose which kind of gearbox, how much CO_2 emission and many other specifications. In the Renault representation there is only one way to determine which specification is representing for instance fuel type. That has to be done with string comparison. In the data there is two ways of finding a specification. All configuration variables have a label indicating what specification they represent. They also have a property *co:confVarId* which contain a string ID that determines what specification type it contains. In Listing 4.5, it is shown the configuration variable for fuel type in one car model's lexicon. Further, one has to match the value from the specification with what the user may have chosen. This is done within a specification instance which holds the particular value. Here the actual specification value is stored in another label as seen in Listing 4.6. If we save this specification's URI we can use this to check if there is a valid configuration link in the current configuration. In Listing 4.7 we can see that there is a possible link to a new configuration if we choose the fuel type Diesel. This will then lead us to a new configuration, or state, in the graph.

```
:var_PT1628
  a        owl:Class , co:ConfigurationVariable ;
  rdfs:label "Fuel Type"@en ;
  co:confVarId "PT1628" ;
  co:hasValue
    <http://uk.co.rplug.renault.com/spec/BAm/PT1628_diesel#this> ,
    <http://uk.co.rplug.renault.com/spec/BAm/PT1628_unleaded_petrol#this> ;
  co:lexicon :this ;
  owl:oneOf
   (<http://uk.co.rplug.renault.com/spec/BAm/PT1628_diesel#this>
    <http://uk.co.rplug.renault.com/spec/BAm/PT1628_unleaded_petrol#this>) .
```

Listing 4.5: One car model's Configuration Variable representing Fuel Type

```
<http://uk.co.rplug.renault.com/spec/BAm/PT1628_diesel#this>
      a        co:Specification , :var_PT1628 ;
      rdfs:label "Diesel"@en ;
      co:specId "PT1628_diesel" .
```

Listing 4.6: One car model's Specification representing Diesel fuel

After this we can start the process of choosing a new specification and do the same again, but this time with a smaller set of specification options.

In both these listings the resources have their own ID. These two ID properties were earlier not mentioned in the Configuration Ontology reference, but were later on added to the reference. [20] These IDs identify what a Configuration Variable and a Specification contains. For instance if a configuration variable has the ID "PT1628" it contains specifications about fuel type.

In the ontology's reference it is mentioned that it is possible to represent each variable in different ways than with their classes. One example they show is with the use of the Vehicle Sales Ontology. Since the ontology is generic one could use any kind of ontology to describe the desired specification.

In Figure 4.3 on the next page, we can see an example of how the data interact within the Renault data graph. The figure shows us that every configuration have a lexicon which keeps tracks of all the possible specifications. This lexicon is the same for every configuration about this particular model. The configuration also has several edges to possible configuration links. The figure only shows the graph that unfolds when picking one specification from a configuration. At this point one would have to choose a specification from the lexicon and then check if there was a valid link in the current configuration. One possibility here is to choose the fuel type Diesel. If we do, we will end up in another configuration which has

```
<http://uk.co.rplug.renault.com/c/BAm/AAI#this>
    cold:possible
      [ a        cold:ConfigurationLink ;
        cold:linkedConf <http://uk.co.rplug.renault.com/c/BAm/AAIZA#this> ;
        cold:specToBeAdded <http://uk.co.rplug.renault.com/spec/BAm/PT1628_diesel#this>
        ] .
```

Listing 4.7: Showing a possible configuration link if Diesel is chosen

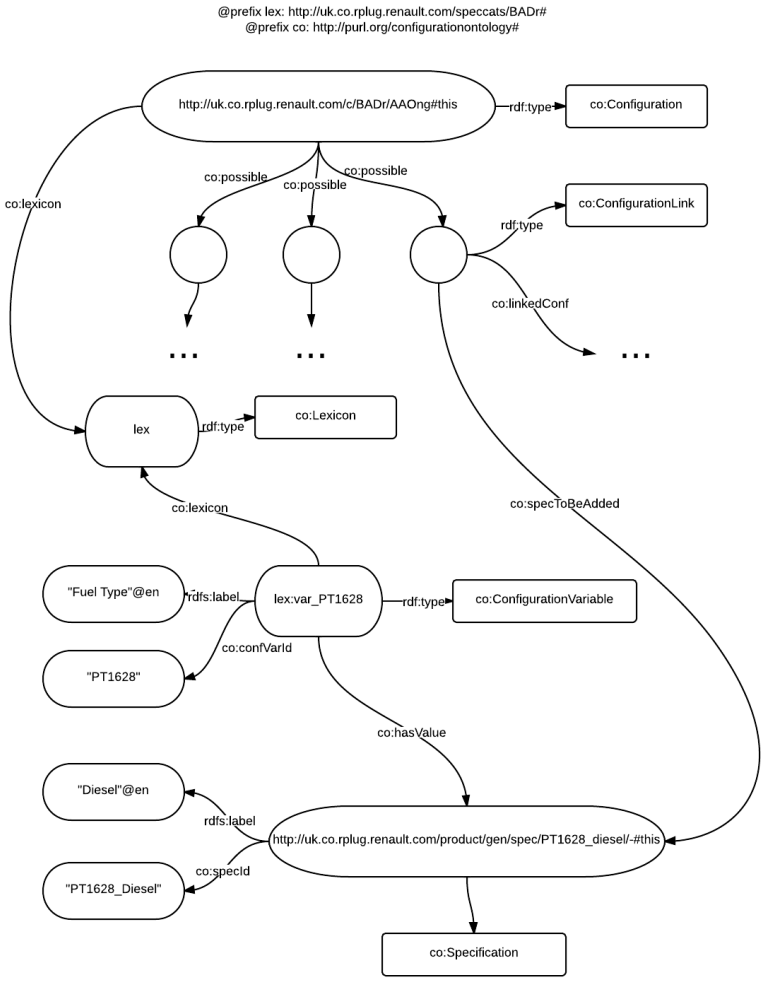

Figure 4.3: RenaultValueChart

several new possibilities, but only possibilities which are compatible with the fuel type Diesel.

In comparison to the ontology diagram in Figure 3.3 on page 34 we can see that the actual data representation that Renault has on their end is a lot bigger and complicated than the ontology implies.

4.4 Alignment

Before making the search feature in the application we had to align the data so that we would not need to make two different applications. First we had to decide which way to align. Would it be satisfactory to align from CO to VSO and COO or would it be better to do it the other way around.

```
daim:A200CDI_gearsTotal_5 a gr:QuantitativeValue ;
        co:confVarId "PT47" ;
        rdfs:label "Forward gears (total number)" ;
        gr:hasValue "5"^^xsd:int ;
        gr:hasUnitOfMeasurement "C62"^^xsd:string .
```

Listing 4.8: Daimler specifications after alignment attempt

4.4.1 VSO/COO -> CO

Our first guess was to align from VSO and COO to CO by adding the right configuration variable ID to each value. To manage this we could not use OWL axioms since we had to search the Daimler graph for properties and then add a literal value. That is why we chose to use SPARQL constructs since it had the functionality needed to do the alignment.

After the first alignment attempt we ended up with these triples in the Daimler graph as seen in Listing 4.8. At the first glance this looked correct, but then we realized that each specification is identified with *co:specId* not *co:confVarId*. This meant that if we wanted to align the specifications we would have to access the value within each specification to create the correct ID. That is because Renault makes their IDs by adding the value to the configuration variable ID as seen in the example below:

```
<http://uk.co.rplug.renault.com/product/gen/spec/PT1628_diesel/-#this>
        a           cold:Specification , :var_PT1628 ;
        rdfs:label  "Diesel"@en ;
        cold:specId "PT1628_diesel" .
```

This made it impossible to either use OWL axioms or SPARQL constructs since we would have to generate complex IDs. It would need a programming solution to be feasible, but even then the alignment would not yield any good results because it still would demand string comparison to find similar instances.

4.4.2 CO -> VSO/COO

After trying for some time to figure out the best possible alignment option, it occurred to us that Configurations Variables were just the same as the VSO properties except one was a class and one was a property. A configuration variable was just a place holder for one or more specifications and the type of the specifications. This would be the same as having several specifications in the range of a VSO property. In the example below one can see a configuration variable and a VSO property represent the same thing, and that was what we would try to align.

```
#CO representation of fuel type
:var_PT1628 a           co:ConfigurationVariable , owl:Class ;
        rdfs:label      "Fuel Type"@en ;
        co:confVarId    "PT1628" ;
        co:hasValue     <http://uk.co.rplug.renault.com/product/gen/spec/PT1628_diesel/-#this> .

#VSO representation of fuel type
daim:A200CDI vso:fuelType daim:A200CDI_fuelType_Diesel_fuel .
```

To do this we used SPARQL constructs to add properties from the lexicon to each specification. We made one construct and then split it

```
CONSTRUCT {
        ?lexicon vso:fuelType ?fuel .
        ?fuel a gr:QualitativeValue .
        ?lexicon vso:gearsTotal ?gears .
        ?height a gr:QuantitativeValue
} WHERE {
        [ co:lexicon ?lexicon ;
                co:confVarId "PT1628" ;
                co:hasValue ?fuel ] .
        [ co:lexicon ?lexicon ;
                co:confVarId "PT47" ;
                co:hasValue ?gears ] .
}
```

Listing 4.9: A snippet from the construct used in aligning the ontologies

into two to make the workload a little smaller. We did this because there were a lot of different resources to match since each configuration variable had their own URI. In the first construct the Cartesian product[11] became so large that the construct never ended within a reasonable time. In Listing 4.9, we see a small snippet from the resulting two constructs. Here we detect the variables for fuel type and the number of gears. Then we added the property *vso:fuelType* and *vso:gearsTotal*, and linked them to the proper specifications. This could be done because *vso:fuelType* means the same as *co:hasValue* combined with the configuration variable ID "PT1628". We also added the classes QuantitivativeValue and QualitativeValue to the proper specifications to satisfy the range of the VSO properties. In Figure 4.4 on the next page, we can see how the ontology looked after the alignment. In comparison to the Figure 3.3 on page 34 we can see that the configuration variable is gone and instead we got VSO properties linking directly to the specifications. This means that instead of four triples defining a specification, we only need one because each VSO property is defined in the ontology. This also makes the search more universal because the property has the same URI for all models. Before the alignment one would have to detect each configuration type with string comparison of the label and not the URI. This makes the data more machine readable which is one of the key concepts of semantic technologies.

In the end, we can see that the ontology definition became bigger after the alignment, but the in the actual representation there would be fewer triples and they would be more understandable.

[11] http://en.wikipedia.org/wiki/Cartesian_product

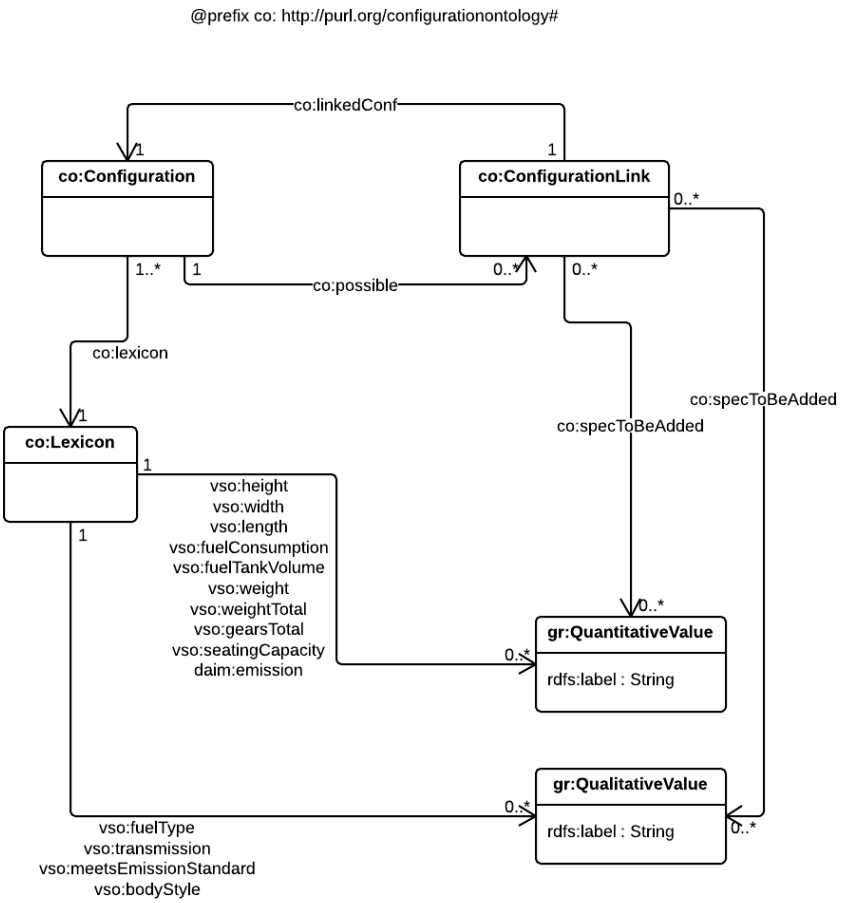

Figure 4.4: Figure showing CO after alignment

4.5 From data to application

As stated before, we chose to use the programming language Java and the framework Jena to develop the application. We used 2/3 of the main components in Jena. The only component that was not used was the *Triple Store* component, which contain Fuseki and TDB. We needed the two other components to do SPARQL queries, handle RDF graphs, do reasoning and to get full RDFS and OWL support. The application took about a month with a full time workload to finish. The application contains approximately 1350 lines of Java code distributed over these packages:

- controllers: ≈ 250 LoC
- matching: ≈ 800 LoC
- utils: ≈ 300 LoC

To manage all the files and packages we used Apache Maven, a software project management and comprehension tool. [8] We wanted the application to be able to handle different HTTP requests so that it would be as realistic as possible to an application on the web today. To realize this we chose to use Spring, which is a framework for making fast, simple and flexible JVM-based systems and applications. [9] Our application uses the MVC, model-view-controller, part of spring which is the component for developing a web application and using the HTTP protocol. Further on in this section we will take a look at how the application execute the data search and query building.

4.5.1 Post to form

First a user has to perform a HTTP post to the service with a form. This form can contain everything from a full form with all specifications declared to a form with only one declared specification. The data from Renault is at this point ready for queries, but the data from Daimler needs to be reasoned over before it can be queried. The possibilities are either forward or backward reasoning. In the application we chose to do forward reasoning due to the simplicity. This is done so that all incompatibility triples are inferred. In the example shown in Listing 4.10, we see a code snippet from the reasoning in the application. We chose to use the OWL reasoner contained in Jena because it had the functionality we needed from a reasoner. One could have used a specialized reasoner to get better performance since the OWL reasoner applies more rules than needed. This results in more inferred triples. This will be discussed in more detail later on.

After reasoning and loading in the data models, the application proceeds to do the actual search. To do the actual search we programmed it so that every car model had their own thread. This was done to make the experiments faster than they initially were. This would not affect the performance comparison between the ontologies because it was programmed identical for both.

```
Model model = FileManager.get().loadModel(<Data Model>);
model.add(FileManager.get().loadModel(<Compatibility triples>));
model.add(FileManager.get().loadModel(<COO ontology>));
InfModel inferredModel =
   ModelFactory.createInfModel(ReasonerRegistry.getOWLReasoner(), model);
```

Listing 4.10: How reasoning was done in the application

```
SELECT ?specification ?vsoValue
WHERE {
  <http://folk.uio.no/magnudae/daimler#B200> vso:fuelType ?specification .
  { ?specification gr:hasValue ?vsoValue . }
UNION
{ ?specification gr:name ?vsoValue . }
FILTER NOT EXISTS  {
   ?specification coo:incompatibleWith
      <http://folk.uio.no/magnudae/daimler#B200_transmission_automatique> .
  }
}
```

Listing 4.11: Query for data in the VSO/COO graph

4.5.2 Queries

Each thread iterates through the form of defined specification and queries the RDF graph for each iteration. The queries for both ontologies were almost the same after the alignment, but with some exceptions. One exception was that the queries for VSO/COO has a FILTER NOT EXIST clause at the end of the query to check for incompatibilities. In Listing 4.11, we can see a query from VSO/COO where transmission is already chosen and at this point we are querying for fuel type. We chose to save both the value and the specification URI. The value was saved because we needed a way to check if the value in this model was valid with the data the user declared. It was easier to do the value matching in the application than in the query due to the diversity of values in the RDF graph and that we wanted an interval search. The specification URI was saved because it was needed to check compatibility in later queries. This was also different from the CO query. In Listing 4.12, we can see a query against the CO data. Here it was not needed to save the specification URI. This was because the compatibility check was not done in the car lexicon itself. Another exception was that when querying the CO data for a value, we had to query for *rdfs:label* and not *gr:hasValue* or *gr:name*.

```
SELECT ?value
WHERE {
  ?lexiconURI vso:transmission ?specification .
  ?specification rdfs:label ?value .
}
```

Listing 4.12: Query for data in the CO graph

4.5.3 Compatibility

There were a big difference in how the ontologies had solved the compatibility constraints. This was an issue when programming a solution for it in the application. As we saw in Listing 4.11, the compatibility issue in the VSO and COO ontology was easily solved with adding a *FILTER NOT EXIST* clause in the SPARQL query. The only thing the application had to do was to save the previous chosen specifications. With the CO ontology it had to be done differently because each lexicon did not have any information about the compatibility. As described earlier the compatibilities in CO are handled with configuration links. This means that for each chosen specification one has to move one step along in the configuration graph. At each configuration in the graph there are several possibilities and a chosen specification from the lexicon has to be a possibility in the configuration to proceed. This is how the compatibility is handled in the CO ontology.

In the application we then had to collect the value from the lexicon with one SPARQL query. Then do another query in the current configuration to check if it was a valid value. If the value was valid we would also with that query get the new configuration through *co:linkedConf*, as seen in Listing 3.7 on page 31. In order to proceed after choosing a manual transmission, we had to save the new configuration as the next step in the search process. Each step was identified with an URI which we stored in a list. This list represent one path in the graph. This was because when we encountered a specification that was not valid in the current configuration we had to be able to restore the last configuration and try the next valid option.

4.5.4 Finalizing

In both ontologies, the specification search was done through a recursive function called *recursiveCompatibilityCheck*. It iterated through the form of specification and did the querying described in this previous sub-section. When the function had matched all values in the form, it saved the specifications found as a valid way to build this car model. Then it went on iterating to find other valid ways to build the car model through the recursive function.

The user was in the end presented with all the car models which had at least one valid configuration of itself with the user's defined specifications. In figure 4.2 on page 38, we can see what the user was presented after a successful search for these specifications:

- Fuel type = Diesel
- Transmission = Automatic
- Nr. of gears > 4
- Seating capacity < 6

VSO/COO average values

Specification	Average value	Nr. of values
Weight	1451.25	≈ 3.9
Weight Total	1981.25	≈ 2.6
Nr. of Gears	≈ 6.105	≈ 2.3
Seating capacity	4.0625	1.625
Emission	128.6875	3.75
Fuel consumption	5.33125	3.75
Doors	5	1
Fuel type	NaN	1
Transmission	NaN	≈ 1.3

CO average values

Specification	Average value	Nr. of values
Weight	≈ 1381.96	≈ 5.2
Weight Total	≈ 2115.1	≈ 4.24
Nr. of Gears	≈ 5.72	≈ 1.2
Seating capacity	≈ 5.01	≈ 1.3
Emission	≈ 140.43	≈ 6.4
Fuel consumption	≈ 5.61	≈ 5.8
Doors	≈ 4.24	1.3
Fuel type	NaN	≈ 1.54
Transmission	NaN	≈ 1.6

Table 4.1: Data mined from the VSO/COO and CO RDF graph

4.6 Data mining

The last thing we did with our application was to mine the ontologies for data that were useful for the experiments and discussion, which we will take a closer look on in the remaining chapters. This included finding average values and number of values. It was a simple task which involved querying the RDF graphs for values and aggregating them. In Table 4.1, we can see the data mined from the RDF graphs. For fuel type and transmission there were no average value due that these specifications are qualitative values.

Chapter 5

Results

The results and evaluation of the experiments on the ontologies are presented in this chapter. The experiments are the backbone of the discussion and conclusion. To do this evaluation wecould have used the practice of benchmarking which is widespread as a appropriate way of testing an application. It would often consist of several micro benchmarks, which intend to do different tasks against an application and measure performance. This performance testing often consists of testing several factors around application performance like, scalability, throughput and accuracy. Depending on what is important for that particular use case.

This is common approach and works great for application studies, but we are more interested in finding the significant factors which affects the ontologies. With this in mind benchmarking comes across as not precise enough to handle the amount of factors which might affect the end result. We could set up several benchmarks and be happy with the results, but unless we test every case, which would be inefficient, we can not be sure that these results are right. In the industry it is sufficient with results which indicate what is right and what is wrong, but in a paper by Kjernsmo and Tyssedal (2013) they argues that benchmarking is flawed as empirical research. [33] This is why we have chosen to employ techniques from the field of statistics known as Design of Experiments(DoE).

In this chapter we will describe the background for our experimental approach. We will also present all our findings after using the application we created to test the ontologies. We will present the small scale experiments first before moving onto the more larger experiment done against the application.

5.1 Design of Experiments

In this section we will take a look at some techniques from DoE that are relevant to our experiments.

5.1.1 Planning the experiment

As other approaches, there are several steps in doing experiments with DoE. First one has to state an objective which is important to give the experiment a purpose. The next step is to choose a response. The response is the experimental outcome or observation. There may be multiple responses in an experiment. In this experiment there will be one response and that is the response time on a HTTP post against the application. These first two steps are the same for all the experiments in this thesis.

The next two steps of planning an experiment is specific to each experiment itself. This means that these steps will be described more in depth in each experiment because it may vary depending on the actual experiment. The third step in planning an experiment is choosing factors and levels. This is a key part of DoE. A factor is a variable that is studied in the experiment, and to be able to study this factor it is needed to use two or more values of this factor. These values are referred to as levels. This varies from benchmarking because often these levels are not required to do proper benchmarking. The levels will allow us the check each factor for its significance. It is important to identify the key factors in the planning stage. This is to get the maximum effect out of the experiment. Factors may be *quantitative* or *qualitative*. Quantitative factors are often numerical values that represent an interval. For instance the weight of a car is considered a Quantitative value. Qualitative factors are predefined values within a known set of values. In our case a Qualitative factor can be the type of fuel, for instance Diesel.

The fourth step in planning is to choose the experimental plan. Here we will use one specific approach. The approach is called *full factorial experiment* which we will go into more detail about in the next sub-section. There are other approaches like fractional factorial experiment, but they were not applicable or relevant to this thesis. The last three steps of planning the experiment are performing the experiment, analyzing the output and in the end drawing conclusions. More about these steps will be taken individually at each experiment. [45]

5.1.2 Full factorial experiment

A full factorial experiment is when one got k factors and n amount of levels. This means that there is n^k *factorial designs* which result in n^k designs to execute. In the thesis every experiment will only have two levels, but the factors may vary. This means that we will use a 2^k full factorial design for every experiment. The experiments consist of 2^k combinations with k factors. For instance if we have three factors where every factor got two levels we get 2^3 designs, also called *runs*. In Table 5.1 we can see all the possible runs we get from 3 factors. The + and - represent the level for each factor. When running the actual experiment one should use a *planning matrix* to display the actual runs in the experiment. In Table 5.2 we can see the same table as before, just that now it is filled out with actual factors and their different levels.

Factor 1	Factor 2	Factor 3
+	+	+
-	+	+
+	-	+
-	-	+
+	+	-
-	+	-
+	-	-
-	-	-

Table 5.1: 2^3 possible runs

Run	Fuel type	Transmission	Speed
1	Diesel	Automatic	100
2	Petrol	Automatic	100
3	Diesel	Manual	100
4	Petrol	Manual	100
5	Diesel	Automatic	200
6	Petrol	Automatic	200
7	Diesel	Manual	200
8	Petrol	Manual	200

Table 5.2: A filled planning matrix with 3 factors

Two key properties of full factorial designs are *balance* and *orthogonality*. Balance means that each factor level appears the same amount of times in all the runs. For instance that the factor Diesel appears the same amount of runs as Petrol. Orthogonal means that two factors level combinations will appear in the same number of runs. As an example will Diesel fuel combined with Automatic transmission appear in the same number of runs as Petrol fuel with manual transmission. We can see in Table 5.2 that both these level combinations appear two times. Diesel and automatic appears in run 1 and 5, while Petrol and manual appears in run 4 and 8.

Experiments like this can also be replicated, which just means that the experimental plan is run more than one time. It is not always efficient to replicate an experiment. Sometimes it can be too expensive to run it more than one time. According to Wu and Hamada(2009) replication is unnecessary for computer experiments because repeated computer runs with the same input give the same output. If it is replicated the plan should be randomized to get the best result. One reason for randomizing is to identify and get rid of *lurking variables*. A lurking variable is a factor that is not taken into account when doing an experiment and that might affect the outcome of the experiment. For instance if one is doing an experiment outside. The outdoor temperature might be a affecting the experiment if not taken into account. [46]

5.2 Experiments

In this section we will present several experiments with different amount of factors. The objective is to find out how the ontologies will perform in comparison and identifying which factors affect the outcome of each experiment. We did the ontology alignment and development of the application to prevent the loss of precision with either ontology.

In the experiments the definition for VSO/COO is *-vso* and for CO is *-co*. These two definitions stand for which ontology are tested in a particular run. In practice this means that when we send in the keyword *-vso* we only run the specification search against the car models represented by VSO/COO, in our case the data from Daimler. This is referred to as the ontology factor in each experiment and is a factor in all the experiments which are presented in this thesis. The ontology factor represents the difference in how the data are represented and how the ontologies are structures, and with this factor we want to determine how this affects the overall performance. We have programmed the application so that the querying of the ontologies are done as similar as possible on both levels. This was done to eliminate any interference from the application. There are some differences in the programming due that the ontologies were different and they will be discussed later on. In Chapter 4, the application and its differences are explained in depth.

Before starting on the experiment we had to find a way to do the actual experiment. We wanted to first create the planning matrix and then execute each run as a HTTP post. To do this we needed to make a small script to

calculate the 2^k runs and perform the HTTP posts. We chose to use the programming language *python* to do this. That was because python was a familiar language and it also had a package for DoE[1]. With the DoE package we just declared the amount of factors and the matrix was created. At that point the matrix looked more like Table 5.1, which meant that we had to substitute the + and - with the levels of each factor. Further in the script we used the package *request*[2] to do the HTTP posts in python. Below one can see the line of code executing the request.

```
r = requests.post('http://localhost:8080/linkedopendata-magnudae/index',
                  data=payload)
```

The *payload* is a dictionary of all the values to post to the application. After the request we saved the response which contained the response time of a run. After each request we saved the run plus the response time and wrote it to file on CSV format. To evaluate the result we used the programming language R with the packages *DoE.base* [24], *FrF2* [25] and *BsMD* [16]. R allowed us to parse the CSV files created from the experiments and output them in two types of graphs. A similarity with all the experiments were that they all contained at least the factors ontology, transmission and fuel type.

To evaluate the graphs we have used a method in the *FrF2* R package called *DanielPlot*. This method assumes that the estimated effects are normally distributed with means equal to the effects. The means of all estimated effects are zero. Resulting in a plot where the estimated effect would end up on a straight line. This plot is then testing whether all the estimated effects have the same distribution. All deviations of the straight line indicates a significant factor. In our results we have used the absolute value of the effects, also called a half-normal plot. The advantages of using the half-normal plot is that all large estimated effects will end up in the top right corner. This means that the highest performance significance will end up in the top right corner. [47]
Here is a quick explanation of the axes shown in each graph. The X axis will show the absolute effect of each factor, shown in time. The values on the X axis is represented in seconds which means that the a plot will have a significance measured in seconds. The Y axis shows the half-normal scores. There are two way of representing the graph and that is with either half-normal plots or normal plots. With half-normal the effects on the X axis are shown with the absolute effect. This means that all factors are placed with a positive half-normal score. This is to avoid visual misleading scores because with the normal scores, deviations from the line on negative scores might confuse the reader who are evaluating the charts. The scores them selves are a measure for which factor that is the most significant. We have also set the *alpha* to 0.05 which means that we will tolerate a false percentage of 5%.

All of the planning matrices, graphs and results will be presented with each experiment later on in this section. The discussion and interpretation of the results will be presented in Chapter 6. Before starting the experiments we will present abbreviations for each factor because their full name made the graphs totally illegible. These abbreviations will be used

[1]http://pythonhosted.org/pyDOE/
[2]http://requests.readthedocs.org/en/latest/

in the graphs and tables further on in this section. The results from all the experiments can be found in appendix C.

- Ontology = O
- Fuel Type = FT
- Transmission = T
- Weight = W
- Total Weight = WT
- Emission = E
- Nr. of Gears = G
- Seating Capacity = SC
- Fuel Consumption = FC
- Doors = D

5.2.1 Experiment with three factors

The first experiment that was done, was the experiment with only three factors. The goal here was to see how the ontologies behave with a small amount of querying and factors. Table 5.3 show the different factors and their levels. We chose to start with a small experiment using only simple factors which had few possible values. That is why we chose transmission and fuel type as factors as well as the ontology factor. With these three factors we got 2^3 number of runs. This meant that we had 8 runs to execute against our application. This experiment was also replicated four times to get the most accurate results. The experiment was small so the replication did not cause any issues. The results were calculated by finding the average response time for each run between all four replications.

The next step was to create the planning matrix for this experiment. As mention in the beginning of this section we used a package in python to create and fill out the planning matrix. Table 5.4 shows all the 8 runs for the experiment with three factors. We also ran a randomized version the experiment to check whether the randomization would influence the experiment. The results were almost equal which eliminated randomization as a factor. Both results can be found in appendix C. After each run the response time were logged to that particular run. The response time were then used to check for significant effect when the experiment were evaluated

Like the rest of the experiments we used DanielPlot to evaluate and a package in R to visualize the result. The Figure 5.1 on the next page shows the results. Here we see no significant factors. Still there are small effects from several factors. In Table 5.5 on page 66 we can see the time effects from largest to smallest. We see that the ontology factor has the largest

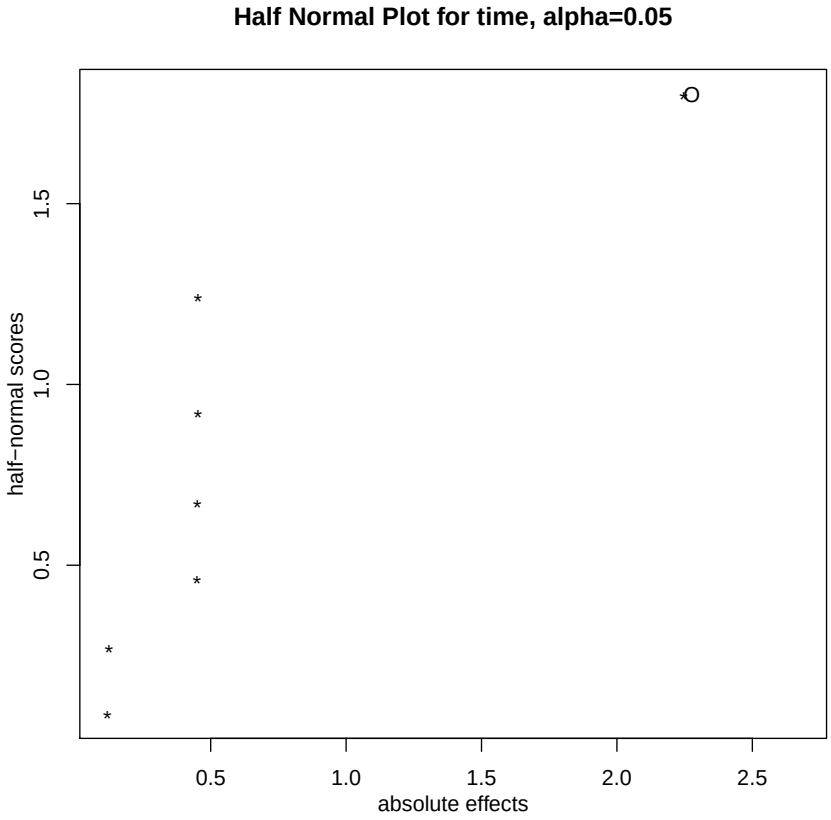

Figure 5.1: Resulting graph for the three factor experiment

effect on approximately 2.2 seconds which means that one of the ontologies overall is 2.2 seconds faster. By reading the results from the linear model it is the factor level *-vso* which has an overall faster response time.

5.2.2 Experiment with four factors

The goal with this experiment was to find the turning point of the ontology factor. Here we also chose to use simple factors like *fuel type*, *transmission* and *nr. of gears*. By simple factors we mean that each car model in the ontologies did not have many values of each factor. We chose to use 6 gears as the middle value for number of gears due that the average number of gears value was closest to 6 from both the Daimler data and Renault. We also chose 6 because a car can not have 5.72 gears. The other values were the same as the experiment for three factors. There is also another difference with the nr. of gears factor. The difference is that that factor is a quantitative value, which means that in our application it can have an upper limit or a lower limit. That is represented with less than(<) and the greater than(>) signs. In further experiments there will be more quantitative values were they will be using the same notation. With quantitative value we chose

to use the average as an upper and lower limit to hit as many values as possible in the RDF graphs. This was done because there was only feasible to run a 2-level experiment.

In this experiment we had four factors which resulted in 2^4 runs. In Table 5.6 on page 67 we can see the planning matrix with all the 16 runs. This experiment was replicated four times because it was not too expensive to run and finished within a feasible time frame. Similar to the three factor experiment, it was also randomized and the results were similar. The results from both the randomized and the non-randomized experiment can be found in appendix C.

In the Figure 5.2 on the facing page we can see the graph representation of the results. Here we got two points that are deemed significant. The first one is the ontology. Here the factor ontology have jumped far up on the top right. If we take a look at the Table 5.7 on page 67 we can see that it is approximately a 3.3 seconds difference between the ontologies, meaning the average difference of all the runs. To find out which ontology that were 3.3 seconds faster we checked the linear model behind the graph. It showed the factor level *-vso*, which means VSO/COO, had a faster response time. There were also some other significant plots in the graph. The interaction between ontology and transmission was deemed significant. As we see in Table 5.7 on page 67 the performance difference with that interaction was minimal, but it indicates that more than just the ontology will affect the results. We also see some other factors and factor interactions being deemed significant, but the effect in time is minimal. By just adding another factor we got some significant results. Furthermore, we will present the results for both a six factor experiment and ten factor experiment.

5.2.3 Experiment with six factors

With this experiment we wanted to further pressure the ontologies to show us how many factors could be searched at once. In addition to the factors used in the four factor experiment, we added *emission* and *total weight*. We wanted to add factors that were more the opposite of the factors in the first two experiments. These two factors have more possible values to search through, which could affect the performance. This was an important aspect to test and a goal for this experiment.

With six factors we got 2^6 runs. The planning matrix, which contains these 64 runs, is shown in Table 5.8 on page 68. We used the same values for the factors that were present in the four factor experiment. The value for total weight was the average between the values in VSO/COO and CO, $(2115 + 1981)/2 = 2048$, as seen in Table 4.1 on page 54. The same procedure was used to calculate the emission value. There the average value was 134.

In the graph 5.3 on page 70 we can see the results from running this experiment. Here we got three different stages. In the first stage there are a lot of factor interactions that do not have any significant effect on the experiment. The majority of interactions are found here. In the second stage we got some significant factor interactions and the factors them selves. We see that there are a majority of interactions around between

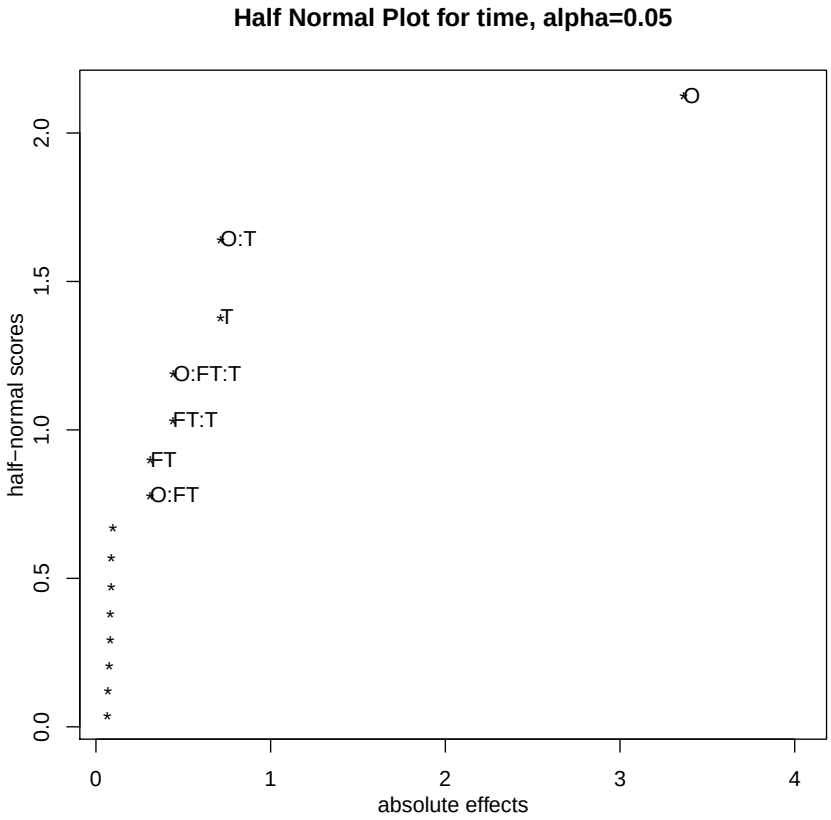

Figure 5.2: Resulting graph for the four factor experiment

the ontology factor and others. For instance the interaction between ontology and emission. We see three factors playing a role here. Those are transmission(T), emission(E) and total weight(WT). Emission is standing out as the most significant factor of those three. In Table 5.9 on page 69 we can see that in the 15 significant interactions, emission is present in 8 out of 15. Most of them are high on the effects list. In the last group seen in the graph, there is one lonely significant factor. That is the ontology factor. It stands out as the absolute most significant factor. The effect ontology has twice as much as the next on the list which is emission. If we compare this graph to the other two we do see a pattern that the ontology will affect the performance, depending on the amount of factors.

The linear model, which the results are visualized from, tells us that it is the factor level option *-vso*, that gives the fastest response time.

5.2.4 Experiment with ten factors

The last experiment we did was a ten factor experiment. The goal here was to see which factors would be deemed significant in a complete car model search. With the results from this experiment we could discuss several aspects of using the different ontologies. Here we added 4 new factors in comparison to the six factor experiment. The newly added factors were *weight*, *nr. of doors*, *nr. of seats* and *fuel consumption*. We used the same approach here as in the previous experiments to calculate the average value for the new factors. The average values can be seen in the planning matrix in appendix B. With 10 factors the amount of runs were 2^10. This results in 1024 number of runs which made the planning matrix too big to show here.

In the graph 5.4 on page 71 we can see the results from running all the 1024 runs. We can see that there are a lot of significant effects present here. There are more than any of the previous experiments, but it is also more factors and factor interactions. Unfortunately there are so many significant factors that there are some of them that are illegible, but most of them have a low impact on the significance graph. Like the six factor experiment we also see here that there are three groups of significant factors. The first group where most of them are placed range from around 2 seconds effect to 11. Here there are a lot of factor interactions and why they are there will be explained in the discussion. The second group is very like the second group we saw in the six factor experiment, the only difference is that the total weight (WT) factor is substituted with fuel consumption (FC). The last group is the same for all the experiments, except the two factor experiment. Here we find the ontology factor. In Table 5.10 on page 69 we can see the top twenty absolute effects. This includes all the significant factors from the last group and the second one, but also 5 from the first group. We can see that emission also here are present in a lot of the entries, but so is ontology as well. The new thing here is the fuel consumption has entered the most significant factors, and is by itself higher than emission. The linear model again tells us that the option *-vso* has a significantly lower response time than *-co*

When we accumulated the results from all the runs we ended up with

the total amount of response time to be approximately 30095 seconds, which is 8 hours, 21 minutes and 35 seconds. This were the total response time after running all the runs. This did not take into account all the other steps during the experiments which took time, like writing results to file and initiating the application. This discouraged the replication of the experiment due to the size and was also a reason why we did not do experiments with more factors.

Factor	Level 1	Level 2
Ontology	-vso	-co
Fuel type	Diesel	Unleaded Petrol
Transmission	Manual Gearbox	Automatic Gearbox

Table 5.3: Factors and levels for experiment 1

Run	Ontology	Fuel type	Transmission
1	-vso	Diesel	Automatic Gearbox
2	-co	Diesel	Automatic Gearbox
3	-vso	Unleaded Petrol	Automatic Gearbox
4	-co	Unleaded Petrol	Automatic Gearbox
5	-vso	Diesel	Manual Gearbox
6	-co	Diesel	Manual Gearbox
7	-vso	Unleaded Petrol	Manual Gearbox
8	-co	Unleaded Petrol	Manual Gearbox

Table 5.4: Planning matrix for experiment with three factors

Factors	Absolute Effect
O	2.2463476
O:T	0.4536135
O:FT:T	0.4534723
T	0.4513712
FT:T	0.4499820
O:FT	0.1240831
FT	0.1189629

Table 5.5: Table of effects for the three factor experiment

Run	Ontology	Fuel type	Transmission	Nr. of Gears
1	-vso	Diesel	Automatic Gearbox	6<-
2	-co	Diesel	Automatic Gearbox	6<-
3	-vso	Unleaded Petrol	Automatic Gearbox	6<-
4	-co	Unleaded Petrol	Automatic Gearbox	6<-
5	-vso	Diesel	Manual Gearbox	6<-
6	-co	Diesel	Manual Gearbox	6<-
7	-vso	Unleaded Petrol	Manual Gearbox	6<-
8	-co	Unleaded Petrol	Manual Gearbox	6<-
9	-vso	Diesel	Automatic Gearbox	-<6
10	-co	Diesel	Automatic Gearbox	-<6
11	-vso	Unleaded Petrol	Automatic Gearbox	-<6
12	-co	Unleaded Petrol	Automatic Gearbox	-<6
13	-vso	Diesel	Manual Gearbox	-<6
14	-co	Diesel	Manual Gearbox	-<6
15	-vso	Unleaded Petrol	Manual Gearbox	-<6
16	-co	Unleaded Petrol	Manual Gearbox	-<6

Table 5.6: Four factor experiment

Factors	Absolute Effect
O	3.36425047
O:T	0.71360066
T	0.71319784
O:FT:T	0.44379428
FT:T	0.44038347
FT	0.31174941
O:FT	0.31029022
O:G	0.09871266
G	0.08849497
O:FT:T:G	0.08782691
FT:T:G	0.08316709
FT:G	0.08228584
O:FT:G	0.07596003
T:G	0.06858634
O:T:G	0.06607516

Table 5.7: Table of effects for the four factor experiment

Run	Ontology	Fuel type	Transmission	Emission	Total Weight	Nr. of Gears
1	-vso	Diesel	Automatic Gearbox	134<-	2048<-	6<-
2	-co	Diesel	Automatic Gearbox	134<-	2048<-	6<-
3	-vso	Unleaded Petrol	Automatic Gearbox	134<-	2048<-	6<-
4	-co	Unleaded Petrol	Automatic Gearbox	134<-	2048<-	6<-
5	-vso	Diesel	Manual Gearbox	134<-	2048<-	6<-
6	-co	Diesel	Manual Gearbox	134<-	2048<-	6<-
7	-vso	Unleaded Petrol	Manual Gearbox	134<-	2048<-	6<-
8	-co	Unleaded Petrol	Manual Gearbox	134<-	2048<-	6<-
9	-vso	Diesel	Automatic Gearbox	-<134	2048<-	6<-
10	-co	Diesel	Automatic Gearbox	-<134	2048<-	6<-
11	-vso	Unleaded Petrol	Automatic Gearbox	-<134	2048<-	6<-
12	-co	Unleaded Petrol	Automatic Gearbox	-<134	2048<-	6<-
13	-vso	Diesel	Manual Gearbox	-<134	2048<-	6<-
14	-co	Diesel	Manual Gearbox	-<134	2048<-	6<-
15	-vso	Unleaded Petrol	Manual Gearbox	-<134	2048<-	6<-
16	-co	Unleaded Petrol	Manual Gearbox	-<134	2048<-	6<-
17	-vso	Diesel	Automatic Gearbox	134<-	-<2048	6<-
18	-co	Diesel	Automatic Gearbox	134<-	-<2048	6<-
19	-vso	Unleaded Petrol	Automatic Gearbox	134<-	-<2048	6<-
20	-co	Unleaded Petrol	Automatic Gearbox	134<-	-<2048	6<-
21	-vso	Diesel	Manual Gearbox	134<-	-<2048	6<-
22	-co	Diesel	Manual Gearbox	134<-	-<2048	6<-
23	-vso	Unleaded Petrol	Manual Gearbox	134<-	-<2048	6<-
24	-co	Unleaded Petrol	Manual Gearbox	134<-	-<2048	6<-
25	-vso	Diesel	Automatic Gearbox	-<134	-<2048	6<-
26	-co	Diesel	Automatic Gearbox	-<134	-<2048	6<-
27	-vso	Unleaded Petrol	Automatic Gearbox	-<134	-<2048	6<-
28	-co	Unleaded Petrol	Automatic Gearbox	-<134	-<2048	6<-
29	-vso	Diesel	Manual Gearbox	-<134	-<2048	6<-
30	-co	Diesel	Manual Gearbox	-<134	-<2048	6<-
31	-vso	Unleaded Petrol	Manual Gearbox	-<134	-<2048	6<-
32	-co	Unleaded Petrol	Manual Gearbox	-<134	-<2048	6<-
33	-vso	Diesel	Automatic Gearbox	134<-	2048<-	-<6
34	-co	Diesel	Automatic Gearbox	134<-	2048<-	-<6
35	-vso	Unleaded Petrol	Automatic Gearbox	134<-	2048<-	-<6
36	-co	Unleaded Petrol	Automatic Gearbox	134<-	2048<-	-<6
37	-vso	Diesel	Manual Gearbox	134<-	2048<-	-<6
38	-co	Diesel	Manual Gearbox	134<-	2048<-	-<6
39	-vso	Unleaded Petrol	Manual Gearbox	134<-	2048<-	-<6
40	-co	Unleaded Petrol	Manual Gearbox	134<-	2048<-	-<6
41	-vso	Diesel	Automatic Gearbox	-<134	2048<-	-<6
42	-co	Diesel	Automatic Gearbox	-<134	2048<-	-<6
43	-vso	Unleaded Petrol	Automatic Gearbox	-<134	2048<-	-<6
44	-co	Unleaded Petrol	Automatic Gearbox	-<134	2048<-	-<6
45	-vso	Diesel	Manual Gearbox	-<134	2048<-	-<6
46	-co	Diesel	Manual Gearbox	-<134	2048<-	-<6
47	-vso	Unleaded Petrol	Manual Gearbox	-<134	2048<-	-<6
48	-co	Unleaded Petrol	Manual Gearbox	-<134	2048<-	-<6
49	-vso	Diesel	Automatic Gearbox	134<-	-<2048	-<6
50	-co	Diesel	Automatic Gearbox	134<-	-<2048	-<6
51	-vso	Unleaded Petrol	Automatic Gearbox	134<-	-<2048	-<6
52	-co	Unleaded Petrol	Automatic Gearbox	134<-	-<2048	-<6
53	-vso	Diesel	Manual Gearbox	134<-	-<2048	-<6
54	-co	Diesel	Manual Gearbox	134<-	-<2048	-<6
55	-vso	Unleaded Petrol	Manual Gearbox	134<-	-<2048	-<6
56	-co	Unleaded Petrol	Manual Gearbox	134<-	-<2048	-<6
57	-vso	Diesel	Automatic Gearbox	-<134	-<2048	-<6
58	-co	Diesel	Automatic Gearbox	-<134	-<2048	-<6
59	-vso	Unleaded Petrol	Automatic Gearbox	-<134	-<2048	-<6
60	-co	Unleaded Petrol	Automatic Gearbox	-<134	-<2048	-<6
61	-vso	Diesel	Manual Gearbox	-<134	-<2048	-<6
62	-co	Diesel	Manual Gearbox	-<134	-<2048	-<6
63	-vso	Unleaded Petrol	Manual Gearbox	-<134	-<2048	-<6
64	-co	Unleaded Petrol	Manual Gearbox	-<134	-<2048	-<6

Table 5.8: Six factor experiment

Factors	Absolute Effect
O	26.606142437
E	13.677912625
O:E	13.647011375
E:WT	11.981633625
O:E:WT	11.949456000
O:T	11.701184688
T	11.655512437
O:T:E	9.149844500
T:E	9.116583125
WT	6.777984563
O:WT	6.749922313
O:T:E:WT	5.839090500
T:E:WT	5.810340250
O:T:WT	4.198654187
T:WT	4.166891812

Table 5.9: The significant effects of the six factor graph

Factors	Absolute Effect
O	58.1785542324229
E:FC	45.6971032050781
O:E:FC	45.5502538925781
FC	38.6140886777344
O:FC	38.5571969433593
O:E	35.3967871113281
E	35.3827983613282
O:T	33.1249425527344
T	33.1154579746093
O:T:E:FC	29.3077075019531
T:E:FC	29.2986405957031
O:T:FC	28.2420862089844
T:FC	28.2325333808593
O:T:E	25.3121093378906
T:E	25.3039236347656
WT:E	11.9191989902344
O:WT:E	11.8908780371094
O:WT:FC	11.7234715332032
WT:FC	11.7140515488281
O:D:E	10.1241599902344

Table 5.10: The top twenty significant effects of the ten factor graph

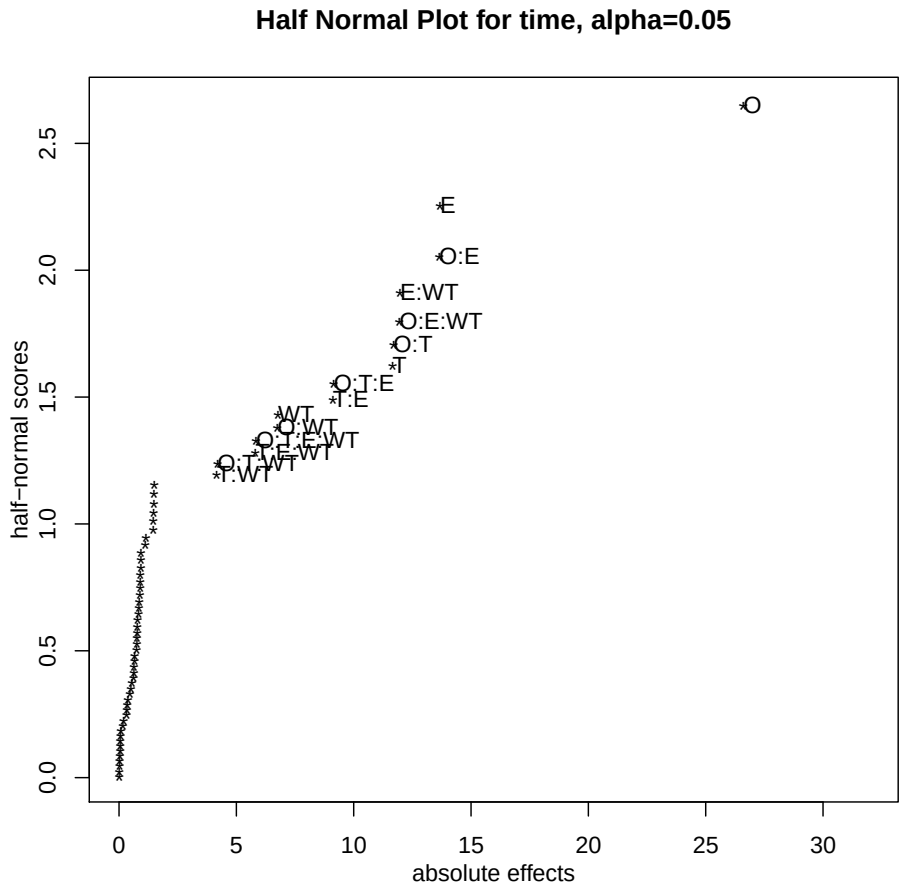

Figure 5.3: Resulting graph for the six factor experiment

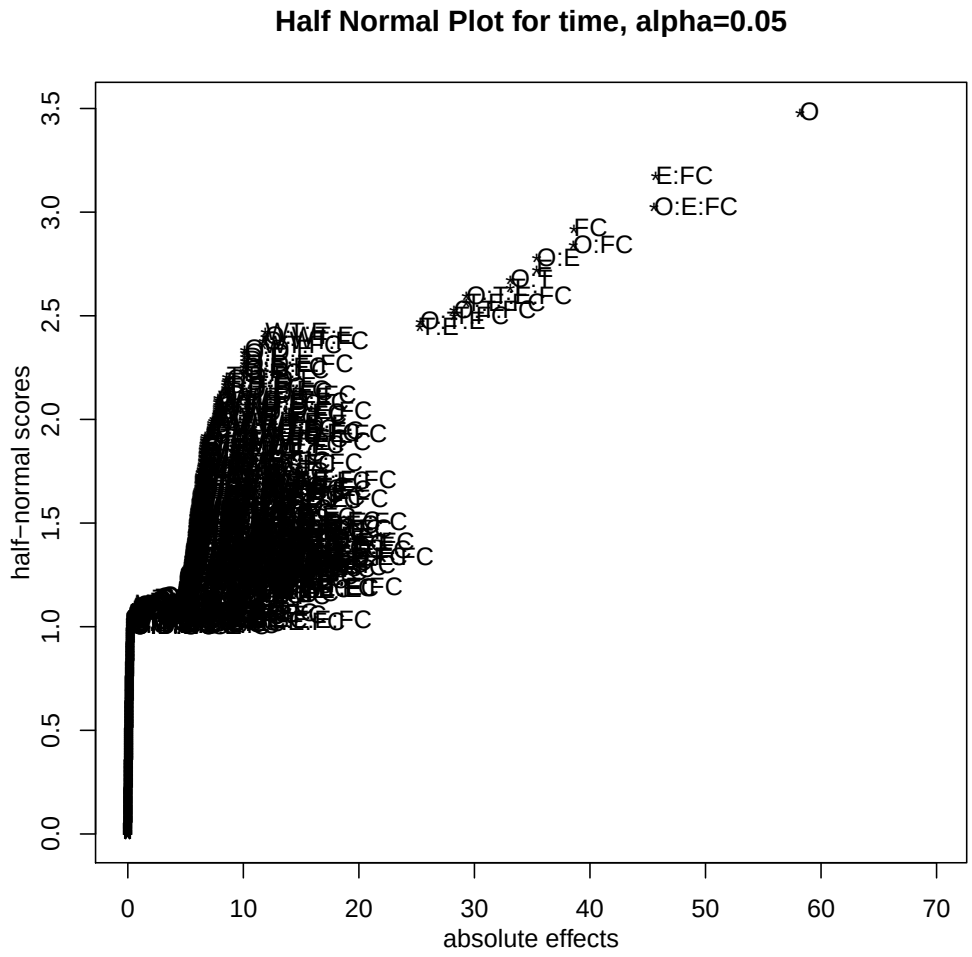

Figure 5.4: Resulting graph for the ten factor experiment

Chapter 6

Discussion

In this chapter we will discuss the results presented in the previous chapter. We will take a look at similarities and differences between the two ontology approaches, and also present possible ways to solve issues that we found in both. In this chapter will use the two different terms, *representation* and *structure*, to discuss the ontology approaches. The structure is the ontology definitions from classes to properties. The representation is how the actual data is represented with each ontology. For instance, how the fuel type *Diesel* is represented. First we will take a look at the different experiments and discuss the findings, then we will discuss the generic versus the specific ontology aspect.

6.1 Evaluating the results

The ontologies are focused on defining a partially defined product (PDP), which in our case is car models. The main difference between them is that CO was made to be a generic ontology which could represent any PDPs. The biggest ontology difference we found was their way of preserving compatibility between components. This difference was the main focus for our experiments and was explored to find out the strengths and weaknesses of both ontologies. Compatibility information is what defines a PDP, which means that this study will not necessarily be applicable to ontologies defining simple products.

The two small experiments were replicated four times to ensure that the results would be as correct as possible. The average response time were calculated between the four replications. This was done to remove any anomalies in the experiments. They were also randomized to check whether the order of each run would affect the performance. There were a slight difference in the *-co* runs with about some tenths of a second. This will be discussed in Section 6.2.2. The two large experiments were not replicated because their workload were too big to replicate. Especially the ten factor experiment was too expensive to run more than once.

The results presented here studies the ontologies based on use cases similar to the one described in Chapter 4. This means that the scope of the discussion is based on the assumption that the ontologies should

manage to query more than one specification. The results show how the data respond to this kind of big operations. The thesis did not investigate every application use case, which is a weakness with this study. Regardless, the use case that was chosen emphasized that the ontologies would need to handle a large workload. This was a reasonable assumption we made because when handling data from several sources, the ontology structure and representation could have a large impact.

6.1.1 The small scale experiments

The first experiment, using only three factors, was done to simulate a small workload on the ontologies. This can represent an application that present the user for a specification one at the time. The configurator Renault has on their web page does exactly that.[1] Our application made the workload on both ontologies to be as equal as possible. The only difference from a programming perspective was that with CO we had to move to a new configuration URI after each specification choice.

In the Figure 5.1 on page 61 we can see that there were no significant effects, but there was one factor that was starting to move away from the others. In the results data found in appendix C we can see that the Configuration Ontology was almost two seconds slower than the VSO/COO runs. Three out of four CO runs kept the response time around two seconds, but one run had to use approximately 3.5 seconds which resulted in an average run time of ≈ 2.3 seconds. In a study done to research the amount of time a user would be willing to wait, they found out that the tolerable waiting time for any information retrieval was approximately 2 seconds. [36] This means that with three factors we were still within the time frame for a good user experience.

One reason that CO was slower than VSO/COO for this experiment is that all the car models represented with CO is placed in a separate RDF graph. Each model had their own lexicon with all the information. This means that an application has to retrieve in all 22 RDF graphs, the amount of cars Renault has present in their API. These 22 retrievals are done right after a HTTP post is detected. This differs from how the cars in VSO/COO are represented. There it is one RDF graph per base model, which means that there were only two RDF graphs to retrieve. In our application this results in 20 more RDF graph retrievals with CO than VSO/COO and because the three factor experiment was so small, it had a higher impact than in the later experiments.

Another factor that would influence the results is the amount of values the search has to iterate over. In Table 4.1 on page 54, we presented all the average values and the average number of values present in the Daimler and Renault data. We can see that CO have 0.54 more fuel types on average to iterate over and 0.3 more transmissions. This is not a lot, but can account for some delay in the response time.

Moving over to the experiment with just one more factor, we hoped for

[1] http://www.semanlink.net/2012/cold/configurator.html

a more significant result. The Figure 5.2 on page 63 is looking very similar to the figure from the three factor experiment. It is only the ontology factor that sticks out from the line of other factors and factor interactions. In Table 5.7 on page 67 we can see that the ontology gap is starting to increase. Here the experiment gave results pointing to that VSO/COO had a ≈ 3.4 seconds faster response time. If we take a look at the results in appendix C, we see that VSO/COO is still keeping the response time below a second. While CO has moved from average of ≈ 2.3 seconds to an average of ≈ 3.5 per run. This happened after introducing another factor, *nr. of gears*. The average number of gears possible for one car model in CO is approximately 1.3 while in the VSO/COO it is 2.3. We also see in the effect table that the number of gears has a minimal significance, only a 0.11 second impact.

Another observation is that the ontology followed by transmission interaction is almost deemed a significant factor. This is most likely a combination of number of values and the order each specification is queried by the application. This will be further explained and discussed in the larger experiments where it becomes a bigger factor and we can see a bigger significance. In the larger experiments the 22 graph retrievals done in a HTTP post can be disregarded as a factor due that it will be insignificant.

6.1.2 The large scale experiments

The last two experiments we define as large scale experiments. This is because the workload and time consumption were significantly larger than the two previous experiments. Both of these experiments were done to simulate a workload for an application were one could define all the specifications at once. Defining all specifications means that a user can query for all specifications that may be present in a product model. This differs from the environment simulated with the two previous experiments. Today there are no complete applications where one can do a car search without defining one factor to start with, like car model or fuel type. Even the application where one can define a car on Volkswagen's homepages, one first have to choose the model before defining the specifications[2].

The first of the large scale experiments were the one with six factors. This experiment added two more factors to each run. In Figure 5.3 on page 70 we can see that there are a lot more significant factors than the earlier experiments. The absolute effects have increased by a significant amount. Similar to the other experiments, this experiment also shows that the ontology factor take the rank as the most significant effect. Another observation is that the ontology factor has twice the effect than the closest factor, as seen in Figure 5.9 on page 69. We see that the difference between the ontologies have increased to 26.6 seconds effect. The average response time for VSO/COO is ≈ 0.23 seconds while the average response time for CO is ≈ 27 seconds. This is a significant amount of time used to query the CO ontology and this difference will be discussed later on in a separate section.

We also got several other significant factors with the six factor exper-

[2] http://www.volkswagen.co.uk/new/range

iment. The closest one to the ontology factor was Emission (E). We also have Total Weight (WT) and Transmission (T) in the group of significant factors. Emission stands out as a most significant factor besides the ontology because it has a larger number of values present in the RDF graphs. In Table 4.1 on page 54 we can see that each car in the CO model has an average of 6.4 values of emission. This is the highest amount of values for any of the factors. Within the VSO/COO models there are an average of 3.75 values. This means that with the emission factor we have to search over more values than other factors. We see that total weight also has a lot of values present in the CO ontology, but the effect is half the effect of emission. The order which each specification is queried also affects the result. This is because a specification can narrow down the possible new values to iterate over. The query order for the six factor experiment is as follows:

1. Gears (G)
2. Transmission (T)
3. Emission (E)
4. Total Weight (WT)
5. Fuel Type (FT)

E comes just before WT, which most likely narrows down the search. We also see T as a significant factor. This is most likely because of the order each specification is queried. T will be queried early and is the deciding factor for which values that will be checked later on. We also see that G is queried early, but has almost no impact on the experiment. This is most likely because G does not narrow down the result as T is. The transmission will most likely decide what the emission rate and fuel type can be, while the *nr. of gears* will not. There are some factor interaction which are present on the top absolute effect list because of both the query order and the number of values.

Number of values in the RDF graph and the order of which each specification is queried are linked together. The order especially, can be altered with programming to make the performance better.

The last experiment done was the one with ten factors. This experiment had four more factors than the six factor experiment. We chose to add factors which represented specifications that were present in every car model in each RDF graph. That was because some specifications like height was not always declared in a car model and then would skew the results. These incidents were present in CO's RDF graphs.

In Figure 5.4 on page 71 we can see the results from this experiment. There are unfortunately a lot of illegible significant factors, but these are the less significant ones. In Table 5.10 on page 69 we see the top twenty absolute effects and similar to all the other experiments, O is on top of the list. The main difference between this experiment and the six factor experiment is that it is a lot closer between the significant factors than before. The impact of the ontology factor has increased from the last

experiment, but there are other factors that now have a much greater impact than before. The average response time of a run done against CO was ≈ 58.5 seconds while a run against VSO/COO had an average response time of ≈ 0.3 seconds. This corresponds with the absolute effect of the O factor to be 58 seconds.

If we take a look at the other factors, Fuel Consumption (FC) has taken the top single factor spot on the list. This can be explained with the same reason as when emission was topping the list in the six factor experiment. Each car has a lot of possible FC values to iterate over. In CO, each car has an average amount of values of approximately 5.8. For VSO/COO it is 3.75. This is the second highest overall after emission. This by itself does not explain why FC is on the top of the list, but when we take a look at when FC is queried it becomes more clear. The order each specification is queried is shown below. We can see that FC comes right before E which means that it will have a greater impact on the amount of values checked later.

1. Gears (G)
2. Transmission (T)
3. Fuel Consumption (FC)
4. Emission (E)
5. Doors (D)
6. Total Weight (WT)
7. Fuel Type (FT)
8. Weight (W)
9. Seating Capacity (SC)

If we take another look at Table 5.10 we see that there are only three single factors except O that are in the top 15 significant factors. After the 15 significant factor the absolute effect drops significantly. The three factors are E, FC and T. E and T are present because of the same reasons as in the six factor experiment. We also see in the top 15 that there are only these three factors present in all the factor interactions as well, which indicates that T, FC and E affects each other. This is logical because if a car has a high fuel consumption it also has a high emission rate. There are some possible explanations for why there are so many significant factor interactions in some of the experiments. The order will affect this as mention above. Another explanation is that we chose an average value to be an upper and lower limit of a factor. We used these as factor levels to avoid an unrealistic high number of levels. This could have opened for more new specifications to check on one level of the factor. As we see in the Table 5.10 on page 69, *E:WT* is higher than *WT* itself. This might indicate that one of the levels of E will open up for more values to traverse in one of the levels of WT. This is

a complex matter which will depend on each product model as well as the interaction between the specifications.

In the next section we will focus on the ontologies and the work done around them.

6.2 The ontology impact

The goal with this thesis was to study the use of a generic ontology versus a product specific ontology to represent complex products. We also wanted to try to work out a recommendation for further use. During the experiments it became clear that the ontology structure as well as the data representation had a major impact on the performance. In this section we will take a look at several aspects on how the ontologies are used today and what can be done differently.

6.2.1 Programming, point of view

An important aspect of the recommendation is how manageable the ontologies are. Renault emphasizes in their paper about their ontology, that *simplicity* for a third party user is a key factor in their choice of solution. They argue that one can not expect strong reasoning capabilities from client agents, and that is one of the reasons why they have chosen to do all reasoning in their API. [22] This is why they chose to represent their data as a graph where one chooses a model, then moves for each new specification to a new valid configuration in the graph. This in theory is a great solution, but this expects a user to start at a specific *starting point* in the graph. Renault has made the assumption that the user wants to choose the car model first, which is one way to solve it. The problem arises when a user has little to no previous knowledge about the car models. Then it might be better suited for a user to start specifying the fuel type before choosing a model. This was one of the major issues working with the Renault RDF graph. Our application wanted to let the user choose which specification to begin with. This meant that we needed to have access to anywhere in the graph. That was achieved through the lexicon of each car, which is a recipe for the product model. Each lexicon had defined all the specifications and had after our alignment VSO properties pointing at them. Based on our investigations, we disagree with the statement made by Renault (2013), that we can not expect any reasoning capabilities from third party users. The third party users are not just an average person, but most likely retailers and other online vendors wanting to enrich their data with constraints directly from the manufacturer. This means that there is already a need for a some computer science background in order to utilize the data properly.

To get access to the lexicon was easy enough, but understanding the data there were not the simplest task. In Listing 6.1 on the next page, we can see a representation about the same specification, represented in both CO and VSO/COO. We can see that both values are represented as

```
<http://uk.co.rplug.renault.com/product/gen/spec/PT1121_118_190_/-#this>
  a            gr:QuantitativeValue , co:Specification , :var_PT1121 ;
  rdfs:label   "118(190)"@en ;
  co:specId  "PT1121_118_190_" .

daim:A180CDI_speed_190_0 a gr:QuantitativeValue ;
  gr:hasValue "190.0"^^xsd:float ;
  gr:hasUnitOfMeasurement "km/h"^^xsd:string    .
```

Listing 6.1: Representation differences in CO and VSO/COO

string literals. The top speed specification in CO is represented as two values in one specification. That is because they have chosen to store both miles per hour and kilometres per hour in the same value. This makes the interpretation of the variable ambiguous. Another problem is that this specification does not have any indication on what it represent. One have to go a step back into the configuration variable to find a label telling us that it represent the top speed. One of the key concepts of semantic web is to make the data more machine readable with properties and value indicators. In the specification from VSO it is both used a property which defines the literal to be a value and there is a value indication telling the machine that this is a float value. There is a property from CO called *hasValue*, but unfortunately it can not be used here. See paragraph 3.4.2 on page 30 for information.

Another example of inconsistency with the literals is the use of the decimal mark within the Renault representation. There are several floating point values which are represented, but these are sometimes represented with "." as a decimal mark and sometimes with ",". This seems like a minor issue, but from a programming perspective there has to be a interpreter for both.

These issues with the data representation can easily be altered just by representing the data more consistently and in line with the semantic principles.

The unit of measurement in CO is represented in the configuration variable which is different from the VSO/COO representation. The configuration variables were disregarded during the alignment due that they had the same point of interest as the VSO properties. That is why unit of measurement is lost in the representation after the alignment.

6.2.2 Ontology performance

Here we will try to find out why CO had a longer response time than VSO/COO in the experiments. The overall performance against the Renault data set were unsatisfactory.

We believe that the problem is based in the representation of the RDF graph constructed by Renault and not necessary in the ontology structure. We can take a look at the difference in how a query is executed against both ontologies. The way to choose a specification is to find if it exist in a car model, then check if that specification is valid with any previously

chosen specifications. The validity is determined based on the previous specifications selection. In the Renault RDF graph this means for n amount of cars we have to do n searches and then move n times for each configuration. This is just for finding one specification. If we want to search for m specifications the worst case scenario, with every specification being compatible, will be $O(m(2n))$. This equation does not take into account the number of values present in each specification. The complete equation would have the number of values for each specification added. The result from this calculation represent the number of SPARQL queries which are done for one run of the application with m specifications. The amount of queries that have to be executed to find a valid car in VSO/COO is $O(m * n)$ which in comparison is only the half of CO.

If every car model in both representations would have the same amount of values for every specification the only theoretical difference in performance would be the double searches. Unfortunately that is not the case and it would unlikely occur in a real life scenario. If we take another look at the Table 4.1 on page 54 which contains the average number of values in each ontology representation, we can see that in CO there are overall more values present. This will contribute to some of the performance issues with the CO representation. That is because for each valid value, it has two more SPARQL queries and retrievals to execute. It should not alone be responsible for the effect of around 58 seconds in the ten factor experiment. This in conjunction with the compatibility validation is what delays the response time heavily within the Renault RDF graph.

There are some *lurking variables* in our experiments. One of them were the performance of the Renault servers. It may have been performance differences on their servers. This can be seen in the randomized and non-randomized runs on the two smaller experiments. Even though the difference was small, it indicates that servers will affect the performance. Another lurking variable which affected the experiments was the order each specification were queried. This was not taken into account as a factor in the experiments. After analysing the experiments, the order has become a part of the discussion of why there are so many significant factor interactions.

A possible bottleneck in the VSO/COO RDF graph is that it depends on reasoning to be effective. In our case there were needed reasoning to get all symmetrical properties around the compatibility. In the application this was done with the in-built OWL reasoner in Jena. This reasoning job was done after the HTTP post, which means that it affected the response time of all *-vso* runs. This was necessary to check because reasoning was one of the reasons why Renault chose to propose their API. In the experiments the reasoning had small impact on the actual results, but we have been working with small data sets from both Renault and Daimler. The computation of the reasoning will increase as more car models are added to the RDF graphs. The reasoning can be a hard computation. One solution can be to contain all the compatibility information in a separate RDF graph or let each product model have their own RDF graph with the information. This may let the reasoner return more quickly with the inferred triples. With reasoning there is the possibility of pre-computing the inferred triples.

This demands a bigger back-end for the third party user. If that is a too big task for the user, one can do the reasoning in real time. Here one can make a reasoner specific for the needs of the RDF graph. In our case, we only needed to infer the rule of symmetrical properties because *coo:incompatibleWith* is defined as *owl:SymmetricalProperty*. Another solution would be to use the Semantic Web Rule Language, SWRL[3] , to make some simple rules to apply to the RDF graphs. With SWRL we would only need a rule to say that *coo:incompatibleWith* is symmetrical. It can be seen below. This would make the operation of inferring triples less complex.

```
incompatibleWith(?x, ?y) -> incompatibleWith(?y, ?x)
```

It has some drawbacks which is that SWRL is not supported directly in Jena. This means that one would have to execute the SWRL rules with Pellet, an OWL 2 reasoner, either through Jena or through Protege. This also demands more than just basic knowledge about semantic technologies. Maybe the easiest approach to this problem is to divide the compatibility triples into separate RDF graphs before doing a general reasoning. Then afterwards add the inferred triples to the RDF graph about the product models to avoid inferring unnecessary information.

In both representations there are possible bottlenecks. Some of them have shown themselves to be more severe than others. In Renault's representation the bottleneck was the way of representing compatibility and in Daimler's representation it was the reasoning. Both cases may result in a loss of user experience, but this can be solved with more back-end functionality and smarter, more aimed solutions in the application.

6.2.3 Generic versus product specific

Creating ontologies to represent data about complex products are a new field of study. There are not many ontologies which can represent such data in a good way. The biggest challenge is to represent component constraints without losing precision. Renault presented a generic ontology which they argue that can be used on any complex product, for example cars or computers.

The ways of representing a specification is almost the same in both ontology approaches. The class *Specification* in CO can be seen as equal or as a sub class of either *QualitativeValue* or *QuantitativeValue* from GoodRelations. The VSO representation is similar, but demands more knowledge about the RDF graphs. This is because the two classes have different properties to indicate the actual value contained, respectively *gr:name* and *gr:hasValue*. Before acquiring the actual value one have to get the correct specification which in VSO can be done by following the correct property, like *vso:speed*. CO relies on instances of the class *ConfigurationVariable* to declare in a string literal what specification it contains. This is done to keep it totally generic. It is unfortunately not any way to keep it generic without using string literals as identifications. This

[3]http://www.w3.org/Submission/SWRL/

is not an ideal situation because it makes the data less machine readable which is a step away from one of the key concepts of semantic web. In a paper published in 2006, Tim Berners-Lee mention the importance of URIs because it allows machines to process the data directly. [40] In VSO there are not until the application retrieves the actual value that it deals directly with a literal.

One similarity between both ontologies are the way they represent special equipment. Here both ontologies have to use a string literal as identification. For instance if one wants to have an option on how many cup holders there are possible in this particular car model. In VSO/COO it is first wrapped in a *coo:SpecItemCollection* which then have URIs to each possibility, represented as a class *coo:ChoiceOrComponent*. This is almost the same as in CO where they use the same representation like all the other specifications. First there are a *co:ConfigurationVariable* which contains the ID of the component and URIs to all the possible specifications.

The generic ontology allows one to represent any data with some kinds of constraints, but as we have discussed it adds more workload on the actual representation. This is because in the ontology itself there are few constraints and guidelines on how to set up your representation. It means that if a user represent their data about car models and another user does the same, it is not certain that they represent the same things in the same way. With a product specific ontology one would often get a more controlled representation of the data which removes ambiguous information. The ranges and domains are more clearly specified. This can also be too rigid when describing complex products. Sometimes one might want to use another ontology to represent the actual values than the classes from GoodRelations. An example could be if a third party user needs to describe more than one type of product they would need an ontology for each type of product.

6.3 Pros and cons

There are several ways of making an application handle PDPs. As mentioned there can be differences in collecting data, presenting data and using data. This means that a recommended ontology and data representation needs to be robust. This means that it has to be able to perform well under different scenarios. The ideal semantic situation would not rely on string comparison, but with both ontologies one have to deal with string literals eventually. A string literal is still the value a user is presented with. It depends on how much of the data one wants machine accessible and how much is for internal human reading.

The ontologies in this thesis are not made for tackling every possible problem, but can perform well within their domain. The ontology structure of CO in conjunction of the data representation, is made for one particular scenario. That is to specify one specification after another, not all at once. Here CO has several tools at disposal to make it easier trimming down the result sets. The property *co:impossible* was to no use in our application

due that our application specified all specifications before searching. This property lets the application remove possibilities if a user have chosen some specifications. One would still has to use the lexicon as reference, but could eliminate further possibilities. The property *co:impliedSpec* does something similar and gives the user a list of specifications that are implied and do not need to be searched for. COO have some of the same functions and possibilities, but some of these has to be reasoned over to be useful. One can produce the same functionality by using *coo:incompatibleWith* with reasoning.

From the performance tests and programming, the product specific ontology came out as the best. This, as discussed, is not solely due to the ontology structure. In the generic ontology, many of the issues lies within the representation. The generic ontology can easily be used to represent different PDPs which will let the user stick to one ontology. Some alterations to the representation is needed to make it more robust than what Renault have done. They mention that reasoning is something a third party user should be shielded from. We disagree because if the user can choose when the reasoning is done they can avoid possible bottlenecks. This means that for the representation to work properly we recommend using a similar compatibility system to COO. Here one can add the triples to each RDF graph of the product model and do reasoning here. This gives the user freedom to choose whatever way of reasoning and storing they might find most applicable, which could be to pre-compute any big reasoning operations that may be unsuited for real time. This will remove all the extra queries which has to be done to move through Renault's large graph of configurations. This representation puts more workload on the third party user, but it will most likely decrease the response time drastically.

6.3.1 Alterations for the future

In addition to changing how the compatibility is done in CO, we would also do some other alterations. We would still keep the string IDs for every specification, but added a *co:hasValue* for value representation in each specification. It could be called *co:specValue*. Then it would not only be configuration variables which have a property like this to point to all the specifications it contains. We would also add type to all the literals. With RDF 1.1, W3C is trying to introduce a better literal standard than before. [26] Now every literal is at least typed as a string. If all values were typed, one could easily determine if one is working on a float or a string. Both these actions are done to make the representation more human and machine readable. To keep the string IDs for each specification one could use a lexicon of IDs or something similar to make it easier for third party users to understand the data representation. With these measures CO will become more similar to VSO/COO, but still maintain the possibility of being a generic ontology. There is also the possibility of using CO in conjunction with other ontologies. For instance that one can use VSO properties/classes to define specifications on cars and use the regular CO properties/classes to define computers.

CO should anyway be combined with GR. GR adds valuable functionality that CO could benefit from. For instance the property for stating unit of measurement, *gr:hasUnitOfMeasurement* and many more. This could make the specifications more informative than they are today.

In VSO/COO there are not much room for alterations, but one can add new properties and classes if it is necessary. As an example we added the property *daim:emission* to be able to represent emission rates of a car model, seen in Listing 4.3 on page 42. With VSO/COO one also get more functionality in the ontology which can move some of the functionality from the application to the representation. This will utilize semantic technologies more efficient and will be more in line with the semantic principles. With VSO/COO today one can use reasoning, SWRL rules or other technologies to decrease the workload of an application. An example would be that with COO a derivative can inherit features from a trim and a base model with some simple SWRL rules. This can decrease the amount of triples and take some workload of an application. With CO it is not that easy with the representation they have today. This is because the properties like *co:impossible*, *co:impliedSpec* and others are not meant to derive new information, just exclude or include new information in each configuration step. These properties are made as an alternative to reasoning to eliminate the need to infer new information. This is done to hide reasoning from the user. There would be possible to apply other semantic technologies to try to derive new useful information, but due to the large amount of configurations and configuration links that job would have been too huge to manage. They state in their paper that they got more than 10^{20} valid car configurations, which would be impossible to reason over because there is a configuration instance for every valid state.

Chapter 7

Conclusion

Research around complex products and semantic web are a young field of study. Today, there are not many vendors representing data like this, but the possibilities are endless. With more and more information being loaded to the internet, there is need for a way for both humans and machines to understand the information. The main challenge has been to represent component constraints about product models. Semantic web has been a proposed solution to that problem and is more widely used today[1].

In this thesis we have investigated the differences and possibilities between a generic ontology versus a product specific ontology to represent complex products.

In order to do this, we have used several ontologies and their ways of representing data. The research was based on two alternative structures and representations, CO and VSO in conjunction with COO. These ontologies used linked oped data from Renault and Daimler, and referred to car models and their specifications. Furthermore, our investigation have been based on experiments done against a prototype that represented a real life application using the data from both car manufacturers.

We recommend that both alternatives should be used for representing complex products. Using only CO there should be done some alterations to reduce bottleneck issues. We would not recommend using the data represented by Renault today, because it is not robust enough and has a too long response time for bigger operations. It is also important that the data representation is done properly to ensure readability for both humans and machines. Today, the Renault representation is neither very understandable for machine nor for human. This is reflected in the discussion about the application development. The results from the experiments done against the application shows that the current representation is not feasible with today's performance standards and user expectations. This is the area where most of the workload is on the use of the ontologies. This means that for a third party user VSO in conjunction with COO are the best solution to work with product models. However, this approach is limited to only represent data about car models. It would also require some reasoning to be effective, but with the technology and

[1]Google Knowledge Graph. The expanding of data available on dbpedia and freebase.

knowledge today that should not be a to difficult issue. For other users wanting to represent constraints between product models, CO can be a viable options if the proper alterations is done. It can be very useful when working with several data domains, but then the compatibility issue has to be solved. One solution could be to use the compatibility approach that COO offers. This would result in adding several new triples in each lexicon to avoid the configuration traversals. It would also add the need for reasoning to be effective, which is opposite of what Renault proposed. If the compatibility issue is solved we believe the performance can be feasible for almost every scenario. It will demand more from the third party user, but can utilize more than just data about car models. To make it even more effective, there should be done several other alterations to both the representation and the ontology as discussed in Section 6.3.1.

Our recommendation is based on giving the user freedom to query the data in several ways. A future goal would be to do a more thorough test of the ontologies for all possible scenarios, since the versatility of the ontologies are huge. We believe that ultimately the data from manufacturers and others can be profitable for several online vendors and third party sites. Other manufactures of complex product could profit from using semantic technologies to represent their data.

Based on the investigations done in this thesis we conclude:

- Both ontology approaches can be useful, but the generic needs several alterations to be more robust.
- The representation of the data is just as important as the ontology structure.
- Data should be as machine readable as possible, and properties denote more information, rather than classes if it is possible.
- To speed up the performance of the generic approach, the compatibility issue has to be resolved. One solution is to use the approach proposed in COO.
- Reasoning over the product models should be applied to increase efficiency and versatility.

Chapter 8

Further research

This thesis just touches some of the issues with complex products and semantic web. Our application, at this point, can only test the worst case scenario with minimal pre-computations and local storage. This was done to put most of the workload on each representation and the structure of each ontology.

There are several things which can be further researched around this field of study. Here are some of the possibilities we found during this thesis:

- Ontology structure and data representation
  - Research more ways of representing data. This includes what is the best way for both humans and machines to understand and use the data. This means looking into how to represent ID's, values and constraints to avoid ambiguity.
  - Research different ways of querying data. Researching which order one should query for a complex product. In our application there was room for making the performance better with a smarter querying. This could be done by identifying the specifications with a large number of values and constructing the queries with the goal of checking the fewest number of values.
- Third party user
  - Researching different ways of pre-computing and using local storage to improve performance for end-users. There is room for performance improvement with more pre-computations and storing some information locally. This puts more workload on the third party user. For further research one can take a look at lightweight solutions to help third party users. A guideline to use the linked open data.

Furthermore this field desperately needs more data. Today there is only one manufacturer which has opened their data for the public and that is Renault. Volkswagen had a project where they opened their data, but that was canceled and the data was removed in early 2013. To get the full potential out of applications using this data, everyone has to open up their

data and represent it in a way which is easy to acquire. It is needed that not only the car manufacturers, but also other companies which makes and sells complex products. We think there is a big possibility for third party users in cooperation with the manufacturers to make applications around these products which will make it easier for the costumers.

Bibliography

[1] URL: http://wiki.goodrelations-vocabulary.org/References (visited on 01/04/2013).

[2] URL: http://wiki.goodrelations-vocabulary.org/References (visited on 10/02/2014).

[3] URL: http://www.heppnetz.de/ontologies/goodrelations/v1.html#MasterCard (visited on 10/02/2014).

[4] URL: https://developers.google.com/events/io/sessions/351310959 (visited on 01/12/2013).

[5] URL: http://nodejs.org/ (visited on 20/03/2013).

[6] URL: http://benchmarksgame.alioth.debian.org/u32/javascript.php (visited on 15/02/2014).

[7] URL: http://jena.apache.org/ (visited on 15/03/2013).

[8] URL: http://maven.apache.org/ (visited on 12/02/2014).

[9] URL: http://spring.io/ (visited on 12/02/2014).

[10] H. Afsarmanesh and M. Shafahi. 'Specification and Configuration of Customized Complex Products'. In: (2013). URL: http://link.springer.com/chapter/10.1007/978-3-642-40543-3_9#page-1 (visited on 20/02/2014).

[11] *ARQ - A SPARQL Processor for Jena*. The Apache Software Foundation, 2011 - 2013. URL: http://jena.apache.org/documentation/query/index.html (visited on 15/03/2013).

[12] 2006. URL: http://www.w3.org/DesignIssues/LinkedData.html (visited on 10/03/2014).

[13] D. Brickley and R. V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. World Wide Web Consortium, 2004. URL: http://www.w3.org/TR/rdf-schema/ (visited on 02/04/2013).

[14] IKS Semantic CMS Community. *Semantic Lifting For Traditional Content Resources*. Lecture. 2012. URL: http://www.slideshare.net/IKS_Project.eu/lecture-semantic-liftingpresentation (visited on 19/03/2014).

[15] R. V. Guha D. Brickley and Layman A. *Resource Description Framework(RDF) Schemas, Working Draft*. World Wide Web Consortium, 1998. URL: http://www.w3.org/TR/1998/WD-rdf-schema-19980409/ (visited on 10/04/2013).

[16] Ernesto Barrios based on Daniel Meyer's code. *BsMD: Bayes Screening and Model Discrimination*. R package version 2013.0718. 2013. URL: http://CRAN.R-project.org/package=BsMD.

[17] W. Durant. *Story of Philosophy*. 1926.

[18] W. S. Means E. R. Harold. *XML in a nutshell*. 3rd ed. O'Reilly, 2004. Chap. 24, pp. 469, 496.

[19] R. Fielding et al. 'Hypertext Transfer Protocol – HTTP/1.1'. In: (1999). URL: http://tools.ietf.org/html/rfc2616 (visited on 01/02/2014).

[20] Edouard Chevalier Francois-Paul Servant. *Configuration Ontology: Modeling Product Customization as Linked Data*. Renault. URL: http://uk.co.rplug.renault.com/configurationontology (visited on 25/01/2014).

[21] Edouard Chevalier Francois-Paul Servant. 'Describing Customizable Products on the Web of Data'. In: (2013). URL: http://events.linkeddata.org/ldow2013/papers/ldow2013-paper-11.pdf.

[22] Edouard Chevalier Francois-Paul Servant. *Product Customization as Linked Data*. Springer, 2012, pp. 603–618. URL: http://link.springer.com/chapter/10.1007/978-3-642-30284-8_47.

[23] *Fuseki: serving RDF data over HTTP*. The Apache Software Foundation, 2011 - 2013. URL: http://jena.apache.org/documentation/query/index.html (visited on 15/03/2013).

[24] Ulrike Groemping. *DoE.base: Full factorials, orthogonal arrays and base utilities for DoE packages*. R package version 0.25-3. 2013. URL: http://CRAN.R-project.org/package=DoE.base.

[25] Ulrike Gromping. 'R Package FrF2 for Creating and Analyzing Fractional Factorial 2-Level Designs'. In: *Journal of Statistical Software* 56.1 (2014), pp. 1–56. URL: http://www.jstatsoft.org/v56/i01/.

[26] W3C Working Group. *What's New in RDF 1.1*. World Wide Web Consortium, 2014. URL: http://www.w3.org/TR/rdf11-new/ (visited on 19/03/2014).

[27] Tom Heath, Christian Bizer and Tim Berners-Lee. 'Linked Data - The Story So Far'. In: (). URL: http://semanticweb.com/volkswagen-das-auto-company-is-das-semantic-web-company_b23233.

[28] Martin Hepp. *Car Options Ontology*. Hepp Research and Volkswagen, 2010. URL: http://www.volkswagen.co.uk/vocabularies/coo/ns.html (visited on 10/02/2014).

[29] Martin Hepp. *GoodRelations Language Reference*. E-Business and Web Science Research Group, 2011. URL: http://www.heppnetz.de/ontologies/goodrelations/v1.html#classes (visited on 25/03/2014).

[30] Martin Hepp. 'Vehicle Sales Ontology'. In: (2010). URL: http://www.heppnetz.de/ontologies/vso/ns (visited on 10/02/2014).

[31] Martin Hepp. 'Vehicle Sales Ontology - Classes'. In: (2010). URL: http://www.heppnetz.de/ontologies/vso/ns#classes (visited on 10/02/2014).

[32] *Jena Ontology API*. The Apache Software Foundation, 2011 - 2013. URL: http://jena.apache.org/documentation/query/index.html (visited on 15/03/2013).

[33] Kjetil Kjernsmo and John S. Tyssedal. 'Introducing Statistical Design of Experiments to SPARQL Endpoint Evaluation'. In: (2013). URL: http://folk.uio.no/kjekje/2013/iswc.pdf (visited on 28/02/2014).

[34] O. Lassila and R. R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification*. World Wide Web Consortium, 1999. URL: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ (visited on 02/04/2013).

[35] M. Véron M. Aldanondo and H. Fargier. 'Configuration in manufacturing industry requirements, problems and definitions'. In: (1999). URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=816691 (visited on 20/02/2014).

[36] Fiona Fui-Hoon Nah. 'A study on tolerable waiting time: How long are Web users willing to wait?' In: (2004). URL: http://cba.unl.edu/research/articles/548/download.pdf (visited on 25/03/2014).

[37] Natalya F. Noy and Deborah L. McGuinness. 'Ontology Development 101: A Guide to Creating Your First Ontology'. In: (2000). URL: http://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html (visited on 13/03/2014).

[38] *Quantity units*. IBM, 2014. URL: http://pic.dhe.ibm.com/infocenter/wchelp/v7r0m0/index.jsp?topic=/com.ibm.commerce.developer.doc/concepts/cosquant.htm (visited on 31/03/2014).

[39] *Reasoners and rule engines: Jena inference support*. The Apache Software Foundation, 2011 - 2013. URL: http://jena.apache.org/documentation/query/index.html (visited on 15/03/2013).

[40] Nigel Shadbolt, Wendy Hall and Tim Berners-Lee. 'The Semantic Web Revisited'. In: (2006). URL: http://eprints.soton.ac.uk/262614/1/Semantic_Web_Revisted.pdf (visited on 05/03/2014).

[41] J. Hendler T. Berners-Lee and O. Lassila. 'The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities'. In: (2001). URL: http://www.nature.com/doifinder/10.1038/scientificamerican0501-34 (visited on 14/11/2014).

[42] *TDB*. The Apache Software Foundation, 2011 - 2013. URL: http://jena.apache.org/documentation/tdb/index.html (visited on 15/03/2013).

[43] *The core RDF API*. The Apache Software Foundation, 2011 - 2013. URL: http://jena.apache.org/documentation/rdf/index.html (visited on 15/03/2013).

[44] *The ElementTree XML API*. Python Software Foundation, 2013. URL: http://docs.python.org/2/library/xml.etree.elementtree.html (visited on 04/10/2013).

[45] C. F. Jeff Wu and M. S. Hamada. *Experiments: Planning, Analysis, and Optimization*. 2nd ed. John Wiley and Sons, Inc., 2009. Chap. 1, pp. 4, 8.

[46] C. F. Jeff Wu and M. S. Hamada. *Experiments: Planning, Analysis, and Optimization*. 2nd ed. John Wiley and Sons, Inc., 2009. Chap. 4.1 - 4.3, pp. 155, 169.

[47] C. F. Jeff Wu and M. S. Hamada. *Experiments: Planning, Analysis, and Optimization*. 2nd ed. John Wiley and Sons, Inc., 2009. Chap. 4.8, pp. 177, 180.

[48] Jennifer Zaino. 'Volkswagen: Das Auto Company is Das Semantic Web Company!' In: (2011). URL: http://semanticweb.com/volkswagen-das-auto-company-is-das-semantic-web-company_b23233 (visited on 11/03/2014).

Appendices

Appendix A

Daimler RDF

The RDF graph of the Daimler data are found in the URLs beneath. The first two are about the A-Class cars and B-Class cars while the last one is the compatibility triples for both RDF graphs.

- http://folk.uio.no/magnudae/LinkedOpenData/daimlerVsoFictive.ttl
- http://folk.uio.no/magnudae/LinkedOpenData/daimlerVsoFictive2.ttl
- http://folk.uio.no/magnudae/LinkedOpenData/compatibilityTriples.ttl

They were too big to be displayed directly in the thesis.

Appendix B

Planning matrix for the large experiment

The planning matrix for the thirteen factor experiment is found here http://folk.uio.no/magnudae/LinkedOpenData/largeTable.pdf. The table was way to large to contain in the thesis so that is why I have placed it in a seperate PDF on my homepage.

Appendix C

Results from the experiments

The results are found here http://folk.uio.no/magnudae/Results/ in csv format. All files include header which indicates the factors used in the experiment.

Appendix D

Code base

All the code can be found on bitbucket here https://bitbucket.org/magnudae/thesisprototype/src.