UNIVERSITY OF OSLO
Department of Informatics

Checking Quality Criteria for ISO 15926-8 compliant Installation Descriptions

Master thesis

Håvard Mikkelsen Ottestad

May 2, 2013

Checking Quality Criteria for ISO 15926-8 compliant Installation Descriptions

Håvard Mikkelsen Ottestad

2nd May 2013

Abstract

An implementation of ISO 15926-8 compliant Installation Descriptions must comply with a set of quality criteria to verify typing, literals, subclasses, part 8 adherence and be a conservative extension of ISO 15926-8. The first criteria can be checked with a combination of SPARQL and DL queries together with an Integrity Constraints Validator. A typical implementation was found to be a model conservative extension of ISO 15926-8, and limiting the allowed axioms is sufficient to ensure a model conservative extension. An extended model conservative extension with stricter requirements was found to limit concept unsatisfiability.

Martin Giese is the supervisor for this thesis.

Contents

Part I

Introduction

Chapter 1

Thesis

> Is it possible to computationally discover if an implementation according to ISO 15926 part 8 is of high quality?

This thesis is based on a number of criteria by Martin Giese that any implementation must satisfy. I attempt to automate the checking of these criteria and explore if and how they can be improved.

The Integrated Operations in the High North project[1] was a basis for this thesis and the example ontologies where sourced from the project and provided by Stiftelsen Det Norske Veritas[2].

Chapter 2

Fundamentals

I base this thesis heavily on Semantic Technologies. This includes the Web Ontology Language[3] (OWL), the Resource Description Framework[4] (RDF), Description Logics[5] (DL) and the SPARQL Protocol and RDF Query Language[6] (SPARQL).

2.1 RDF

RDF is a data format with a number of serialisations. The data format is based on triples, which means to say that you would specify a subject, then link the subject to an object with property.

```
Subject Property Object

Example:
dbpedia:Peter_Pan :livesIn :Neverland.
```

There are many ways of thinking about this. One way is to look at object orientation where an instance of a class becomes the subject and the property is a variable name and the object is the value for the variable.

Another way of thinking about it is as a table. Where the property becomes the name of the table, and the subject and object become two columns. It is even possible to denormalise it further and just put the property as a column as well.

Since a triple can be used to denote a directed graph, then RDF can be considered a notation for graphs. This corresponds well with the object oriented view where an object can point to other objects creating a directed graph.

To digress it is worth to mention that there do in fact exist object oriented data stores, notably object oriented databases[7] that stand as a counter part to the very common relational databases such as SQL databases. And as of late JSON databases have become common, JSON being the notation for javascript objects which is heavily used on the web. One such database is RethinkDB[8].

RDF also comes with a vocabulary set, which most notably includes support for typing (rdf:type). A simple example is:

```
dbpedia:Peter_Pan rdf:type foaf:Person
```

A resource in RDF is represented by an IRI (Internationalized resource identifier, a superset of URI), so Peter Pan is realised as *http://dbpedia.org/resource/Peter_Pan*. It is easy to see that this becomes an easy way of having unique identifiers as long as you stick to naming things in your own domain.

RDF also supports literals to represent a value that is not a resource, such as a name or an integer. A literal can also have a type, and these are commonly chosen from the XML Schema Definition (XSD).

Finally there is support for anonymous nodes by blank nodes. This can be used to formulate an unknown or to group properties around a node that does not need to be named or should not be named. For easy reference to a blank node it can be given a syntax name or a Skolem IRI.

2.2 RDFS

The RDF Schema is built on top of RDF to allow for a richer data representation. For this thesis, the most important part of RDFS is the subclass property (rdfs:subClassOf) which makes it possible to model inheritance in a data model. Together with subclassing, RDFS also defines what a class is (rdfs:Class), fundamentals such as rdfs:Resource (even though a property is defined in RDF) and domain and range attributes for properties.

2.3 OWL

For a richer language there is OWL, the Web Ontology Language. OWL comes in many flavours, Lite, DL, Full and a number of profiles. All these flavours have different functionality with different computational properties.

Qualified existential restrictions is important functionality in OWL and is used in this thesis. With a qualified existential restriction it is essentially possible to specify variables of a class. By which I mean that it it possible to require that an instance of the class have a specific property to a certain type of individual or literal. Universal restrictions are also important since these can be used to specify that a class instance can only have an object with a limited type for a certain property. As well as cardinality restrictions to specify a certain number of objects or a minimum or maximum.

OWL also has intersection and union. So a class can be a subclass of an intersection of two classes, or the intersection of a class can be the subclass of owl:Nothing (i.e. bottom) to cause them to be disjoint. Intersection and union can also be used as a part of a restriction or even in a general axiom.

There are also many features for properties, such as functional, transitive, symmetric and reflexive among others. These are however not used in this thesis.

Having an ontology may be nice in its own way, however the true power comes when combining an ontology with a dataset and running it through a reasoner. A reasoner is a computer program that understands OWL and can apply the axioms in OWL to the ontology and to the data. An example of this is if Peter Pan is a boy and all boys are human, then Peter Pan is a human. Reasoners can typically also do more advanced things than simple classification. For instance, if Peter Pan is a human and all humans are animals and only animals with wings can fly, then Peter Pan must have wings.

2.4 Description Logic and Manchester syntax.

Since OWL implements DL (Description Logics) features, DL is a natural way of describing axioms in OWL. Manchester syntax is also a common notation for OWL as well as many other serialisations such as RDF/XML and Turtle, these being very detailed and verbose primarily used in machine to machine applications. Manchester syntax is successfully used for modelling in Protégé[9] and is a very human readable OWL serialisation. DL is the mathematical representation of OWL, being very compact and powerful.

For the convenience of the reader table 2.1 is a simple comparison between DL and Manchester syntax.

DL	Manchester syntax	Description
$A \sqsubseteq B$	A *subClassOf* B	Subclassing
$\exists prop.B$	prop *some* B	Existential restriction
$= 1\, prop.B$	prop *exactly 1* B	Cardinality restriction
$\forall prop.B$	prop *only* B	For all restriction
$A \sqcap B \sqcup C$	A *and* B *or* C	Conjunction and disjunction
$A \sqcap B \models \bot$	A *disjointWith* B	Disjoint classes

Table 2.1: DL and Manchester Syntax

2.5 SPARQL

SPARQL is a query language for the semantic web that uses a very similar syntax to RDF. It is a graph query language, as such a user can specify a graph with variables in place of constants for nodes.

Considering a hypothetical data model for Peter Pan where he has friends and those friends have a certain age, then a query to find out how old his friends are could look something like this:

```
SELECT * WHERE{

    dbpedia:Peter_Pan foaf:knows ?friend.
    ?friend foaf:age ?age.
```

```
}
```

Or with a blank node:

```
SELECT * WHERE{

    dbpedia:Peter_Pan foaf:knows [ foaf:age ?age] .

}
```

Joins, which have to be stated in SQL are implicit in SPARQL. There is no need to specify that the ?friend in the first line is the same as the ?friend in the second line.

SPARQL also supports aggregates and order by similarly to SQL, and SPARQL 1.1 also supports inserts and deletes and more including simple negation notation which I have used in this thesis.

2.6 DL-Query

DL-queries as somewhat different from SPARQL. Instead of asking for any variable with a type P (?a rdf:type :P), you would simple query for members of the class P by simple stating the class. All DL syntax is allowed, so you can query for the members in the intersection of two classes, or every individual that is in the abstract concept defined by a restriction, such as "foaf:knows some Person" would return a set of individuals including Peter Pan.

With DL-queries it is also possible to query for things that are not explicitly stated (unlike SPARQL). For instance if all boys are between 1 and 18 years of age, then a query for anyone younger than 25 will return Peter Pan even if there is no explicit mention of his age just as long as he is a boy.

Part II

ISO 15926

Chapter 3

A brief introduction to the standard

ISO 15926 is large standard with eleven parts (as of 2009)[10] used for data integration and sharing between systems. The standard handles modelling of a system, such as oil and gas operations, integrating life-cycle data and sharing this data. An example of how ISO 15926 is used for sharing data is the Daily Drilling Report[11] sent to the Norwegian Petroleum Directorate by all "operating companies drilling wells on the Norwegian Continental Shelf"[11, derived from].

The eleven parts of ISO 15926 are roughly summarised as:

- Part 1: Introductory part.
- Part 2: Data model with time and space.
- Part 3: Geometry and topology reference data.
- Part 4, 5, 6: Combined reference data terms.
- Part 7: Integration of life-cycle data with templates.
- Part 8: Implementation of part 7 in OWL.
- Part 9: Implementation standards including Façades.
- Part 10: Test methods.
- Part 11: Industrial Usage Guidelines

(Heavily derived from [10])

ISO 15926 aims to take a step further when it comes to data integration and sharing by one upping what HTML has done for the Web or what JPEG did for imaging.

Chapter 4

Templates and OWL

Templates[12] is a contstruction for generating instance data. It does this by providing a function that maps its inputs into properties and instances and adds these to an instance of the template class.

A good analogy is constructors in Java. A common way of defining a new instance of a class is to use the "new" keyword in conjunction to a call to the class constructor with suitable arguments. The constructor then adds the arguments to the proper variables in the object and returns the class instance (object).

Similairly a template takes a set of arguments, instansiates the class and adds the arguments. An example of a template is one for a choke change event: ChokeChangeEvent(AdjustableChokeValve hasChoke, String valChoke-Type, DateTime valDateTime).

In OWL this is represented as a class *ChokeChangeEvent* with four "SubClass Of" statements:

```
Class: iohn6tpl:ChokeChangeEvent

    SubClassOf:
        (iohn6tpl:valChokeType only xsd:string)
         and (iohn6tpl:valChokeType exactly 1
            xsd:string),
        (iohn6tpl:hasChoke only
            iohn6:AdjustableChokeValve)
         and (iohn6tpl:hasChoke exactly 1
            iohn6:AdjustableChokeValve),
        (iohn6tpl:valDateTime only xsd:dateTime)
         and (iohn6tpl:valDateTime exactly 1
            xsd:dateTime),
        p7tm:RDLTemplateStatement
```

This template definition can be used by a computer system for inputting data into the semantic model, or by an engineer. It is used to represent when the choke valve or an internal part thereof has been replaced.

Other templates may be used for modelling the system, such as the *WellForPlatform*:

```
Class: iohn6tpl:WellForPlatform

    SubClassOf:
        (iohn6tpl:hasOilAndGasPlatform only
            iohn6:OilAndGasPlatform)
         and (iohn6tpl:hasOilAndGasPlatform exactly 1
            iohn6:OilAndGasPlatform),
        (iohn6tpl:hasWell only iohn6:Well)
         and (iohn6tpl:hasWell exactly 1 iohn6:Well),
        p7tm:RDLTemplateStatement
```

And some templates are explicitly used for lifecycle data generated by sensors. An example of this is for a sand event which is captured by a sensor.

```
Class: iohn6tpl:SandEvent

    SubClassOf:
        (iohn6tpl:hasDataSource only iohn6:DataSource)
         and (iohn6tpl:hasDataSource exactly 1
            iohn6:DataSource),
        (iohn6tpl:valDateTime only xsd:dateTime)
         and (iohn6tpl:valDateTime exactly 1
            xsd:dateTime),
        p7tm:RDLTemplateStatement,
        (iohn6tpl:valSandEventId only xsd:integer)
         and (iohn6tpl:valSandEventId exactly 1
            xsd:integer)
```

From the appendix Martin Giese makes a good definiton of the templating language and how it maps to RDF which I will cite here.

- All data is originally represented as a set of "template instances" and "type assertions"
  - A template instance is an n-ary literal of the shape
    $$p(i_1, \ldots, i_n)$$
    where $i_1, \ldots, i_n$ are literals (strings, numbers, etc) or resources identifiers (URIs), and p is a template (also identified by a URI)
  - A type assertion is a literal
    $$C(i)$$
    where C is an OWL class and i is a resource identifier as before
- For every template p, there is a description giving an RDF property $R_{p,i}$ for each of the arguments of p.

- Any template instance $p(i_1, \ldots, i_n)$ is represented in RDF as a set of $n+1$ triples

  _:x rdf:type p;
  _:x $R_{p,1}$ i_1;
  ...
  _:x $R_{p,n}$ i_n.

- Any type assertion $C(i)$ is represented in RDF by a triple

  i rdf:type C

End citation.

Chapter 5

Example ontology

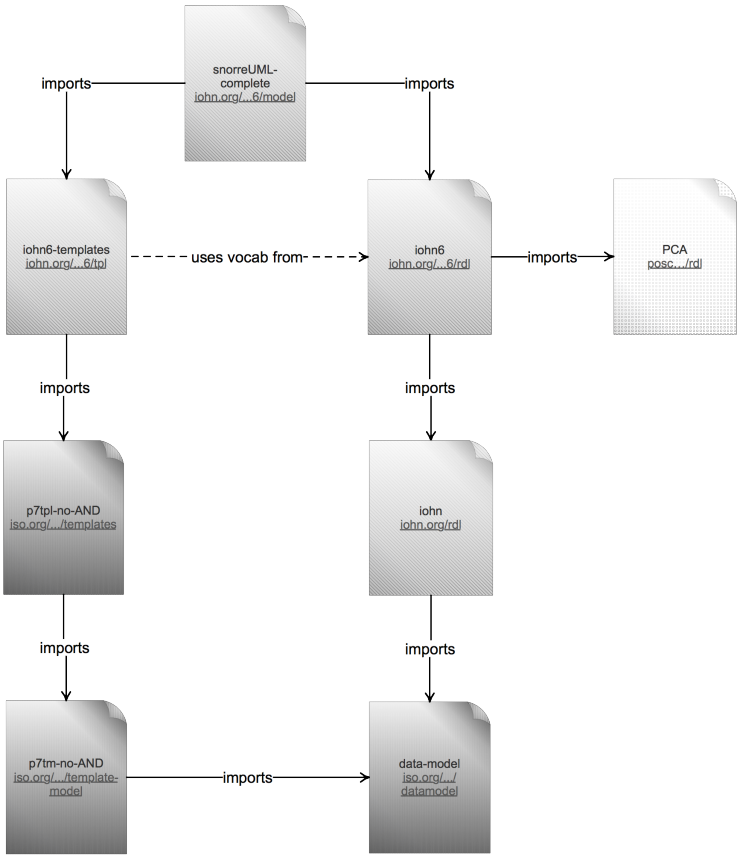

Figure 5.1: Ontology file overview (shading represent different namespace owners)

For this thesis I was provided with an example model to work with consisting of several ontology files. The top ontology file, as shown in figure 5.1 on the preceding page, named snorreUML-complete.owl is the all inclusive file that ties together the entire model. This ontology was constructed from a MS Visio diagram drawing of the constructions at the Snorre oil field. These constructions include platforms, flowlines, pipes, chokes, pressure elements and temperature elements among many, many others.

The snorreUML-complete.owl ontology uses the template definitions in iohn6-templates together with the class listings in iohn6 to model the constructions using template statements. Essentially individuals of type template, such as *Link.19.103* which is of type *ChokeOfWell* defined as a subclass of *RDLTemplateStatement* requiring a *hasWell* and a *hasChoke* of types *Well* and *AdjustableChokeValve* (respectively).

Iohn6.owl contains subclassed classes from PCA.owl, such as *Well*, which is a subclass of *DataSource* and *ContinousFluidTransportationDevice* (iohn6.owl defined) as well as *RDS16458543* defined in PCA.owl with a label "Well". The classes in iohn6.owl are used in the template definitions in iohn6-templates.owl, they become the bridging vocabulary between the templates and the external definitions.

PCA.owl is the reference data library (RDL) from POSC Caesar Association. The RDL is only a subset of the complete RDL by POSC Caesar Association. PCA.owl uses unique id's as resource identifiers and labels as descriptors. The top class of PCA.owl is "ISO 15926-4 THING" with two possible subclasses "ISO 15926-4 ABSTRACT OBJECT" and "ISO 15926-4 POSSIBLE INDIVIDUAL". For an overview of the PCA.owl reference data library view the screenshot from Protégé in figure 5.2 on the next page showing a selection of the classes in the ontology.

There are three other ontologies from ISO. These are p7tpl-no-AND.owl, p7tm-no-AND.owl and data-model.owl. The two templates with the AND suffix have been adapted form their original ontologies to better fit with the ontology managments system from Cambridge Semantics[13] chosen in the Integrated Operations in the High North project.

The base ontology, data-model.owl, with an extract shown in figure 5.3 on page 20, consists of a lot of base classes used as building blocks for further expansions. I did not directly use the ontology, however I would point out the somewhat unlucky choice of name for *Thing* since it is already so widely associated with *owl:Thing*.

On top of data-model.owl comes the first template definitions in p7tm-no-AND.owl, extract shown in figure 5.4 on page 21. For the most it is a set of classes, with the exception of the *MetaTemplateStatement* subclasses which have restrictions on the form of template defenitions.

```
Class: p7tm:TemplateDescription

    SubClassOf:
        p7tm:hasTemplate
```

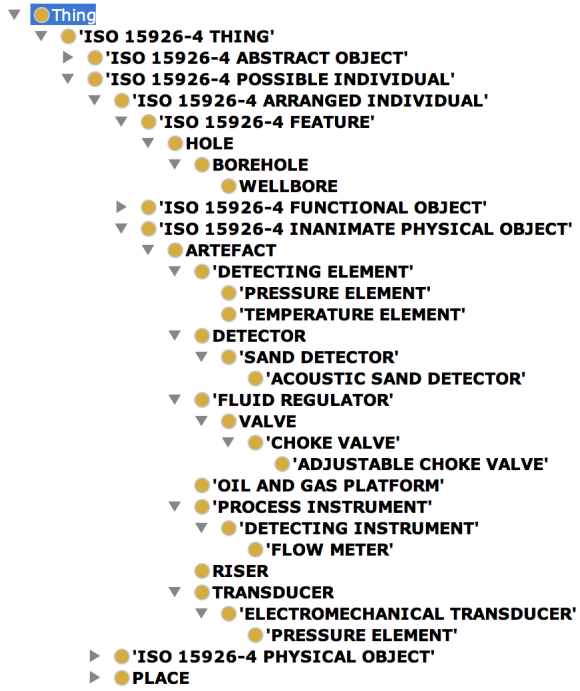

Figure 5.2: PCA.owl

```
                only p7tm:Template,
p7tm:hasTemplate
                exactly 1 p7tm:Template,
p7tm:valNumberOfRoles
                only xsd:positiveInteger,
p7tm:valNumberOfRoles
                exactly 1 xsd:positiveInteger,
p7tm:MetaTemplateStatement
```

The ontology also defines a few object properties which are used in these restrictions. The top-property is *hasObjectRoleFiller* which has two subproperties *hasBaseTemplateObjectRoleFiller* and *hasMetaTemplateObjectRoleFiller*, where the former has no sub-properties. There are also five data properties with the same setup as for the object properties with a top-property *valDataRoleFiller* and two main sub-properties one for *Base* and one for *Meta*. The two data properties in the above example (val as prefix) are sub-properties of *valMetaTemplateDataRoleFiller*.

Above p7tm-no-AND.owl sits p7tpl-no-AND.owl with a strikingly similar name. Where p7tm defines core template concepts, p7tpl actually specifies some base template statements, or definitions if you prefer. It starts by defining the

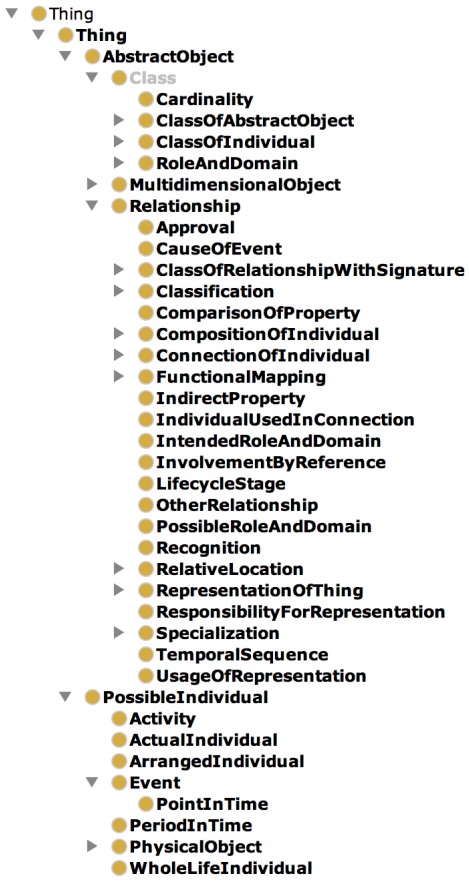

Figure 5.3: data-model.owl

ProtoTemplate with members being individuals and classes (simultaneously), a technique known as punning[14].

P7tpl goes on to define a series of templates within the *InitialSetTemplateStatement* and within the *ProtoTemplateStatement* classes. Examples of these statements are respectively *SymbolOfScale* which has a scale and a symbol; and *CoordinateSystem* which has a domain and a codomain being property spaces and number spaces respectively.

Finally the last unmentioned ontology, iohn.owl, is an ontology containing four classes with several memebers to define datatypes, units of measurements and aggregation types. The contained individuals are for instance *Hour* and *CubicMetrePerHour* within *UnitOfMeasure*; *SandRate* and *Stroke* for *DataType*; *InsideDiameter* and *OutsideDiameter* for *DiameterType;* and *AccumulatedValue* and *WeeklyAveragedValue* for *AggregationType* among others.

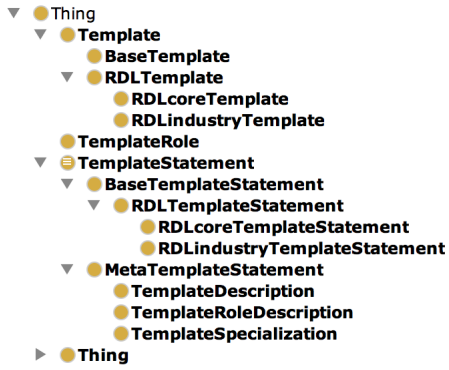

Figure 5.4: p7tm-no-AND.owl

Chapter 6

Criteria

Martin Giese has written the paper Quality Criteria for RDF representations of installations descriptions according to ISO15926 part 8 which is the basis for this thesis. The paper outlines numerous requirements that an ISO 15926-8 ontology must fulfil in order to be considered a good ontology. Some of the requirements are generic, such as 2.4.1 which requires the Abox and Tbox to be consistent, and some of the requirements are more specific such as the requirement for conservative extensions[15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] (3.1.1).

The entire document is available in the appendix. Examples for many of the requirements have been added my me to make the document more comprehensible. The document is divided into three main sections and a forth section on further thoughts.

The first section merely outlines prerequisites related to the templating language and the representation thereof in RDF. Following the prerequisites is the section on "Requirements for RDF representations in general". This is the main section on the generic requirements containing the following topics:

- Separation of Information Levels (2.1)
- Literals (2.2)
- Types (2.3)
- Consistency (2.4)

The first topic is a requirement for the later topics. If there is no good separation of information levels then checking for types becomes much harder. In essence the information should be divided into two main parts, vocabulary and instance data. This corresponds nicely to the Abox and Tbox divisions. Further more the vocabulary should be divided into an application specific vocabulary and a generic vocabulary.

Literals are first class citizens in RDF, and can be typed, untyped and with or without a language tag. The requirement simply outlines that all literals

should be typed, and any literal that is human readable should have a language tag.

For types, there are several levels of requirements depending on how explicit that typing should be. Every individual in RDF can have a type, this type can be stated or inferred. The requirements give several logical levels for typing, from any inferred type, to that all inferable should be stated.

Consistency is an obvious requirement, it is again divided into two separate levels. One level simply requires full consistency, while the other requires a consistent Tbox and any individuals to not be members of disjoint classes (stated or inferred by RDFS). The latter is the more interesting, since it sacrifices soundness for tractability.

The second to last section of the document is on ISO 15926 part 8 specific requirements. There are requirements for part 8 adherence, and a requirement for conservative extensions. Conservative extensions is quite interesting, which is why a large part of this thesis is dedicated to the topic.

Part III

Simple requirements

Chapter 7

Requirement for literals

The requirement for literals is put in place to limit ambiguity by forcing all literals to either be a typed literal or a string literal with a language. By forcing all untyped literals to have a language it is much easier to check for untyped literals that should be typed. Common typed literals are those for integers, floats and dates, the typing of literals are defined by the XML Schema Definition[27]. The definition includes 19 primitive types, all with precision requirements which can be checked by reasoners.

7.1 Verifying with SPARQL

There are several ways to verify literal types of an RDF document. One could iterate over all literals with Jena and check every one, or attempt to write a SWRL rule or even use simple regex to process an RDF file (preferably in turtle).

My approach is to write a SPARQL query to extract all literals that do not adhere to the requirement.

```
SELECT ?a ?b ?c WHERE {
    ?a ?b ?c.
    FILTER(    isLiteral(?c)
            && lang(?c) = ""
            && datatype(?c) = xsd:string
    )
}
```

This SPARQL query works by only keeping literals that have no language and are strings. Fortunately all literals are a minimum of xsd:string, so even untyped literals are picked up by the query.

Unfortunately though, the "datatype" function in SPARQL returns xsd:string for untyped literals, which makes checking literals that are non-human readable and intentionally lacking a language tag more challenging. With RDF 1.0 a literal is either plain or typed, where a plain literal is a string with an optional

language tag[4]. The definition did not specify if the string meant an xsd:string or not, so in RDF 1.1 this was clarified and as of 1.1 all literals are xsd:string by default, unless stated otherwise[28]. Such literals are denoted as simple literals and the lack of xsd:string in syntax form is only available to simplify notation, rather than as a semantic difference.

RDF 1.1 also brings along a new datatype IRI for use with human readable literals. This literal is rdf:langString (http://www.w3.org/1999/02/22-rdf-syntax-ns#langString). The datatype is defined as an if and only if case, so any xsd:string with a language tag must also be a rdf:langString. Rdf:langString is however not implemented, and can not be found in the rdf vocabulary file online[29] or in the list of available literal datatypes in Protégé (version 4.2 early 2013).

There is still a plain literal datatype (rdf:plainLiteral) in the rdf vocabulary. The comments for the definition say it is "the class of plain (i.e. untyped) literal values"[29, Ivan Herman 2010], so as an intermediary solution this datatype would forego the SPARQL query and can be used on non-human readable strings.

Chapter 8

Requirements for types

The requirements 2.3.x are all related to typing of individuals. By default all individuals that are untyped are typed as owl:Thing, however the requirements require more explicit typing.

The main problem with type checking is to only check types of individuals in the Abox and only allowing a selection of types. This selection must be specified to the program in some manner. By the requirement for separation of information levels it is possible to extract the allowed types from the vocabulary ontology.

Since the separation of information levels also separates out the instance data, this model could be loaded into Jena[30] and then iterated over to check that there was an explicit type for every individual. Another option is to write a SPARQL query:

```
SELECT distinct ?a WHERE {
    ?a ?b ?c.
    FILTER(!EXISTS{?a a ?d})
}
```

The query returns all subjects that do not have a type. For objects it is possible to check on ?c instead. The biggest problem with this query is the use of negation, which is very slow. Potentially the query could take $O(n^2)$, depending on how Jena manages to optimise the query.

Another more efficient approach would be to do type checking at parse time. In pseudo code, a reasonable solution:

```
Hashmap untyped
Hashmap typed
Hashmap allowedTypes

for every trippel x,y,z
    if(y = rdf:type && allowedTypes.contains(z))
        untyped.remove(x)
        typed.add(x)
```

```
    else if(!typed.contains(x))
        untyped.add(x)
```

The code does have an Achilles heel, which is that it would not check if z has a type. By normalising the RDF and adding a "z a owl:Thing" for every z in x,y,z, this would easily be solved.

8.1 Allowed types

When using a SPARQL query to check the requirement there are two possible solutions to limiting the allowed types. The first solution is to generate a SPARQL query from the list of allowed types within an exists query. Something like the following:

```
SELECT distinct ?a WHERE {
    ?a ?b ?c.
    FILTER(!EXISTS{    ?a a :AllowedType1
                    || ?a a :AllowedType2
                    || ?a a :AllowedType3
                   })
}
```

Simply adding more "|| ?a a :x" for x an element of the set of allowed types.

Another approach is to trace the allowed types from the vocabulary ontology through a joint model of the instance data and the ontologies. Jena does not have any explicit support for tracing other than allowing a model to be reasoned by an ontology without merging, though this did not work with Pellet[31]. The most feasible approach is to add a tracing element onto the classes for the allowed types, something in the nature of "rdfs:label "uniqueTracingLabel"". Then the SPARQL query would check that the type had such a label.

```
SELECT distinct ?a WHERE {
    ?a ?b ?c.
    FILTER(!EXISTS{?a a [rdfs:label
                         "uniqueTracingLabel"]})
}
```

8.2 Reasoning

The requirements for typing include two requirements with reasoning. One allows for full OWL reasoning (2.3.1) and another limits reasoning to RDFS (2.3.2). There could also we level in between for different classes of OWL.

Jena has support for RDFS reasoning through *ReasonerRegistry.getRDFSReasoner()*, and full OWL reasoning can be provided by Pellet (limited to SROIQ(D)[32]). However, dividing the reasoner and data such that the reasoner can reason on the

data and give every entailed statement from the data by the ontology does not work. There are two ways this should work. One is by defining a schema and model when creating an InfModel: *createInfModel(Reasoner reasoner, Model schema, Model model)*, however Jena simply merges the schema and model by *schema.add(model)* before applying the reasoner. The second is to attach a schema to the reasoner by *reasoner.bindSchema(schema)*, which also does not yield a reasoned dataset without the ontology.

The solution is again to trace the ontology though the reasoner by annotation every subject with rdfs:label "IGNORE" and subtracting these subject in the SPARQL query. If this is not done, then the SPARQL query will return every subject in the ontology, since none of these are of an approved type.

```
SELECT distinct ?a WHERE {
    ?a ?b ?c.
    MINUS{?a rdfs:label "IGNORE"}
    FILTER(!EXISTS{?a a [rdfs:label
                         "uniqueTracingLabel"]})
}
```

8.3 Requirement 2.3.4 requiring complete explicit typing

As stated in the rationale for the requirement, this requirement makes working with RDF data easier by eliminating the need for reasoning when looking for individuals of a certain type. It is also great for working with negation.

An approach to testing this requirement would require full reasoning on a dataset to list all individuals and all reasoned and stated types for every individual. Then checking the dataset in its unreasoned form to see if every individual has every reasoned type stated explicitly. This could be done at parse time, by filling buckets of all types with the individuals that have those reasoned types and then subtracting the individuals from their buckets for every triple with that individual and a stated type.

Another approach would be to generate a SPARQL query. This can be done as conjunctive query.

```
ASK {
    :someIndividual a :Type1, :Type2, :Type3.
}
```

However this approach does not work for anonymous blank nodes. An anonymous blank node is denoted by square brackets in turtle ([]) and does not have a stable name. Jena names all anonymous blank nodes, but the same RDF file will have different identifiers for the blank nodes when parsed into two separate Jena models. If the blank node has a type that specifies a key, then this key can be used to query for the individual. However it adds a fair amount of complexity to the query. A simpler approach is to name all blank nodes with a

UUID at parse time, the non unique name assumption takes care of any blank nodes that are the same but are named differently.

8.4 Eyeball

"Apache Jena provides a collection of tools and Java libraries to help you to develop semantic web and linked-data apps, tools and servers."[30] Among those tools is one called Eyeball[33].

Eyeball is a self-declared RDF linting tool designed to check the quality of a RDF document. I considered the tool as a building block for this thesis, since it allows for user defined SPARQL based constraints. However, since my use-cases required generating queries on the fly, as well as tracing input data I found it easier to simply start from scratch.

With Eyeball however a few of the requirements stated here can be tested to a certain extent. The requirement for literals is easily solvable with Eyeball. As is checking for untyped resources. However, the requirement for resource typing also specifies a vocabulary and a limitation to the typing. How Eyeball will manage this is not tested. There is a mention of a vocabulary inspector in the documentation[33] so there may be some support for this.

There are also a lot of other checks available with Eyeball. Checking for broken RDF list structures may not be useful if these are not used, however checking for ill-formed language tags may be useful where these have been handwritten or checking for unknown classes and properties. Proper tooling (like Protégé) will however severely limit these issues.

Consistent typing can also be checked with Eyeball. The definition for consistent typing is rather vague and either states that for any given class, an individual should only belong to a single subclass, or that all classes are disjoint with all other classes that are not super- or subclasses. It could also mean both if all types are derived and Thing is used as a superclass. This would be quite useful for such things as pipes, where a pipe should only have one diameter and length and thus belong to only one pipe-class. However, if such cases are few and far between, then making classes disjoint is much more precise.

Chapter 9

Subclass requirement

OWL is very good at representing hierarchical data. Classes and subclassing is an integral part of OWL and is used to show, among other things, that a *Flowline_5inch_Sch160* is a *ScheduleFlowline* which is a *Flowline* which is a *ContinousFluidTransportDevice* which is an *Artefact*.

The open world assumption[34, page 372] lets an individual be a member of a class without being a member of any of it's subclasses, this is important because OWL is meant for knowledge representation and such knowledge is not necessarily complete. Take as an example an ontology of all the cars in the world, with a main class called *Car* and several subclasses for every brand and model. If someone sends you a new *Car* individual which is a completely new car, new brand and new model, then you can not assume it must belong to one of your car subclasses because you don't know if you have a complete representation of all car knowledge or not.

However, sometimes it is useful to be able to state that the subclasses represent a complete knowledge base. It is not particularly easy to come up with a good example for when this would be a perfect match, so let us look at a less perfect example. The class *ScheduleFlowline* contains 7 subclasses:

- iohn6:Flowline_5inch_Sch160
- iohn6:Flowline_5inch_Sch120
- iohn6:Flowline_6inch_Sch120
- iohn6:Flowline_8inch_Sch120
- iohn6:Flowline_10inch_Sch120
- iohn6:Flowline_18inch_Sch120
- iohn6:Flowline_10inch_Sch140

Even though we know that there might be a new flowline introduced in a few years, such as a *Flowline_8inch_Sch140*, or that there exists other flowlines

in other installations, for our purpose and application the group of flowlines is complete. In OWL such a statement can be made by saying that all the subclasses of a class are a disjunctive superclass of the class. For *ScheduleFlowline* the definition in manchester syntax is as follows:

```
Class: iohn6:ScheduleFlowline

    Annotations:
        rdfs:label "Schedule flowline",
        rdfs:comment "Superclass of ..."^^xsd:string

    SubClassOf:
        iohn6:Flowline_10inch_Sch120
         or iohn6:Flowline_10inch_Sch140
         or iohn6:Flowline_18inch_Sch120
         or iohn6:Flowline_5inch_Sch120
         or iohn6:Flowline_5inch_Sch160
         or iohn6:Flowline_6inch_Sch120
         or iohn6:Flowline_8inch_Sch120,
        iohn6:Flowline
```

The requirement 2.3.5 defines this subclassing and gives a good rationale for its use. It also gives an example implementation using SPARQL. The implementation is to generate SPARQL queries that checks that every individual of *ScheduleFlowline* is also an individual of one of the subclasses. A good use for this implementation is if we don't want to define the subclass restrictions in OWL, but rather in a separate requirements file. Then we could say that all *ScheduleFlowline* individuals must also be on of *FlowLine_10inch_Sch140*, *FlowLine_18inch_Sch120*, ...

Since OWL reasoners are open world, the definition for *ScheduleFlowline* written in manchester syntax above will not render the ontology inconsistent just because there exists an individual of type *ScheduleFlowline* that is not specified to be a type of any of the subclasses. The ontology will only become inconsistent if the reasoner discovers that the individual can not be a member of any of the subclasses, for instance by negation or by being a member of a disjoint class. The reasoner can also reason the individual to be a member of, say *FlowLine10_inchSch120* if it can reason it to not be a member of any of the other subclasses.

If someone comes up with a new flowline, then any individual of that type will be accepted by the ontology without any problems. The check for the requirement might fail until the ontology is updated to show the new flowline as a subclass of *ScheduleFlowline*, however that will always be the case for incomplete knowledge bases.

To automate the testing of this requirement we could go with the SPARQL implementation, however this would require extracting every set of classes and subclasses that meet the requirement and test all their members. An alternative idea is to write a SPARQL query to find all individuals that are a member of a

class where they should be a member of a subclass but are not. However being able to find out if a an individual should be a member of one of the subclasses is a Tbox query, since using disjunction in a SPARQL, where the reasoner can not return an individual as an instance of a specific subclass, is not possible.

```
SELECT ?a WHERE {
    ?a a ?type.

    FILTER(    ?type = iohn6:Flowline_10inch_Sch120
             || ?type = iohn6:Flowline_10inch_Sch140
             || ...
           )
}

SELECT ?a WHERE {

    OPTIONAL{?a a iohn6:Flowline_10inch_Sch120}
    OPTIONAL{?a a iohn6:Flowline_10inch_Sch140}
    OPTIONAL ...

}
```

Both queries above will only return individuals that are actual members of one or the other subclass. The query will not return any individuals that are members of one of the classes by reasoning when the reasoner can not find out which class the individual should be a member of.

DL-queries however allow this behaviour. A DL-query with disjunction will return all individuals that match the entire disjunction rather than the discrete parts, as SPARQL does. The DL-query "Flowline_10inch_Sch120 OR Flowline_10inch_Sch140 OR ..." will return all members, including individuals that are simple members of *ScheduleFlowline* and not stated members of any of the subclasses.

This way we can loop through all classes in the ontology and make a disjunctive query of all the subclasses for every class. Now there is no need to do a complicated Tbox query to find out which classes a subclasses of their own subclasses.

My implementation used OWLAPI to both loop through all the classes and subclasses and also to run the actual DL-queries against an ontology with data reasoned on by Pellet.

```
for (OWLClass owlClass : reasoner.getClasses()) {

    if (owlClass.getSubClasses(ontology).size() > 0) {
        String subClasses = "";
        for (OWLClassExpression owlClassExpression :
                owlClass.getSubClasses(ontology)) {
```

```
            subClasses += owlClassExpression.toString()
                            + " OR ";
        }

        //remove trailing " OR ".
        subClasses = subClasses.
            substring(0, subClasses.length() - 4);

        // Run query here
    }
}
```

And for running the query I used example code[35] by Matthew Horridge to parse the string into an OWLClassExpression which I could run on the pellet reasoner to get all individuals from the ontology. I then ran a separate query for every one of the subclasses to check that the results from the disjunctive query was equal to the joint result-set for every subclass. Any individual not in the joint result-set for every subclass did not meet the requirement, and could be reported to the user as a violation.

Chapter 10

Part 8 adherence

10.1 Integrity Constraints

With semantic technologies, open world reasoning is usually the norm. With open world reasoning there is in essence a quantum effect of uncertainty for any information that is not stated. Unstated facts are simply unknown, rather than untrue. With the exception of facts that are stated to be untrue.

This is best explained with an example. Consider a simple ontology with two disjoint classes, because I am a possessive person I will call the classes *Mine* and *SomeoneElses*. If an object does not belong to someone else, then it must belong to me (because I am possessive).

```
Class: Mine
    EquivalentTo: NOT SomeoneElses

Class SomeoneElses
```

If we introduce the crown jewels, and fail to specify that it belongs to someone else, then it should belong to me. Since *Mine* is equivalent to the set of things that are not in *SomeoneElses*. However, because of the open world assumption, the reasoner does not assume that the crown jewels can not belong to someone else, simply because they are not stated as such.

However, if the crown jewels is stated as belonging to "NOT SomeoneElses", then by reasoning the crown jewels now belong to me. So it is not a case of simply ignoring negation, but rather about being guarded when considering possible unknown behaviour.

Looking at DBpedia the open world assumption is obviously a good choice, because there are so many cases of missing information it would be impossible to answer a question like, "in which countries can you not find squirrels".

With the open world assumption there is one particular behaviour that is difficult to specify. One great feature of traditional databases is the ability to specify a required column in a table. Every row in that table needs to have a non-null value for that field. Null means unknown, which is different from

stating a value of none[36]. Open world assumption treats a missing value as a null value, so unless the value is stated as missing it will be handled as if it could exist. An existential restriction is a requirement that such a value exist, and due to the open world assumption if the value could possibly exist for the current Tbox and Abox, then the Tbox and Abox are consistent[37, page 129].

For requirement 3.2.2, a reasoner with closed world abilities such as Pellet ICV (Integrity Constraint Validator), or its incarnation in the Stardog[38] tripple store, is a possible solution. An alternative, to using an existing solution, is constructing SPARQL queries to check for missing attributes.

The first query would identify all existential restrictions (in this example), and with that data a series of queries could be constructed to check instance data.

```
select * where {

?class rdf:type owl:Class ;

    rdfs:subClassOf [ rdf:type owl:Restriction ;
                      owl:onProperty ?property ;
                      owl:someValuesFrom ?type
                    ] .
}
```

From this query we would get a list of classes and their existential restrictions. This result would be used to construct queries on the form of this template (handlebars style[39]).

```
select * where {

    ?a a {{?class}}.

    FILTER(!EXISTS{?a {{?property}} [a {{?type}} ]})

}
```

Different queries have to be used for universal restrictions and for cardinality restrictions.

Chapter 11

Satisfiability

One seemingly missing requirement is one for concept satisfiability[40]. When extending an ontology, conservative extensions prevents concepts in the original ontology from becoming unsatisfiable. With the more restrictive addition to the conservative extensions requirement (detailed further in chapter 16 section 16.2 on page 61) the extending ontology must also be satisfiable when incorporated into the final ontology.

However, the first ontology, the very base of all the extensions, does not need to be satisfiable. This should never be a problem, as it is of little use to make an ontology with an unistansiable class. The only time it could become a problem is when a class beomces unsatisfiable because of instances in another class. This could happen if the class is to be be equivalent to a set of individuals and all those individuals become members of disjoint classes.

I would recommend that the intial ontology be an extended model conservative extension of the empty ontology. This will sufficiently limit unsatisfiability.

Chapter 12

Criteria implementation

As part of this thesis I created a Java program to automatically check all the simple criteria. I wrote the program using the Test-driven development methodology[41] (TDD). Essentially every criteria was first specified as a test, then I would write the code to pass the test. After passing a test I would either make changes to it to specify any newly discovered requirements or continue onto the next criteria by writing a new test.

The overall architecture is based on a controller that reads the ontology and data files and then runs the criteria checkers. When a criteria checker requires a model, it requests this from the controller and specifies if it requires reasoning. Essentially Just-in-time-reasoning that can be cached if Jena uses forward-chaining[42] or if the *InfModel* caches its own inferences.

Each criteria is built as an extension to an abstract class, namely *ValidatorAbstract*, which has the abstract method *validate()* which returns void. *ValidatorAbstract* also has a few helper methods and a list for keeping score of errors. The helper methods are for running a SPARQL query and for reporting an error. Also there is a nice method for printing errors to standard output.

When extending *ValidatorAbstract* the *validate()* method must be overridden with an implementation. This implementation typically defines and runs a SPARQL query and reports the results to *reportError()* (which supports Jena *ResultSet*). An example of this is *ValidateSubjectsHaveExplicitType* which checks that subjects have a type.

```
public class ValidateSubjectsHaveExplicitType
                    extends ValidatorAbstract {

    @Override
    public void validate() {
        String query =
            "select distinct ?a where {\n" +
            "?a ?b ?c.\n" +
            "MINUS{?a rdfs:label \"IGNORE\".}" +
```

```
            "FILTER (!EXISTS{ ?a rdf:type ?d. \n" +
            "?d rdfs:label \"VOCAB\"\n" +
            "FILTER(?d != owl:Thing)})\n" +
            "}";

        reportError(sparql(model.add(vocab), query));

    }
}
```

ValidatorAbstract also has three variables for the model, the ontology and the vocabulary. This way the implementation for explicit typing can be extended by a class that requires explicit typing after RDFS reasoning or after OWL reasoning. An example of this is *ValidateSubjectsHaveExplicitTypeRDFS* which has the following validate method and extends *ValidateSubjectsHaveExplicitType*:

```
@Override
public void validate() {

    model = Controller.rdfsReasoning();

    super.validate();

}
```

I also looked at two other ways of implementing this. One was with functors, where the *ValidateSubjectsHaveExplicitType* class (without reasoning) would take a functor class that does all the reasoning. This is a very fancy way of extending the capabilities of a class, but doesn't chime too well with my existing use of an abstract class.

The other alternative I looked at was using annotation[43] to inject code at compile time or runtime. This is a very common approach for big Java systems that run as containers in a server. For instance Spring[44] has support for annotation to specify that a certain method should be executed right before every call to some other method[45]. This way it would be possible to declare a requirement, such as reasoning, instead of writing the code to do the reasoning.

```
@Requires.reasoning.rdfs
@Override
public void validate() {

        super.validate();

}
```

I mainly looked at Plastic[46] from Apache Tapestry[47] to accomplish this. It has a framework for selecting annotations and wiring them up to methods. The catch was that without using a server I would have to manually run the

byte-code through a decompiler and parser such as ASM[48] before running my program. Otherwise I would be stuck with calling methods through the Plastic framework, which would also include class instantiation.

Declaratively writing all the requirements would be very convenient, but creating a language to allow that would go far and wide beyond this thesis. However a pointer to future implementations, designing a Domain-specific language with JetBrains MPS[49] is not all that hard.

The test setup uses JUnit 4.1[50] contained a test for every validator (implementation of criterion) and a helper class called TestFramework that all the tests extended. The TestFramework handles setting up the controller with the correct model, ontology and vocabulary as well as checking if the errors produced by the test are consistent.

When setting up a test, strings for the model, ontology and vocabulary are added to the *TestFramework* with a *setUp()* method annotated with @Before (from JUnit). Then the test method runs, *testValidate()* which first adds all the expected errors, then instantiates the validator, runs the validator and then calls the *TestFramework* to assert the expected errors with by using the *getErros()* method from the abstract class *ValidatorAbstract*.

For testing the literal validator I used the following test data:

```
sim:Homer rdf:type foaf:Person ;
    foaf:name "Homer Simpson";
    foaf:age "36"^^xsd:int.

sim:Marge rdf:type foaf:Person ;
    foaf:name "Marge Simpson"@en;
    foaf:age "33".

sim:Bart rdf:type foaf:Person ;
    foaf:name "Bart Simpson"@en;
    foaf:age "12"^^xsd:int.

sim:Lisa rdf:type foaf:Person ;
    foaf:name "Lisa Simpson"^^xsd:string;
    foaf:age "12"^^xsd:int.
    sim:PlainLiteralSimpson rdf:type foaf:person ;
    foaf:name "PlainLiteralSimpson"^^rdf:plainLiteral.
```

The expected errors are for *Homer* since his name does not have a language tag, for *Marge* who's age is not an *xsd:int* and for *Lisa* who's name, like *Homer's*, does not have a language tag. The difference between *Lisa* and *Homer* is that *Lisa's* name is specified as an *xsd:string*, while *Homer's* is implicitly an *xsd:String*. Which shows that the requirement for literals needs more refining, also to test my hypothesis that *rdf:plainLiteral* passes my tests I added this as an alternative name for Lisa.

The test for *ValidateLiteral* is shown below. Each error is added as an expected error to the *TestFramework*. The format is the same as is returned

from *toString()* for a result from Jena and the ordering was initially a problem until I decided to split the strings and extract the values. Alternatively I could have listed the values as parameters to the method call, or as an array.

```
@Test
public void testValidate() throws Exception {

    addExpectedErrors(
        "( ?a = <http://.../simpsons#Homer> ) " +
        "( ?c = \"Homer Simpson\" ) " +
        "( ?b = <http://xmlns.com/foaf/0.1/name> )"
    );

    addExpectedErrors(
        "( ?b = <http://xmlns.com/foaf/0.1/age> ) " +
        "( ?c = \"33\" ) " +
        "( ?a = <http://.../simpsons#Marge> )"
    );

    addExpectedErrors(
        "( ?a = <http://.../simpsons#Lisa> ) " +
        "( ?b = <http://xmlns.com/foaf/0.1/name> ) " +
        "( ?c = \"Lisa Simpson\"^^xsd:string )"
    );

    ValidatorAbstract val = new ValidateLiteral();
    val.validate();

    assertExpectedErrors(val.getErrors());

}
```

I did not test how well my implementation handled large scale ontologies with large data sets. Also I did not have large data sets at hand and would have had to generate this. My implementation is limited by the performance of SPARQL queries in Jena and by how the reasoner. Pellet is not a particularly fast reasoner, and if the OWL language is sufficiently limited it may have been interesting to try to use Elk[51].

Part IV

Conservative extensions of ISO 15926

Chapter 13

Preliminaries

For DL semantics see the "Handbook on ontologies"[52].

Definition 1. [Concept Satisfiability] Given an ontology O and a class A, there is a model of O in which the interpretation of A is a nonempty set.[53, Derived]

Definition 2. [Model conservative extension]
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be TBoxes. We say that $\mathcal{T}_1 \cup \mathcal{T}_2$ is a model conservative extension of $\mathcal{T}_1$ iff for every model $\mathcal{I}$ of $\mathcal{T}_1$, there exists a model of $\mathcal{T}_1 \cup \mathcal{T}_2$ which can be obtained from $\mathcal{I}$ by modifying the interpretation of the atomic concepts and roles in $sig(\mathcal{T}_2) \backslash sig(\mathcal{T}_1)$ while leaving the domain fixed and all other interpretations fixed. [Derived from definition in [22]]

Definition 3. [Gamma equality] Let $\mathcal{I}$ and $\mathcal{I}'$ be interpretations. $\mathcal{I} =^{\Gamma} \mathcal{I}'$ iff $\Delta^{\mathcal{I}} := \Delta^{\mathcal{I}'}$ and $r^{\mathcal{I}} := r^{\mathcal{I}'}$ and $A^{\mathcal{I}} := A^{\mathcal{I}\prime}$ for all $r, A \in \Gamma$.

Lemma 4. *$\mathcal{I}' \models \varphi$ if $\mathcal{I} \models \varphi$ and $\mathcal{I}$ is a model of a Tbox $\mathcal{T}$ with a signature $\Gamma = sig(\mathcal{T})$ and $\mathcal{I} \overset{\Gamma}{=} \mathcal{I}'$.*

Definition 5. [Original/Base ontology] Given ontologies $\mathcal{T}_1$ and $\mathcal{T}_2$, when $\mathcal{T}_1 \cup \mathcal{T}_2$ is the extended ontology, then $\mathcal{T}_1$ is the original ontology.

Definition 6. [Extended ontology] Given ontologies $\mathcal{T}_1$ and $\mathcal{T}_2$, then $\mathcal{T}_1 \cup \mathcal{T}_2$ is the extended ontology of $\mathcal{T}_1$.

Definition 7. [Extending ontology] Given ontologies $\mathcal{T}_1$ and $\mathcal{T}_2$. When $\mathcal{T}_1 \cup \mathcal{T}_2$ is the extended ontology of $\mathcal{T}_1$ then $\mathcal{T}_2$ is considered the extending ontology.

Definition 8. [Disruptive extension] An extensions is a disruptive extension iff it is not a conservative extension.

Chapter 14

Introduction to conservative extensions

There are two types of conservative extensions. Model conservative extensions[15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] and deductively conservative extensions[ref same as above]. Model conservative extensions is the stronger extension, requiring models to be equal with respect to the original signature. Deductive conservative extensions are weaker, an extension is deductively conservative if and only if the possible deductions are the same for the original ontology and the extended ontology with respect to the original signature.

Loosely put the conservative extension is meant to force an ontology to be considered complete, and any extension should not add knowledge to the original ontology that should have been defined in the first place. If you put on a pair of glasses that are crafted specifically for the original ontology, then however thoroughly you examine the extended ontology, there will be no way of distinguishing it from the original ontology when it is a conservative extension.

The first feature of a conservative extension is the preservation of consistency. If the original ontology is consistent, then the extended ontology will also be consistent. Any Abox limited to the original ontology will be consistent with the extended ontology. Without this limitation it is trivial to craft an Abox that will prove inconsistent with the extended ontology.

Consider two users of an ontology, Steve and Emma. Steve and Emma have for a long time been able to share data between themselves because they have been using the exact same ontologies. When Emma decided to expand her ontology, she did so with a model conservative extension, thus still being able to read and understand Steve's data. Steve could also receive data from Emma, even with her modifications because the ontology was backwards compatible with his own and he could safely ignore unknown data.

Steve could even go so far as to incorporate Emma's ontology, because the systems he had built around his own ontology would not be able to perceive any change.

There is a slight exception to this rule. Consider that Steve and Emma have ontologies of cars with every car brand in the book. Then one day a new brand is introduced, but only Emma adds it to her ontology. Let us call this brand, Z. Z is a brand of car so every vehicle in the Z class must also be of type car. Emma has 100 cars in her dataset, 5 of these being Z brand cars. When she counts all her cars she finds that she has 100 cars. However, unless she explicitly states that her 5 Z brand cars are of type car, then Steve will only count 95 cars when he looks at Emma's dataset. If Emma does explicitly state that her Z brand cars are cars, then Steve will count 100 cars, but suddenly he now has 5 cars of which he does not know the brand.

The correct solution is to ignore Emma's Z brand cars, because Steve does not know if they are really cars. Emma's ontology is a conservative extension of Steve's, so every car Steve has is acknowledged by Emma, but every new car that Emma has does not need to be acknowledged by Steve. However, if Emma states that her 5 Z brand cars are of type car, then Steve must acknowledge this and instead only count cars with a specified brand if he wants to be certain that he can determine the brand of the car.

The definition of a model conservative extension is as follows:

Definition 9. [Model conservative extension]

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be TBoxes. We say that $\mathcal{T}_1 \cup \mathcal{T}_2$ is a model conservative extension of $\mathcal{T}_1$ iff for every model $\mathcal{I}$ of $\mathcal{T}_1$, there exists a model of $\mathcal{T}_1 \cup \mathcal{T}_2$ which can be obtained from $\mathcal{I}$ by modifying the interpretation of the atomic concepts and roles in $sig(\mathcal{T}_2) \backslash sig(\mathcal{T}_1)$ while leaving the domain fixed and all other interpretations fixed. [Derived from definition in [22]]

It builds of the notion of two ontologies, $\mathcal{T}_1$ and $\mathcal{T}_2$. $\mathcal{T}_1$ is the *original* ontology, also known as the *base* ontology. This ontology is considered as a complete and sound ontology that regardless of what instance data is provided will always return complete and sound answers to queries. $\mathcal{T}_2$ is the *extending* ontology, it contains the new axioms that are required by the application. The *extended* ontology is the union of both these ontologies, $\mathcal{T}_1 \cup \mathcal{T}_2$.

The key to a model conservative extension lies in how any model for the base ontology can be morphed into a model for the extended ontology without making any changes to the interpretations that are specific for the base ontology.

Two simple ways to consider conservative extensions. 1. Can the axioms in the extension be interpreted to be empty? 2. Can the axioms in the extension be interpreted as equivalent to axioms in the original ontology interpretation?

An extension that is not conservative is called a disruptive extension (in this thesis). By disruptive it is meant that the extension disrupts and breaks something inherent to the base ontology.

A weaker conservative extension if the deductive conservative extension. Every extension that is model conservative is also deductively conservative, but every deductively conservative extension is not necessarily a model conservative extension.

Definition 10. [Deductive conservative extension]

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be TBoxes and $\mathcal{L}$ be a language. We say that $\mathcal{T}_1 \cup \mathcal{T}_2$ is a deductive conservative extension of $\mathcal{T}_1$ iff for every axiom φ in $\mathcal{L}$ with $sig(\varphi) \subseteq sig(\mathcal{T}_1)$ then $\mathcal{T}_1 \models \varphi$ iff $\mathcal{T}_1 \cup \mathcal{T}_2 \models \varphi$. (Derived from [18])

Deductive conservative extensions are easier to test by computers[22] than model conservative extensions.

As a side note, there is a term called module extraction which is based on conservative extensions. One use for module extraction is for reasoning purposes, if most of the ontology is compliant with a less complex OWL profile, then being able to extract the more complex parts and do reasoning on this part separately will make the reasoning less computationally demanding. The original ontology is seen as a conservative extension to the extracted module[18].

14.1 Query Conservative

One of the most interesting properties of a conservative extension is how queries over the original ontology and queries over the extended ontology have the same results. Essentially we could call it a query conservative extension.

First we need some definitions of soundness and completeness. Informally a sound result from a query is one where there are no wrong answers in the result set. A complete result from a query is a result that contains every single correct result.

Formally soundness with regard to an original ontology an extended ontology:

Definition 11. [Query conservative soundness] Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be Tboxes. Let $\mathcal{I}$ be an Abox restricted to $sig(\mathcal{T}_1)$. For every query Q over $\mathcal{T}_1(\mathcal{I})$, $Q(\mathcal{T}_2(\mathcal{I})) \subseteq Q(\mathcal{T}_1(\mathcal{I}))$.

The assumption is that when running a query over the original ontology and dataset, the results from this query are considered the correct results. If the extension makes new results available, then these must be incorrect since the original ontology already returns all the correct results. In essence the original ontology provides all the complete and sound answers to any query.

Definition 12. [Query conservative completeness] Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be Tboxes. Let $\mathcal{I}$ be an Abox restricted to $sig(\mathcal{T}_1)$. For every query Q over $\mathcal{T}_1(\mathcal{I})$, $Q(\mathcal{T}_1(\mathcal{I})) \subseteq Q(\mathcal{T}_2(\mathcal{I}))$.

Following on the assumption above, completeness dictates that the original ontology provides every answer. Any extension should not remove any of those answers since they are correct.

To start to compare conservative extensions with conservatively preserved query results we must first limit our queries to only make use of elements in the signature of the original ontology. Being able to ask arbitrary queries makes it possible to query something that is only possible in the extended ontology. If

we limit our Abox to the signature of the original ontology, the queries that can be performed are fairly limitless.

Take as an example an ontology for vehicles. It has cars and trucks and busses and so on.

```
Class: Car
Class: Truck
Class: Bus
```

If we extend the ontology by saying that there are small cars:

```
Class: SmallCar
    SubClassOf: Car
```

Then a query for all individuals in Car will exclude SmallCars when asked over the original query but will include them when asked over the extended ontology. Technically it is sufficient to limit the Tbox to the signature of the base ontology, since any Abox assertion implicitly states that the asserted class is an *rdfs:Class* due to the range on *rdf:type*[54].

From the definition of model conservative extension, a model $\mathcal{I}$ can be extended to a model $\mathcal{I}'$ so that $\mathcal{I} \overset{\Gamma}{=} \mathcal{I}'$ where Γ is a signature of $\mathcal{T}_1$ with regard to the definition. This is directly related to the limitation of the Tbox and Abox for queries.

Atomic queries are trivially both sound and complete since they simply list individuals in a class or property. Consider an Abox limited to the original ontology that is consistent with respect to that ontology. This Abox is then the model $\mathcal{I}$ and since the model conservative extension makes this model gamma-equal to the extended model for the extended ontology, $\mathcal{I} \overset{\Gamma}{=} \mathcal{I}'$, then any atomic queries limited to the signature of the original ontology must be equal.

Chapter 15

Approaches to testing conservative extensions

Getting a rough idea of whether an extension was conservative or not was very important in the early stages of this thesis. There are papers on conservative extensions with helpful examples, especially so is the paper[22] by Lutz, Walther and Wolter where they show how existential restrictions together with disjointness causes an extension to be disruptive.

They introduce a term called a witness concept and say that if there exists a witness concept for the original ontology that is not true in the extended ontology, then the extended ontology is not a conservative extension of the base. I based my computer program on this idea in order to brute force check a set of atomic and complex concepts.

All the concepts were limited to the signature of the base ontology, which helpfully always includes $topObjectProperty$ and $Thing$. This way it is easy to see that the following extension is disruptive.

$$\begin{aligned} \mathcal{T}_1 :& A \sqsubseteq \exists r.C \\ \mathcal{T}_2 :& D \sqsubseteq \exists r.M \\ & M \sqcap C \sqsubseteq \bot \end{aligned}$$

Where $A \sqsubseteq_{\leq 1} r.Thing$ is true in $\mathcal{T}_1$ but false in $\mathcal{T}_1 \cup \mathcal{T}_2$.

I found the easiest way of testing was to query for subclasses of the witness concept rather than check if the entire axiom was consistent.

Limited to roles and concepts in the signature of the base ontology, $r, A \in sig(\mathcal{T}_1)$, I generated every possible witness concept on the form of:

$$\exists r.A \tag{15.1}$$
$$\forall r.A \tag{15.2}$$
$$_{=n}r.A \quad | \quad 0 < n < 100 \tag{15.3}$$
$$_{\leq n}r.A \quad | \quad 0 < n < 100 \tag{15.4}$$
$$_{\geq n}r.A \quad | \quad 0 < n < 100 \tag{15.5}$$

The program generated these in Manchester syntax and queried the original ontology for subclasses and the extended ontology for subclasses and printed any differences.

To an ontology that adds the following statement: $SimpleStatement \sqsubseteq_{=1} has.Artefact \sqcap \forall has.Artefact$ where $has, Artefact$ and $SimpleStatement$ are all in the signature for the original ontology, then the program prints the following errors (and more):

```
missing from A:

has min 1 (Artefact)     Node( <.../#SimpleStatement> )
has max 1 (Artefact)     Node( <.../#SimpleStatement> )
has max 2 (Artefact)     Node( <.../#SimpleStatement> )
has exactly 1 (Artefact)Node( <.../#SimpleStatement> )
has only (Artefact) Node( <.../#SimpleStatement> )
```

This picked up most of the disruptive extensions, though still leaving behind more complex ontologies. Lacking so far where more realistic conjunctions of witness concepts. A common trait for template definitions is to require that there be exactly one individual of a limited type for a given property, on the form of $A \sqsubseteq_{=1} r.B \sqcap \forall r.B$.

Such witness concepts were generated by creating all combinations of *only* and *min*, *only* or *min*, *only* and *exactly*, *only* or *exactly*.

$$\forall r.A \quad \sqcap \quad _{=n}r'.A' \tag{15.6}$$
$$\forall r.A \quad \sqcup \quad _{=n}r'.A' \tag{15.7}$$
$$\forall r.A \quad \sqcap \quad _{\leq n}r'.A' \tag{15.8}$$
$$\forall r.A \quad \sqcup \quad _{\leq n}r'.A' \tag{15.9}$$

Where it is important to note that the left and right side do not need to share roles and concepts, since it is quite possible to require all types to be of A and state that it requires a minimum of n individuals of some type that is subclass to A. This may be useful to define a special minimum of some type without restricting all individuals to that type. An example could be an apartment block that requires every apartment to have one paying tenant but also wants to register other non paying tenants: $Apartment \sqsubseteq \exists hasTennant.PayingTennant \sqcap \forall hasTennant.Tennant$.

Also some restrictions may use complex concepts, i.e. $\exists r.(A_1 \sqcap A_2 \sqcup A_3)$ and also with atomic negation. The only way to safely handle this is to generate all possible combinations of these concepts by using the disjunctive normalform.

$$A_1 \sqcap A_2 \tag{15.10}$$
$$\neg A_1 \sqcap A_2 \tag{15.11}$$
$$A_1 \sqcap \neg A_2 \tag{15.12}$$
$$\neg A_1 \sqcap \neg A_2 \tag{15.13}$$
$$(A_1 \sqcap A_2) \sqcup (\neg A_1 \sqcap A_2) \tag{15.14}$$
$$(A_1 \sqcap A_2) \sqcup (A_1 \sqcap \neg A_2) \tag{15.15}$$
$$... \tag{15.16}$$
$$(A_1 \sqcap A_2) \sqcup (\neg A_1 \sqcap A_2) \sqcup (A_1 \sqcap \neg A_2) \tag{15.17}$$
$$... \tag{15.18}$$

Generating all these complex concepts is prohibitively expensive. For n atomic concepts the number of unique complex concepts is 2^{n^n}. With 4 atomic concepts we get 65,536 complex concepts, which is where the limit for reasonable computations. At 5 atomic concepts we have 33,554,432 complex concepts and at 6 atomic concepts this rises to 68,719,476,736 which would require 64 GiB to represent if every complex concept could be represented by only 1 byte (which it can not).

Since anything above 4 atomic concepts was impossible to calculate for me, I simplified this to only creating all the conjugate complex concepts without negation and a special complex concept that was equivalent to $Thing$.

I also tested if every class was satisfiable. This was built into the reasoner I used as a method called $getUnsatisfiableClasses()$. I initially used Pellet, but it crashed on several of the complex concepts with an error in one of its tableaux methods. So I switched to using Hermit[55].

Nesting was not tested. This would have been two intensive, though a strict limit of single level nesting could have been possible to implement.

15.1 Tested ontologies and results

I used four base ontologies in my testing.

Case 1. Empty Statement

```
ObjectProperty: has

Class: RDLTemplateStatement
Class: Artefact

Class: SimpleStatement
    SubClassOf:
```

```
RDLTemplateStatement
```

Case 2. Simple Statement

```
ObjectProperty: has

Class: RDLTemplateStatement
Class: Artefact

Class: SimpleStatement
    SubClassOf:
        RDLTemplateStatement,

        (has only Artefact)
        and (has exactly 1 Artefact)
```

Case 3. Double Simple Statement

```
ObjectProperty: has
ObjectProperty: connectedTo

Class: RDLTemplateStatement
Class: Artefact

Class: SimpleStatement
    SubClassOf:
        RDLTemplateStatement ,

        (connectedTo only Artefact)
        and (connectedTo exactly 1 Artefact) ,

        (has only Artefact)
        and (has exactly 1 Artefact)
```

Case 4. New Simple Statement

```
ObjectProperty: has, has2
ObjectProperty: has2

Class: RDLTemplateStatement
Class: Artefact2
Class: Artefact

Class: SimpleStatement
    SubClassOf:
        RDLTemplateStatement,
```

```
(has2 only Artefact2)
and (has2 exactly 1 Artefact2)
```

For these 4 cases I tested, combinatorially, a selection of extensions. This selection included modifying existing template statements and introducing new template statements.

I tested restrictions using existing properties and classes, new properties and classes and sub-properties and subclasses; and all combinations of those. From this I discovered that introducing a new template statement is always conservative, regardless of if the new template statement includes restrictions utilising new properties or classes, existing properties and classes or sub-properties or subclasses.

Since every class is a subclass of itself, then introducing a new existential restriction that does not force a new individual by disjointness, then this restriction will be satisfied by the fact that the class is a subclass of itself.

$$A \sqsubseteq \exists r.C$$

For a class A that is in the signature of the base ontology and r and C are role and concept introduced by the extension (not in the base ontology signature), then it is seemingly always conservative since in any model, r and C can be satisfied as the equivalent to how the model satisfies $A \sqsubseteq A$.

This leads to a generalisation that if a newly introduced restriction can be interpreted as the equivalent of some existing fact of the original ontology, then it is conservative. For instance:

$$\begin{aligned}\mathcal{T}_1 :& A \sqsubseteq RDLTemplateStatement\\ & A \sqsubseteq= 4\, r.C\\ \mathcal{T}_2 :& A \sqsubseteq\leq 4\, s.D\end{aligned}$$

$\mathcal{T}_1 \cup \mathcal{T}_2$ is a model conservative extension of $\mathcal{T}_1$ because every model $\mathcal{I}$ of $\mathcal{T}_1$ can be extended to a model of $\mathcal{I}'$ of $\mathcal{T}_1 \cup \mathcal{T}_2$ by making $r^{\mathcal{I}} := s^{\mathcal{I}'}$ and $C^{\mathcal{I}} := D^{\mathcal{I}'}$.

However, if C and D are disjoint, then $\mathcal{T}_1 \cup \mathcal{T}_2$ requires the addition of four new elements to satisfy the axiom in $\mathcal{T}_2$.

This can also be extended so that a new restriction is conservative if it can be mapped onto more than one fact. As in the following example:

$$\begin{aligned}\mathcal{T}_1 :& A \sqsubseteq RDLTemplateStatement\\ & A \sqsubseteq= 4\, r.C\\ & A \sqsubseteq= 4\, p.B\\ & B \sqcap C \sqsubseteq \bot\\ \mathcal{T}_2 :& A \sqsubseteq\leq 8\, s.D\end{aligned}$$

As well as being able to generate many DL-queries, my test program was also able to do full OWL reasoning with Pellet and compare the output from the base ontology with the extended ontology limited to the signature of the base ontology. Pellet does not list all consequences of an ontology. I tested it with the example from "Conservative Extensions in Expressive Description Logics"[22] with essentially boils down to:

$$\begin{aligned} A &\sqsubseteq \exists r.B \\ A &\sqsubseteq \exists r.C \\ B \sqcap C &\sqsubseteq \bot \end{aligned}$$

Entailing $A \sqsubseteq \leq 2\, r.\top$ because B and C are disjoint. Pellet does not show this result. Not even when B and C are set to be subclasses of a common class D, where it would entail $A \sqsubseteq \leq 2\, r.D$. This is of course picked up by the DL-queries generated by the other section of the testing program.

15.2 Existing solutions

I gave ProSÉ[19] a try, a Protégé plugin for reusing ontologies. It supports model extraction and engineering and handles the conservative extension parts for the user. I did not manage to get it to work, but it is still worth mentioning.

The Locality Module Extractor[56] based on the work by Jimenez Ruiz, Sattler and Schneider is being mainted by the Information Systems Group at Oxford University. This is much simpler than ProSÉ, however only supports modularisation and not engineering. It does still handle the conservative extension parts for the user.

I did not go into depth with any existing tools for checking conservative extensions.

Chapter 16

Introducing new template declarations.

A new template declaration is represented in OWL as a new class (with a new name) subclassed from *RDLTemplateStatement* and containing a number of superclass restrictions. These restrictions are used to map input from a template statement to the proper types through properties that represent the name of the variable.

A new policy change, such as a requirement to monitor sea temperatures at an oil rig, should be implemented as new template declarations that have no side effects on implementations based on the previous version of the ontology. To achieve this the new template declarations should amount to a conservative extension of the original ontology. Also the ontologies are built as levels on top of each other by importing the one below. These should also be conservative extensions of the one below.

The motivation for a conservative extension is primarily consistency. Consistency at Tbox level and at data level as well as query consistency. The latter requiring a model conservative extension[57, page 4].

Alternately to adding a new template declaration, a company could attempt to modify an existing template declaration by adding new superclass restrictions. This is a very invasive approach, also on the semantics of the implementation of the template declaration in levels above the OWL serialisation. A second approach is to subclass an existing template declaration. More on this later.

16.1 Typical implementation

A typical new template declaration would be a new class subclassing *RDLTemplateStatement*, where the new class is outside the signature of the original ontology. Let $\mathcal{T}$ be a Tbox with the following axioms and Γ be a signature so that $\Gamma = sig(\mathcal{T})$.

$$T \sqsubseteq RDLTemplateStatement$$

$$T \notin \Gamma$$

A template declaration with one input variable will result in one superclass restriction:

$$T \sqsubseteq =_1 p.R \sqcap \forall p.R$$

Where p is a property and R is an atomic class. p is typically newly defined while R is usually an existing concept or a subclass thereof.

$$p \notin \Gamma$$

$$R \vee Q \in \Gamma | R \sqsubseteq Q$$

As a simplification, every sentence in the extension will be on the form of $T \sqsubseteq C$ where T is a new concept defined in the extension and not found in the original ontology. C can be any arbitrary expression, be it *RDLTemplateStatement* or an existential and universal quantifier. This simplification makes it simpler to prove that the common extension is always a conservative extension.

Definition 13. [Gamma equality] Let $\mathcal{I}$ and $\mathcal{I}'$ be interpretations. $\mathcal{I} =^{\Gamma} \mathcal{I}'$ iff $\Delta^{\mathcal{I}} := \Delta^{\mathcal{I}'}$ and $r^{\mathcal{I}} := r^{\mathcal{I}'}$ and $A^{\mathcal{I}} := A^{\mathcal{I}\prime}$ for all $r, A \in \Gamma$.

Lemma 14. *$\mathcal{I}' \models \varphi$ if $\mathcal{I} \models \varphi$ and $\mathcal{I}$ is a model of a Tbox $\mathcal{T}$ with a signature $\Gamma = sig(\mathcal{T})$ and $\mathcal{I} \stackrel{\Gamma}{=} \mathcal{I}'$.*

Theorem 15. *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are Tboxes, Γ is a signature such that $sig(\mathcal{T}_1) = \Gamma$ and $\mathcal{T}_2$ only containing axioms in the form of $T \sqsubseteq C$ where $T \notin \Gamma$ and C is an arbitrary expression, then $\mathcal{T}_1 \cup \mathcal{T}_2$ is always a conservative extension of $\mathcal{T}_1$.*

Proof. $\mathcal{T}_1$ and $\mathcal{T}_2$ are Tboxes and Γ a signature such that $sig(\mathcal{T}_1) = \Gamma$. $\mathcal{T}_2$ contains only axioms in the form of $T \sqsubseteq C$ for a concept symbol $T \notin \Gamma$ with an arbitrary expression for C.

Let $\mathcal{I}$ be an interpretation where $\mathcal{I} \models \mathcal{T}_1$ (1)

Define $\mathcal{I}'$ as $\mathcal{I}' \stackrel{\Gamma}{=} \mathcal{I}$ where $T^{\mathcal{I}'} := \emptyset$ for $T \in sig(\mathcal{T}_2)/sig(\mathcal{T}_1)$ (2)

To prove a model conservative extension $\mathcal{I}'$ must be a model for $\mathcal{T}_1 \cup \mathcal{T}_2$

- $\varphi \in \mathcal{T}_1 \stackrel{(1)}{\hookrightarrow} \mathcal{I} \models \varphi \stackrel{(lemma\,14)}{\hookrightarrow} \mathcal{I}' \models \varphi$
- $\varphi \in \mathcal{T}_2 \hookrightarrow \varphi : T \sqsubseteq C$ for a $T \in sig(\mathcal{T}_2) \backslash sig(\mathcal{T}_1)$
  - $\stackrel{(2)}{\hookrightarrow} T^{\mathcal{I}\prime} = \emptyset \hookrightarrow \mathcal{I}\prime \models T \sqsubseteq C$

□

Since theorem 15 shows that all common template declarations are model conservative extensions, and a model conservative extension is query conservative and consistent if extending a consistent ontology, these common template declarations will not break implementations meant for the original ontology.

To put this into perspective, a client who has extended the base ontologies of ISO 15926 with custom templates on the form described above can send and receive data files with anyone using the original ontology or other model conservative extensions of the original ontology.

16.2 Satisfiability

One of the main drawbacks of only using model conservative extensions is that they do not extend any verification onto the extended model except for consistency and verification related to the original ontology. Mainly this means that the extended ontology can contain named concepts that are unsatisfiable due to either the integration between the original ontology and the extension or simply because they were unsatisfiable in the extension.

$$\begin{aligned}\mathcal{T}_1 :& RDLTemplateStatement \sqsubseteq Thing\\ & r \sqsubseteq rg(A)\\ \mathcal{T}_2 :& D \sqsubseteq RDLTemplateStatement\\ & D \sqsubseteq{=_1}\ r.S \sqcap \forall r.S\\ & S \sqsubseteq \neg A\end{aligned}$$

$\mathcal{T}_1 \cup \mathcal{T}_2 \models D \equiv \bot$ which means that D is unsatisfiable even when $\mathcal{T}_1 \cup \mathcal{T}_2$ is a conservative extension of $\mathcal{T}_1$, which is proven by theorem 15 since every axiom in $\mathcal{T}_2$ is on the form of $T \sqsubseteq C$ where $T \notin sig(\mathcal{T}_1)$. Being unsatisfiable is not a very useful property, and any unsatisfiable concepts should rather be left out to avoid someone trying to instantiate the concept and make the entire ontology inconsistent. To mitigate this, it is clear that a more restrictive supplement to a model conservative extension is needed. One where every concept or role has to be instantiable, or better, defined as having a non-trivial interpretation.

Definition 16. [**Non-trivial interpretation**]
Let $\mathcal{T}$ be a Tbox. $\mathcal{I}$ is a non-trivial interpretation of $\mathcal{T}$ iff every role and concept $r, A \in sig(\mathcal{T})$ have interpretations such that $r^{\mathcal{I}} \neq \emptyset$ and $A^{\mathcal{I}} \neq \emptyset$ (unless $r = Nothing$ or $A = Nothing$). A **non-trivial model** is a non-trivial interpretation that is also a model for the given ontology.

By incorporating the non-trivial model into the definition of a model conservative extension it is possible to force extensions to include only roles and concepts that are satisfiable in the final ontology. However, simply latching the non-trivial model onto the model conservative extension definition would not work, since any empty model of the original ontology could not possible have a

non-trivial model when extended by another ontology since the domain would remain fixed as empty. Another problem is that the model for the original ontology could leave a class interpretation as empty, while the extended model includes a subclass which by the definition of non-trivial model would have to contain an element. The simplest solution is to only require a non-trivial model in the extension where there is a non-trivial model in the original ontology.

Definition 17. [Non-trivial model conservative extension]
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be Tboxes and Γ a signature so that $\Gamma = sig(\mathcal{T}_1)$. We say that $\mathcal{T}_1 \cup \mathcal{T}_2$ is a non-trivial model conservative extension of $\mathcal{T}_1$ iff $\mathcal{T}_1 \cup \mathcal{T}_2$ is a model conservative extension of $\mathcal{T}_1$ and for every non-trivial model $\mathcal{I}$ of $\mathcal{T}_1$ there exists a non-trivial model $\mathcal{I}'$ of $\mathcal{T}_1 \cup \mathcal{T}_2$ so that $\mathcal{I} \overset{\Gamma}{=} \mathcal{I}'$.

With definition 17 it is possible to restrict all extensions on the original ontology such that there is never a case where a class is unsatisfiable.

There are some unforeseen consequences of the restrictions in a non-trivial model conservative extension. Consider an example where we introduce negation to force disjointness.

$$\mathcal{T}_1 : L \sqsubseteq RDLTemplateStatement \tag{16.1}$$
$$L \sqsubseteq =_1 r.L \sqcap \forall r.L \tag{16.2}$$
$$\mathcal{T}_2 : D \sqsubseteq RDLTemplateStatement \tag{16.3}$$
$$D \sqsubseteq =_1 m.S \sqcap \forall m.S \tag{16.4}$$
$$S \sqsubseteq \neg L \sqcup \neg D \tag{16.5}$$

L is a typical template definition. An instance of L has a single parameter r that is of type L. For this specific template r can only take L type objects, which is how other template definitions are implemented.

The extension of $\mathcal{T}_1$ adds a new concept D which is also a typical template definition take one parameter m of type S. The property m could reuse elements from r, so it is not in violation of the non-trivial model conservative extension. The concept S is however in violation of the non-trivial model conservative extension because it is disjoint with L and D. By carefully choosing a model for $\mathcal{T}_1$, this becomes rather obvious.

	$\mathcal{T}_1$
$\Delta^{\mathcal{I}}$	$\{1\}$
$topObjectProperty^{I}$	$\{<1,1>\}$
$L^{\mathcal{I}}$	$\{1\}$
$r^{\mathcal{I}}$	$\{<1,1>\}$

Table 16.1: Model of $\mathcal{T}_1$ 16.1

The interpretation of S now only has one element to choose from, 1. S could of course have an interpretation of $\emptyset$, but then it would not be a non-

trivial model conservative extension. The above example can even be simplified to accommodate this last observation.

$$\mathcal{T}_1 : L \sqsubseteq RDLTemplateStatement \tag{16.6}$$
$$\mathcal{T}_2 : D \sqsubseteq RDLTemplateStatement \tag{16.7}$$
$$S \sqsubseteq \neg L \sqcup \neg D \tag{16.8}$$

And can then be simplified into the bare essentials.

$$\mathcal{T}_1 : L \sqsubseteq RDLTemplateStatement \tag{16.9}$$
$$\mathcal{T}_2 : D \sqsubseteq \neg L \tag{16.10}$$

These are all still conservative extensions, since they are on the form of $T \sqsubseteq C$ where $T \notin sig(\mathcal{T}_1)$.

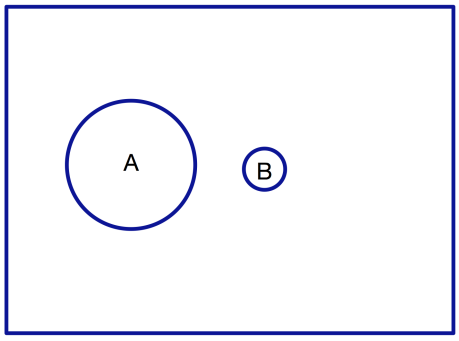

Figure 16.1: Venn diagram - Disjoint

Any extensions that creates a new class that is disjoint with all other classes will not conform to the definitions of a non-trivial conservative model extension. Let the set of all the elements in all the classes in an interpretation be A in the venn diagram 16.1. B represents the elements needed to qualify a new disjoint class. By expanding A to be equal to the domain, then there will be no space for B and it must become equivalent to $\emptyset$. For a model conservative extension this is perfectly sound, however the definition of a non-trivial model requires B to be disjoint with $\emptyset$, so it can never be a non-trivial model conservative extension. In the cases where A can be expanded to include every element in the domain, then B will become empty. If there is no single A to expand, then expanding all the As until they collaboratively force one or more of the Bs to be empty is also viable. This would be the case only if two Bs are disjoint.

It is also possible to construct a useful ontology without resorting to negation that breaks with the non-trivial model conservative extension while still being a valid model conservative extension. Such an ontology extension has to achieve

the same outcome as the above example, to force new elements into existence that can not be extracted from an existing model.

$$\mathcal{T}_1 : L \sqsubseteq RDLTemplateStatement \tag{16.11}$$
$$L \sqsubseteq =_1 r.L \sqcap \forall r.L \tag{16.12}$$
$$\mathcal{T}_2 : D \sqsubseteq RDLTemplateStatement \tag{16.13}$$
$$D \sqsubseteq =_2 m.S \sqcap \forall m.S \tag{16.14}$$

$\mathcal{T}_1$ is again a typical template definition. It subclasses L from *RDLTemplateStatement* and then requires certain attributes to be attached to L, which at this point just just a simple r attribute of type L. It could easily be of any other type, as long as it is not disjoint from L so that a model for $\mathcal{T}_1$ can contain a single element.

The extension is a less typical template definition. It requires two attributes of the same type and with the same property to be attached to any instance of template D. If there is a way to distinguish two attributes, then such an approach makes for a more precise model. As an example of this is the *FlowConnection RDLTemplateStatement*. This connects two *ContinuousFluidTransportationDevice* together so liquid can flow from one to the other. Additionally it is known which device is upstream and which device is downstream, so this is naturally expressed by having two separate properties, one for upstream and one for downstream (*hasDownstreamCFTP*). A recommendation is to have a common super-property so that a flow connection could be inferred regardless of direction of flow.

When there is no particular way of differentiating two attributes, using a cardinality restriction (with more than 1) to specify the multiple of the attribute is an alternative to inventing properties to simply denote the ordering in the template definition when this is semantically unspecified. In the example above, since the interpretation of D can always be set to the empty set, then the cardinality restriction can be ignored and the extension is a valid model conservative extension.

For a non-trivial extension the extended model must be non-trivial as well, so D must contain an element and thus S must contain two because of the cardinality restriction.

With the model for $\mathcal{T}_1$ containing only one element it is trivial to see that a non-trivial model can not exist for $\mathcal{T}_1 \cup \mathcal{T}_2$ because such a model requires a minimum of two elements in the interpretation's domain.

Looking at the venn diagram 16.2 on the next page as a representation of the above problem. A is the set that represents all the required elements to satisfy a non-trivial model of $\mathcal{T}_1$ and B represents the elements required to satisfy a non-trivial model of $\mathcal{T}_1 \cup \mathcal{T}_2$. When the domain can be reduced to the barriers of A, then B becomes equal to A and then there can not exist a non-trivial model of $\mathcal{T}_1 \cup \mathcal{T}_2$ for every non-trivial model of $\mathcal{T}_1$ because the extension requires more elements than are present in A.

	$\mathcal{T}_1$	$\mathcal{T}_1 \cup \mathcal{T}_2$
$\Delta^{\mathcal{I}}$	$\{1\}$	$\{1, ?\}$
$topObjectProperty^{\mathcal{I}}$	$\{< 1, 1 >\}$	$\{< 1, 1 >, < 1, ? >, <?, 1 ><?, ? >\}$
$RDLTemplateStatement^{\mathcal{I}}$	$\{1\}$	$\{1\}$
$L^{\mathcal{I}}$	$\{1\}$	$\{1\}$
$r^{\mathcal{I}}$	$\{< 1, 1 >\}$	$\{< 1, 1 >\}$
$D^{\mathcal{I}}$	-	$\{1\}$
$S^{\mathcal{I}}$	-	$\{1, ?\}$
$m^{\mathcal{I}}$	-	$\{< 1, 1 >, < 1, ? >\}$

Table 16.2: Model of $\mathcal{T}_1$ and $\mathcal{T}_1 \cup \mathcal{T}_2$ for 16.11

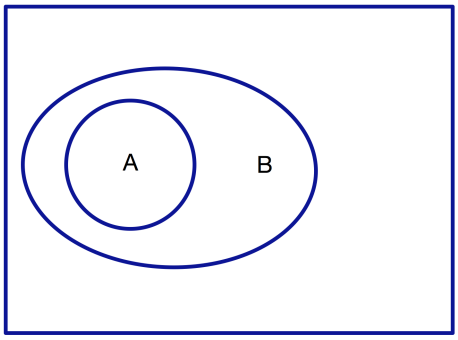

Figure 16.2: Venn diagram - subset

Every extension that requires more elements than are strictly required in the non-trivial model of the original ontology will result in a violation of the non-trivial model conservative extension. To rectify this and allow these extensions, the definition of non-trivial model conservative extensions needs to be loosened.

Categorically from the example above and from the example equation 16.9 on page 63 it is clear that two attributes are required for the models:

1. An expandable domain
2. Expandable classes and properties

A new extensions definition would require some way of making the interpretations for the domain, classes and properties expandable.

Definition 18. [Model subset] Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be Tboxes and $\mathcal{I}$ be a model for $\mathcal{T}_1$ and $\mathcal{I}'$ be a model for $\mathcal{T}_2$. $\mathcal{I} \trianglelefteq \mathcal{I}'$ iff for every atomic role and concept $r, A \in sig(\mathcal{T}_1)$ the interpretation $r^{\mathcal{I}} \subseteq r^{\mathcal{I}'}$ and $A^{\mathcal{I}} \subseteq A^{\mathcal{I}'}$ and $\Delta^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}'}$.

Then by adding this definition as a restriction to a regular model conservative extension we get the following definition.

Definition 19. [Extended non-trivial model conservative extension]

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be Tboxes. $\mathcal{T}_1 \cup \mathcal{T}_2$ is an extended non-trivial model conservative extension of $\mathcal{T}_1$ iff $\mathcal{T}_1 \cup \mathcal{T}_2$ is a model conservative extension of $\mathcal{T}_1$ and every non-trivial model I of $\mathcal{T}_1$ can be extended to a non-trivial model $\mathcal{I}'$ of $\mathcal{T}_1 \cup \mathcal{T}_2$ such that $I \trianglelefteq I'$.

16.3 Improving on model conservative extension

Definition 19 explicitly requires the extended model to be a model conservative extension of the base model as well as giving a new restriction on top of this to require subset models for non-trivial models. This second attribute is the important addition to the regular model conservative extension definition that guarantees satisfiability. One question is, what would happen if we did not require the extended model to be a model conservative extension of the original model?

Let us state a new, simpler definition for non-trivial model conservative extensions. Calling it instead, model subset extension.

Definition 20. [Model subset extension]

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be Tboxes. $\mathcal{T}_1 \cup \mathcal{T}_2$ is a model subset extension of $\mathcal{T}_1$ iff every non-trivial model $\mathcal{I}$ of $\mathcal{T}_1$ can be extended to a non-trivial model $\mathcal{I}'$ of $\mathcal{T}_1 \cup \mathcal{T}_2$ such that $\mathcal{I} \trianglelefteq \mathcal{I}'$.

The missing part of the definition is obviously a way to limit the use of trivial models. So right off the bat the definition will not hold for ontologies without a non-trivial model. Such an ontology would have to have a concept become equivalent to Nothing by definition, reasoning or a combination of the former together with single elements as members of the other concepts or roles.

If the ontology does in fact have a non-trivial model, which it should, then the extension is now restricted to models that are supersets of the former model. In comparison to regular model conservative extensions the following (table 16.3) is now allowed.

	MCE	MSE
$r, A \in sig(\mathcal{T}_1)$	$r^{\mathcal{I}} = r^{\mathcal{I}'}, A^{\mathcal{I}} = A^{\mathcal{I}'}$	$r^{\mathcal{I}} \subseteq r^{\mathcal{I}'}, A^{\mathcal{I}} \subseteq A^{\mathcal{I}'}$
$r, A \in (sig(\mathcal{T}_2) \backslash sig(\mathcal{T}_1))$	$r^{\mathcal{I}'} \subseteq \Delta^{\mathcal{I}}, A^{\mathcal{I}'} \subseteq \Delta^{\mathcal{I}}$	$r^{\mathcal{I}'}, A^{\mathcal{I}'}$

Table 16.3: Comparison of model conservative extension (MCE) and model subset extension (MSE)

Exploring the comparison table 16.3 it becomes clear that there are two possible extensions that would not be model conservative but still a model subset extension. Namely an extension that either forces roles and concepts to contain new elements when extended or forces the extension roles and concepts to require elements not in the domain.

One attempt at forcing new elements is to specify that they are required.

$$\mathcal{T}_1 : L \sqsubseteq \top \tag{16.15}$$
$$\mathcal{T}_2 : a : L \tag{16.16}$$

This seems to be a disruptive extension since the interpretation of L could be empty, and the extension would then cause it to contain at least one element. Also it is a model subset extension because every model of $\mathcal{T}_1$ can be extended by adding a new element to the domain to interpret a and adding this element to the interpretation of L this making the model a model for $\mathcal{T}_1 \cup \mathcal{T}_2$.

However, since model conservative extensions only apply to the Tbox of an ontology, the above example is actually still a model conservative extension. The concept assertion is essentially ignorable because it is in the Abox.

To create an ontology then, that is a model subset extension while not being model conservative requires us to only use the Tbox to force either interpretations of predicates in $sig(\mathcal{T}_1)$ to include new elements, or to force new elements in the domain by the interpretations of predicates in $sig(\mathcal{T}_2) \backslash sig(\mathcal{T}_1))$.

The first thing we need is to force the existence of more elements. Without resorting to the Abox, the easiest way to do this is to add a superclass to an existing class that requires a certain number of elements by a cardinality restriction.

$$\mathcal{T}_1 : L \sqsubseteq \top \tag{16.17}$$
$$\mathcal{T}_2 : L \sqsubseteq = 2\, r.M \tag{16.18}$$

Such an extension can not be model conservative, since any model of $\mathcal{T}_1$ where $L^{\mathcal{I}} := \{1\}$ would now have a domain with two elements when extended to a model for $\mathcal{T}_1 \cup \mathcal{T}_2$. In contrast, the extended model would still be a subset of the original model, so would still be model subset extension. If $\mathcal{T}_1 \cup \mathcal{T}_2$ is in fact a model subset extension, then the definition for a model subset extension is obviously not strong enough.

Proposition 21. *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are Tboxes where $\mathcal{T}_1 = \{C\}$ and $\mathcal{T}_2 = \{C \sqsubseteq = 2\, r.M\}$, then $\mathcal{T}_1 \cup \mathcal{T}_2$ is a model subset extension of $\mathcal{T}_1$.*

Proof. $\mathcal{T}_1$ and $\mathcal{T}_2$ are Tboxes and $\mathcal{T}_1 = \{C\}$ and $\mathcal{T}_2 = \{C \sqsubseteq = 2\, r.M\}$. (0)

Let $\mathcal{I}$ be an interpretation where $\mathcal{I} \models \mathcal{T}_1$. Without loss of generality, $2, 3 \notin \Delta^{\mathcal{I}}$ (1)

Define $\mathcal{I}'$ as $\mathcal{I} \trianglelefteq \mathcal{I}'$ where $\Delta^{\mathcal{I}'} := \Delta^{\mathcal{I}} \cup \{2, 3\}$, $C^{\mathcal{I}} := C^{\mathcal{I}'}$, $M^{\mathcal{I}'} := \{2, 3\}$ and for every $i \in C^{\mathcal{I}}$, $< i, 2 >$ and $< i, 3 >$ are elements in $r^{\mathcal{I}'}$. (2)

To prove that $\mathcal{T}_1 \cup \mathcal{T}_2$ is a model subset extension of $\mathcal{T}_1$, then $\mathcal{I}'$ must be a model of $\mathcal{T}_1 \cup \mathcal{T}_2$.

- $C \sqsubseteq \top$: fulfilled by $\mathcal{I}'$ with lemma 4 because $sig(\mathcal{T}_1) = \{C\}$ and $C^I := C^{I'}$
- $C \sqsubseteq = 2\, r.M$: $C^{\mathcal{I}} := C^{\mathcal{I}'}$ and $(= 2rM) = C^{\mathcal{I}} = C^{\mathcal{I}'}$ because $r^{\mathcal{I}'} = \{< i, 2 >, < i, 3 > |\ i \in C^{\mathcal{I}}\} \hookrightarrow \mathcal{I}' \models C \sqsubseteq_{=2} r.M$

□

Chapter 17

Conclusion

SPARQL queries and DL queries can be utilised to check for simple quality requirements. Checking that literals have types can be done with a SPARQL query, however because of RDF 1.1, machine-readable strings can not simply be of type *xsd:string* and must instead be of a specific type such as rdf:plainLiteral. Type checking of individuals is also possible with a SPARQL query, however there needs to be some way of limiting the scope to only instance data and the types to only vocabulary types. This can be done by adding annotations to the data.

For checking if an individual needs to belong to a subclass but is not stated to be, DL queries are the simplest approach. Every class can be queried for individuals that conjunctively belong to the subclasses and then compared to the actual listed members of the subclasses.

It is also recommended to use an Integrity Constraints Validator to check that all restrictions are fulfilled. As an alternative it is possible to generate SPARQL queries to check the constraints automatically.

The requirement for model conservative extensions is possible to achieve without any testing as long as the extending axioms are on a specific form where the left side (subclass) of a subclass expression is outside of the scope of the base ontology. The model conservative extension leaves a lot to be desired for limiting unsatisfiable concepts in the extended ontology. The new definition for an extended non-trivial model conservative extension alleviates these issues and is preferable to only using a model conservative extension. The new definition still lacks an implementation and is unfeasible to test in its current state.

Bibliography

[1] Iohn – posc caesar. Accessed: Apr 2012. [Online]. Available: https://www.posccaesar.org/wiki/IOHN

[2] Dnv arbeider for å sikre liv, verdier og miljøet. Accessed: Apr 2012. [Online]. Available: http://www.dnv.no

[3] Owl 2 web ontology language document overview (second edition). Accessed: Apr 2012. [Online]. Available: http://www.w3.org/TR/owl2-overview/

[4] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. Accessed: Mar 2013. [Online]. Available: http://www.w3.org/TR/rdf-concepts/#section-Literals

[5] F. Baader, *The description logic handbook: theory, implementation, and applications*. Cambridge: Cambridge University Press, 2003.

[6] S. Harris and A. Seaborne. Sparql 1.1 query language. Accessed: Apr 2012. [Online]. Available: http://www.w3.org/TR/sparql11-query/

[7] Object database - wikipedia. Accessed: Apr 2012. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Object_database&oldid=550812425

[8] Rethinkdb - an open-source distributed database built with love. Accessed: Apr 2012. [Online]. Available: http://www.rethinkdb.com/

[9] Manchester owl syntax - protege wiki. Accessed: Apr 2012. [Online]. Available: http://protegewiki.stanford.edu/wiki/Manchester_OWL_Syntax

[10] Iso 15926 - wikipedia, the free encyclopedia. Accessed: Apr 2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=ISO_15926&oldid=542467164

[11] Ncs drilling. Accessed: Apr 2013. [Online]. Available: http://drilling.posccaesar.org

[12] J. W. Klüwer, M. G. Skjæveland, and M. Valen-Sendstad, "Iso 15926 templates and the semantic web," in *Position paper for W3C Workshop on Semantic Web in Energy Industries; Part I: Oil and Gas*, 2008.

[13] Smart data management - cambridge semantics. Accessed: Apr 2012. [Online]. Available: http://www.cambridgesemantics.com/nb

[14] C. Golbreich and E. K. Wallace. Owl 2 web ontology language new features and rationale (second edition). Accessed: Apr 2012. [Online]. Available: http://www.w3.org/TR/owl2-new-features/#F12:_Punning

[15] J. Bao, G. Slutzki, and V. Honavar, "A semantic importing approach to knowledge reuse from multiple ontologies," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 22, no. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007, p. 1304.

[16] P. Doran, I. Palmisano, and V. Tamma, "Somet: algorithm and tool for sparql based ontology module extraction," in *Proceedings of the 2008 ESWC international workshop on ontologies: reasoning and modularity (WORM-08), Tenerife, Spain*, 2008.

[17] B. C. Grau, I. Horrocks, O. Kutz, and U. Sattler, "Will my ontologies fit together?" in *Proceedings of the 2006 International Workshop on Description Logics (DL 2006)*, 2006.

[18] B. C. Grau, I. Horrocks, Y. Kazakov, and U. Sattler, "Extracting modules from ontologies: A logic-based approach," in *Modular Ontologies*. Springer, 2009, pp. 159–186.

[19] E. Jiménez-Ruiz, R. Berlanga, B. C. Grau, T. Schneider, and U. Sattler, "Prose: A protégé plugin for reusing ontologies, safe andeconomique user manual," 2008.

[20] E. Jiménez-Ruiz, B. C. Grau, U. Sattler, T. Schneider, and R. Berlanga, "Safe and economic re-use of ontologies: A logic-based methodology and tool support," in *The Semantic Web: Research and Applications*. Springer, 2008, pp. 185–199.

[21] R. Kontchakov, F. Wolter, and M. Zakharyaschev, "Logic-based ontology comparison and module extraction, with an application to< i> dl-lite</i>," *Artificial Intelligence*, vol. 174, no. 15, pp. 1093–1141, 2010.

[22] C. Lutz, D. Walther, and F. Wolter, "Conservative extensions in expressive description logics," in *Proc. of IJCAI*, vol. 7, 2007, pp. 453–458.

[23] C. Lutz and F. Wolterinst, "Conservative extensions in the lightweight description logic\ mathcal {EL}," in *Automated Deduction–CADE-21*. Springer, 2007, pp. 84–99.

[24] N. Nikitina, "Semi-automatic verification of ontology compatibility supported by reasoning," Technical report, Institute AIFB, KIT, Karlsruhe,(February 2010), Tech. Rep.

[25] J. Santos, L. Braga, and A. G. Cohn, "Fonte: A protégé plug-in for engineering complex ontologies," in *Enterprise Information Systems*. Springer, 2011, pp. 222–236.

[26] J. Santos, L. Braga, and A. Cohn, "A protégé plugin for engineering complex ontologies by assembling modular ontologies of space, time and domain concepts."

[27] D. Peterson, S. S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson. W3c xml schema definition language (xsd) 1.1 part 2: Datatypes. Accessed: Apr 2012. [Online]. Available: http://www.w3.org/TR/xmlschema11-2/

[28] R. Cyganiak and D. Wood. Resource description framework (rdf): Concepts and abstract syntax. Accessed: Mar 2013. [Online]. Available: http://www.w3.org/TR/rdf11-concepts/

[29] The rdf vocabulary (rdf). Accessed: Mar 2013. [Online]. Available: http://www.w3.org/1999/02/22-rdf-syntax-ns

[30] Apache jena. Accessed: Apr 2013. [Online]. Available: http://jena.apache.org

[31] ellet: Owl 2 reasoner for java. Accessed: Apr 2012. [Online]. Available: http://clarkparsia.com/pellet/

[32] Wikipedia: Semantic reasoner. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Semantic_reasoner&oldid=544983328

[33] Eyeball 2.1. Accessed: Apr 2013. [Online]. Available: http://jena.sourceforge.net/Eyeball/full.html

[34] P. Hitzler, M. Krotzsch, and S. Rudolph, *Foundations of semantic web technologies*. Chapman and Hall/CRC, 2011.

[35] M. Horridge. Dlqueryexample. Accessed: Nov 2012. [Online]. Available: http://smi-protege.stanford.edu/repos/protege/icat/tags/semtech-demo/owlapi-mt/examples/src/main/java/org/coode/owlapi/examples/dlquery/DLQueryExample.java

[36] Null versus none. Accessed: Apr 2013. [Online]. Available: http://c2.com/cgi/wiki?NullVersusNone

[37] S. Koide, "Theory and implementation of object oriented semantic web language," 2010.

[38] Stardog: Documentation: Integrity constraint validation. Accessed: Apr 2013. [Online]. Available: http://stardog.com/docs/sdp/

[39] R. Morin. Demo 1a - sparqly guis. Accessed: Apr 2013. [Online]. Available: http://richmorin.github.io/SPARQLy-GUIs/recipes/demo_1a.html

[40] B. C. Grau. Owl 1.1 web ontology language tractable fragments. Accessed: Apr 2013. [Online]. Available: http://www.w3.org/Submission/owl11-tractable/

[41] S. W. Ambler, "Introduction to test driven development (tdd)," 2006.

[42] Forward chaining - wikipedia. Accessed: Apr 2012. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Forward_chaining&oldid=541154143

[43] Annotations. Accessed: Apr 2013. [Online]. Available: http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

[44] Springsource.org. Accessed: Apr 2013. [Online]. Available: http://www.springsource.org

[45] Chapter 6. aspect oriented programming with spring. Accessed: Apr 2013. [Online]. Available: http://static.springsource.org/spring/docs/2.0.x/reference/aop.html

[46] Plastic | java magic. Accessed: Apr 2013. [Online]. Available: http://tawus.wordpress.com/category/plastic/

[47] Apache tapestry home page. Accessed: Apr 2013. [Online]. Available: http://tapestry.apache.org

[48] Wikipedia: Domain-specific language. Accessed: Apr 2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=552566919

[49] Jetbrains :: Meta programming system. Accessed: Apr 2013. [Online]. Available: http://www.jetbrains.com/mps/

[50] Junit. Accessed: Apr 2013. [Online]. Available: http://junit.org

[51] elk-reasoner - a java-based owl 2 el reasoner. Accessed: Apr 2013. [Online]. Available: https://code.google.com/p/elk-reasoner/

[52] S. Staab and R. Studer, *Handbook on ontologies*. Springer, 2009.

[53] B. C. Grau, B. Motik, Z. Wu, A. Fokoue, and C. Lutz. Owl 2 web ontology language:profiles. Accessed: Apr 2013. [Online]. Available: http://www.w3.org/TR/2008/WD-owl2-profiles-20080411/

[54] D. Brickley and R. Guha. Rdf vocabulary description language 1.0: Rdf schema. Accessed: Apr 2013. [Online]. Available: http://www.w3.org/TR/rdf-schema/#ch_type

[55] Hermit reasoner: Home. Accessed: Apr 2012. [Online]. Available: http://www.hermit-reasoner.com/

[56] Locality module extractor. Accessed: Jan 2012. [Online]. Available: http://www.cs.ox.ac.uk/isg/tools/ModuleExtractor/

[57] Y. Zhou, B. C. Grau, I. Horrocks, Z. Wu, and J. Banerjee, "Making the most of your triple store: Query answering in owl 2 using an rl reasoner."

[58] Pellet integrity constraint validator. Accessed: Apr 2013. [Online]. Available: http://clarkparsia.com/pellet/icv

[59] Asm - home page. Accessed: Apr 2013. [Online]. Available: http://asm.ow2.org

[60] J. Pathak, T. M. Johnson, and C. G. Chute, "Survey of modular ontology techniques and their applications in the biomedical domain," *Integrated computer-aided engineering*, vol. 16, no. 3, pp. 225–242, 2009.

[61] F. Baader, "What's new in description logics," *Informatik-Spektrum*, vol. 34, no. 5, pp. 434–442, 2011.

Part V

Appendix

Quality Criteria for RDF representations of installations descriptions according to ISO15926 part 8

Martin Giese, Håvard Mikkelsen Ottestad

criteria.tex 2492 2013-04-30 17:08:30Z martingi

1 Prerequisites

We assume the following:

- All data is originally represented as a set of "template instances" and "type assertions"
  - A template instance is an n-ary literal of the shape
    $$p(i_1, \ldots, i_n)$$
    where $i_1, \ldots, i_n$ are literals (strings, numbers, etc) or resources identifiers (URIs), and p is a template (also identified by a URI)
  - A type assertion is a literal
    $$C(i)$$
    where C is an OWL class and i is a resource identifier as before

Do we allow blank nodes in template instances?

- For every template p, there is a description giving an RDF property $R_{p,i}$ for each of the arguments of p.
- Any template instance $p(i_1, \ldots, i_n)$ is represented in RDF as a set of $n+1$ triples

  _:x `rdf:type` p;
  _:x $R_{p,1}$ i_1;
  ...
  _:x $R_{p,n}$ i_n.
- Any type assertion $C(i)$ is represented in RDF by a triple

  i `rdf:type` C
- These representations are complemented by an ontology, including the ISO15925 part x/y/z vocabularies, and the template and meta-template ontologies, as well as a domain-specific ontology built on top of these.

2 Requirements for RDF representations in general

2.1 Separation of Information Levels

Requirement 2.1.1 *There is a clear separation of the set of triples into*

- *a vocabulary description, and*
- *instance data.*

This separation might be effected through the storage in different files, in different graphs underlying a SPARQL endpoint, etc. In any case, it must be possible to say which part a given triple belongs to.

Counterexample:

Mixed Ontology and Instance Data - Turtle:

```
:TemperatureSensor  rdfs:subClassOf :Sensor.
:tempSensor1 :minTemp "275.0"^^xsd:float;
:LightSensor  rdfs:subClassOf :Sensor.
```

Example:

Ontology - Turtle:

```
:TemperatureSensor  rdfs:subClassOf :Sensor.
:LightSensor  rdfs:subClassOf :Sensor.
```

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;
```

Rationale: The main purpose of this separation is to provide a basis for the remaining criteria. The rules applying to the vocabulary definition and the instance data will be different.

Implementation: This is not a requirement that can be tested. Rather, implementations testing the other requirements will need this separation as part of their input.

Note that information both of the vocabulary description and the instance data may well be spread over several documents or data sources. The important point is that it must be clear for each triple which part it belongs to.

Requirement 2.1.2 *There is a clear separation of the vocabulary definition into*

- *an application specific vocabulary*
- *a generic vocabulary*

For the meaning of 'separation', see Req. 2.1.1.

Counterexample:

Ontology - Turtle:

```
:Sensor  rdfs:subClassOf :PhysicalObject.
:TMP36  rdfs:subClassOf :Sensor.
```

Example:

Generic Ontology - Turtle:

```
:Sensor  rdfs:subClassOf :PhysicalObject.
```

Domain Ontology - Turtle:

```
:TMP36  rdfs:subClassOf :Sensor.
```

Rationale: Like Req. 2.1.1, this is mainly to provide a basis for the remaining criteria. For instance, resources will be required to carry a sufficiently specific type, in the sense that types mentioned in the application specific vocabulary and not the generic vocabulary are considered specific enough.

Implementation: See Req. 2.1.1. Again, each part of the vocabulary may well be spread over several documents or data sources. The important point is that it must be clear for each triple (RDFS/OWL axiom) which part it belongs to.

Definition 2.1.3 *An* application class *is a class that is mentioned in the application specific vocabulary, but not in the generic vocabulary. Any class mentioned in the generic vocabulary is called a* generic class.

2.2 Literals

Requirement 2.2.1 *Any literal should be either typed (by one of the standard datatypes defined by OWL 2) or carry a language tag.*

Counterexample:

Instance Data - Turtle:

```
  :tempSensor1 :minTemp "275.0";
  rdfs:label "This is a temperature sensor".
```

Example:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;  #float number
  rdfs:label "This is a temperature sensor"@en.   #english lang
```

Rationale: Any literal is either intended for machine processing or to be read by humans. In the first case, it should carry a datatype delimiting the interpretation (the datatype might by xsd:string if that is the intention). In the latter case, it is a natural language string, which should carry its language tag.

Implementation: This should be easy to implement using SPARQL queries with suitable filter expressions.

2.3 Types

Requirement 2.3.1 *Any resource mentioned should have a type, either explicitly given with* `rdf:type`*, or inferable, that is related to the application domain. This type should in other words be an* application class *in the sense of Def. 2.1.3.*

Counterexample:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;
```

Example:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;

#Either#
:tempSensor1 rdf:type :AboveFreezingTemperatureSensor. #explicit
    type
```

Ontology - Manchester:

```
#Or#
Class: AboveFreezingTemperatureSensor
    EquivalentTo: minTemp some double[> 273.15]
```

Rationale: All objects referred to the description of an installation should have to do directly with that installation. They must therefore have a more concrete type than being a "Thing", a "possible individual", etc. Any kind of further processing of the RDF description will require more concrete types.

Note that this requirement does not require types to be given explicitly. Types maybe inferred from an ontology and the relations in which a resource stands with other resources, e.g. domain and range reasoning.

Since it cannot be decided from the data alone what constitutes a sufficiently specific type, the separation of the vocabulary according to Req. 2.1.1 is used to decide which parts of the vocabulary are to be counted as the domain vocabulary, and which are generic terms.

Implementation: This requirement is not easy to implement, since it requires reasoning. However, if the model and accompanying ontology can be classified, the available types for all individuals can easily be checked.

The previous requirement is comparatively weak, and hard to check, but it can be seen as a minimal requirement. We can ask for more by restricting the reasoning to simple cases.

Requirement 2.3.2 *Requirement 2.3.1 is refined by requiring that a application class can be derived as type of any resource by some simple reasoning regime, e.g. RDFS entailment*

Counterexample:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;
```

Example:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float.
```

Ontology - Turtle:

```
:minTemp rdf:type rdfs:domain :TemperatureSensor . #inferrable
    type
```

Rationale: This requirement is easy to check algorithmically, although it cannot be checked with a SPARQL query.

Implementation: RDFS reasoning (domain/range, subclass, subproperty) can be used to check for the existence of types. In fact, reasoning can be aborted for each individual, as soon as at least one type has been derived.

The latter requirement can be seen as one element in a family of requirements, indexed by the reasoning regime to be supported. At the extreme end, we can require that no reasoning is needed at all:

Requirement 2.3.3 *Requirement 2.3.1 is refined by requiring at least one application class to be explicitly given as type with an* `rdf:type` *triple for every resource.*

Counterexample:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;
#inferrable type, but at least one must be listed
```

Ontology - Turtle:

```
:minTemp rdfs:domain :TemperatureSensor . #inferrable type
```

Example:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;
  rdf:type :AboveFreezingTemperatureSensor.
```

Ontology - Turtle:

```
:minTemp rdfs:domain :TemperatureSensor . #inferrable type
```

Rationale: No reasoning is needed to check this requirement. And it can be implemented without adding too much overhead to the RDF representation.

Implementation: The presence of `rdf:type` triples can probably be checked with a SPARQL query.

To ease further processing, we can add the following requirement:

Requirement 2.3.4 *Requirement 2.3.3 is refined by requiring that* all *(named) types that can be inferred for a resource are explicitly given by* `rdf:type` *triples.*

Counterexample:

Instance Data - Turtle:
```
:tempSensor1 :minTemp "275.0"^^xsd:float;
  rdf:type :AboveFreezingTemperatureSensor.
```

Ontology - Turtle:
```
:minTemp rdfs:domain :TemperatureSensor . #inferrable type
```

Example:

Instance Data - Turtle:
```
:tempSensor1 :minTemp "275.0"^^xsd:float;
  rdf:type :AboveFreezingTemperatureSensor, :TemperatureSensor.
```

Ontology - Turtle:
```
:minTemp rdfs:domain :TemperatureSensor . #inferrable type
```

Rationale: This makes it easy for software that processes the RDF representation to find all instances of any given type, without requiring reasoning.

Implementation: Checking this requires reasoning again. The model needs to be classified, and for each individual i and each type C it belongs to, the presence of a triple i `rdf:type` C needs to be checked.

A certain kind of "useless" type information allowed by the previous requirements can be eliminated by the following one:

Requirement 2.3.5 *If a resource i has (can be inferred to have) type C and C is (can be inferred to be) covered by some of its sublcasses $D_1, \ldots, D_n \sqsubseteq C$, with $C \sqsubseteq D_1 \sqcup \cdots \sqcup D_n$, then there is a $k \in \{1, \ldots, n\}$ such that i has (can be inferred to have) type D_k.*

Counterexample:

Instance Data - Turtle:
```
:tempSensor1 :minTemp "275.0"^^xsd:float;
  rdf:type :Sensor.
```

Ontology - Manchester:

```
Class: TemperatureSensor
    SubClassOf: Sensor

Class: LightSensor
    SubClassOf: Sensor

Class: Sensor
    SubClassOf: LightSensor or TemperatureSensor
```

Example:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;
  rdf:type :Sensor, :TemperatureSensor.
```

Ontology - Manchester:

```
Class: TemperatureSensor
    SubClassOf: Sensor

Class: LightSensor
    SubClassOf: Sensor

Class: Sensor
    SubClassOf: LightSensor or TemperatureSensor
```

Rationale: C plays the role of an "abstract superclass": any element of C must also belong to one of the subclasses $D_1, \ldots D_n$. We require the model to say explicitly which of the subclasses this is. For instance, if "sensor" is covered by "temperature sensor", "pressure sensor", and "motion sensor", then the model should say for any concrete resource with type "sensor" which of the subclasses it belongs to.

Note: it is not clear whether this is always a desired property. For a given application, "sensor" may sufficiently concrete, and giving the actual type of sensor would not add any value. On the other hand, the ontology used to define sensors might include covering axioms for types of sensors. In such a case, this requirement should not be enforced. It only makes sense if the "granularity" of the ontology is not finer than required by the application.

Implementation: This requires reasoning in general. Given an ontology, reasoning could be used to find all sets of covering axioms as asked for in this requirement, and then generate a single SPARQL query that checks the requirement, provided type information is represented explicitly. E.g. given the sensor ontology, a SPARQL query could be written that checks that whenever a triple i `rdf:type :Sensor` is present, there is also a triple i `rdf:type :TemperatureSensor` or i `rdf:type :PressureSensor` or i `rdf:type :MotionSensor`.

The previous requirements require types of resources to be *inferable*, and require at least one of them to belong to the domain vocabulary. This is not the same as the following mostly orthogonal property:

Requirement 2.3.6 *Requirement 2.3.1 is refined by requiring that all (named) types any resource* belongs to *are explicitly given or can be inferred.*

Rationale: What this means is that anything that is e.g. a temperature sensor actually carries that type, explicitly or implicitly. For the previous requirements, it would be enough if a temperature sensor belonged to the supertype "sensor", assuming that "sensor" is part of the domain vocabulary.

Implementation: This cannot be checked mechanically, since it requires knowing what identifiers refer to.

2.4 Consistency

Requirement 2.4.1 *The RDF representation and its accompanying ontology must be consistent.*

Counterexample:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;
             :minTemp "100.5"^^xsd:float.
```

Ontology - Manchester:

```
Class :TemperatureSensor
  SubClassOf: :minTemp exactly 1 double
```

Rationale: An inconsistent model is obviously faulty.

Implementation: The implementation requires full OWL DL reasoning.

If the model is too large to be classified, this requirement could be weakened be restricting the reasoning performed, for instance as follows:

Requirement 2.4.2 *The ontology TBox must be consistent, and no individual is asserted (or can be inferred by RDFS reasoning) to belong to two classes that are known to be disjoint from TBox reasoning.*

Counterexample:

Instance Data - Turtle:

```
:tempSensor1 :minTemp "275.0"^^xsd:float;
rdf:type :TemperatureSensor,
   :LightSensor.
```

Ontology - Turtle:

```
:TemperatureSensor owl:disjointWith :LightSensor.
```

Rationale: For large ABoxes with reasonably sized TBoxes, this might be tractable, and catch most potential mistakes.

Note that it is hard to give a semantic explanation for such a restriction. If the model as a whole is inconsistent, then it has no interpretation, and no amount of restricted reasoning can change that.

3 Requirements for RDF representations in accordance with ISO15926 part 8

3.1 Conservative Extensions

Requirement 3.1.1 *The application specific vocabulary must be a conservative extension of the underlying ISO15926 part x/y/z, template, and meta-template vocabularies.*

Rationale: This ensures that the application specific ontology does not "modify" the meaning of the underlying ontologies.

Implementation: This requires reasoning, and that reasoning is non-standard. Not sure which implementations exist at all.

This is a nice minimal requirement, but might be hard to check.

It might be possible to relax this, by restricting the way in which the domain vocabulary can refer to the imported ontologies, e.g. only by subclassing. Such restrictions can easily become over-restrictive however.

3.2 Part 8 adherence

We want to ensure that all information is actually represented as "reified" template statements.

Requirement 3.2.1 *For any triple x y z belonging to the instance data, where y is different from* `rdf:type`*, and y is not declared by the vocabulary description to be an* `owl:AnnotationProperty`*, there is a (possibly inferred) triple x* `rdf:type` *p where p is in turn of type ... (what? abc:Template?)*

Rationale: We want only template instances and type assertions in the RDF data. However, annotations are allowed, like for instance `rdfs:label` to give a human readable identifier to a resource.

Note: in combination with Req. 2.3.1, this means that the template types p should also be part of the application specific vocabulary.

Implementation: This can be checked with SPARQL queries generated from the domain vocabulary, provided that the types are explicitly represented. Maybe with a static SPARQL query.

Requirement 3.2.2 *For any resource or blank node x mentioned in the instance data, such that x `rdf:type` p for some n-ary template p with declared properties $R_{p,i}$, there is a triple x $R_{p,i}$ y_i for all $i = 1, \ldots, n$, either explicitly or implicitly, for some literal or named resource y_i.*

Rationale: Whenever a template instance is given, all arguments should be given. Not just inferred to exist.

TODO: Have to think about blank nodes for the y_i. It might make sense to have blank nodes in the template instances. But then, such blank nodes have to be kept apart from things that can be inferred to exist simply from the template axioms.

Implementation: This can be checked with SPARQL queries generated from the domain vocabulary, provided that the types are explicitly represented. Maybe with a static SPARQL query.

4 Further Thoughts

1. Many of the previous requirements can be read the "semantic" way, i.e. as requirements on inferable information, or as requirements on explicitly given information. Which to choose depends on how much reasoning we are willing to perform. In any case, it would be good to find a structure for this document which allows referring to requirements together with the implied amount of reasoning without reduplicating all the requirements.

2. We have several recurring categories of implementability:
   a) undecidable
   b) algorithm not known to us
   c) algorithm known but non-standard
   d) implemented by standard reasoning tools
   e) implementable as a set of SPARQL queries that can be computed from the domain ontology
   f) implementable as a static SPARQL query, independent of the domain vocabulary.

   Should make reference to these categories more systematic.

3. At some point, we might want to look into an axiomatisation of properties of templates, like symmetry, transitivity, etc. which may not be easy to express after the part 8-isation.

5 TODO

- Criteria for URIs, prefixes/namespaces
- No relative URIs?
- Running Example
- Positive/negative examples for each criterion