LOD2 Deliverable D3.4.1: Report on Relevant Automatically Detectable Modelling Errors and Problems

Lorenz Bühmann, Szymon Danielczyk, Jens Lehmann

Abstract: This report documents a survey about modelling errors and problems in semantic web knowledge bases. It identifies different types of errors which can typically occur during the creation and lifecycle of knowledge bases like OWL ontologies or interlinked data. Additionally, an overview about existing tool support is given. This will show which tool covers which kinds of errors/problems. From this overview, we conclude with requirements for the ontology repair and enrichment (ORE) tool in LOD2.

Collaborative Project

LOD2 - Creating Knowledge out of Interlinked Data

Project Number: 257943 Start Date of Project: 01/09/2010 Duration: 48 months

Deliverable 3.4.1
Report on Relevant Automatically Detectable Modelling Errors and Problems

Dissemination Level	Public
Due Date of Deliverable	Month 6, 28/2/2011
Actual Submission Date	02/3/2011
Work Package	WP3, Knowledge Base Creation, Enrichment and Repair
Task	Task T3.4
Type	Report
Approval Status	Approved
Version	1.0
Number of Pages	67
Filename	deliverable-3.4.1.pdf

Abstract: This is a survey on modelling errors and problems in semantic web knowledge bases. It identifies different types of errors as well as methods and tools to detect and remove those errors.

Project funded by the European Commission within the Seventh Framework Programme (2007 – 2013)

History

Version	Date	Reason	Revised by
0.1	2010-12-01	Initial version	Lorenz Bühmann
0.2	2011-01-04	Added preliminaries section	Jens Lehmann
0.3	2011-01-24	Added semantic errors section	Lorenz Bühmann
0.4	2011-01-29	Added possible reasons of modeling problems	Lorenz Bühmann
0.5	2011-01-30	Added section about reasoning performance problems	Lorenz Bühmann
0.6	2011-01-30	Added structural errors section	Lorenz Bühmann
0.7	2011-02-01	Added section for existing tools	Lorenz Bühmann
0.8	2011-02-03	Added section about problems in Linked Data	Szymon Danielczyk
0.9	2011-02-14	Added description about inspector to tools section	Szymon Danielczyk
0.9.1	2011-02-14	Added syntactic errors section	Szymon Danielczyk
1.0	2011-02-27	Final version after internal review	Jens Lehmann

Author list

Organisation	Name	Contact Information
ULEI	Lorenz Bühmann	buehmann@informatik.uni-leipzig.de
NUIG	Szymon Danielczyk	szymon.danielczyk@deri.org
ULEI	Jens Lehmann	lehmann@informatik.uni-leipzig.de

Executive summary

This report documents a survey about modelling errors and problems in semantic web knowledge bases. It identifies different types of errors which can typically occur during the creation and lifecycle of knowledge bases like OWL ontologies or interlinked data. Additionally, an overview about existing tool support is given. This will show which tool covers which kinds of errors/problems. From this overview, we conclude with requirements for the ontology repair and enrichment (ORE) tool in LOD2.

Contents

1 Introduction **7**

2 Preliminaries **9**
2.1 Ontology 9
2.2 Description Logics 9
2.3 OWL 17

3 Ontology errors **20**
3.1 Syntactic errors 22
3.1.1 Parsing and Syntax 22
3.2 Semantic errors 26
3.3 Structural errors 31
3.4 Reasoning performance problems 37
3.5 Problems in Linked Data 43

4 Tool Support **55**
4.1 Tools related to syntactic errors 55
4.2 Tools related to semantic errors 55
4.3 Tools related to structural errors 58
4.4 Tools related to performance errors 60
4.5 Tools related to problems in Linked Data 60

5 Conclusion **62**

List of Figures

3.1 The differences/relationships between incoherency and inconsistency. 27
3.2 Classification of the taxonomy errors 33

List of Tables

2.1 $\mathcal{SHOIN}$ Syntax and Semantics 13
2.2 Mapping between OWL and DL 18

4.1 Overview about reasoning systems, the algorithm they use and the supported expressivity. 57
4.2 Feature Overview of Various Tools. 59

Chapter 1

Introduction

The number of data sets published in the Semantic Web has seen a steep rise over the past years. While the quantity of data is continuously increasing, its quality is still considered problematic. For instance, in a position paper [19], it has been claimed that "Linked Data is merely more data". One particular problem are syntactic, structural and semantic errors in Semantic Web knowledge bases, which are the scope of this survey. While it is widely acknowledged that completely resolving those problems is impossible given the size, heterogeneity and decentralised nature of the Web of Data, reducing the frequency of errors would enable an easier use of existing knowledge. In particular, modelling errors and other problems can prevent reasoning over data, integrating data or even accessing data.

Those modelling errors and other problems in semantic data can occur on different levels. In this report, we first distinguish between the most common error types: syntactic and semantic errors. Syntactic errors are mainly violations of conventions of the language in which the ontology is modelled, e.g. the validity of XML, whereas semantic errors are considered in this report as contradictions in the underlying formal logics. Another type of problem we consider are structural errors. By this, we basically mean problems in the taxonomy, like for example circularities. Beyond this distinction there are two additional more task-focused types we will analyse: (a) tools and methods which allow to detect problems which negatively affect the performance of reasoning over expressive knowledge bases and (b) problems which are the specific to publishing RDF using the Linked Data principles. In the following sections we will discuss those errors in detail.

We begin with a preliminaries part (Chap. 2) to provide a basis for the rest of the report. In the subsequent Chapter 3, we will describe the different types of problems and errors that can occur during the life-cycle of ontological knowledge bases. In particular, the chapter contains sections about syntactic

errors (Sec. 3.1), semantic errors (Sec. 3.2), errors in the taxonomy (Sec. 3.3), problems which can decrease the performance of tableau-based reasoners (Sec. 3.4) and problems which can be found in the context of Linked Data (Sec. 3.5) After that, in Chapter 4 we will give an overview of existing tools which have been developed to detect some of these errors.

Chapter 2

Preliminaries

2.1 Ontology

The word "ontology" is used with different meanings in different communities. The most radical difference is perhaps between the philosophical sense and the computational sense. The first one stands for a branch of philosophy which deals with the nature and structure of "reality". In the second case, which emerged in the recent years in the knowledge engineering community and reflects the most prevalent use in Computer Science, an ontology can be understood as a "formal, explicit specification of a shared conzeptualization"(Studer et al.[34]). A *conceptualization* can informal be seen as an abstract, simplified view of the world that we wish to represent for some purpose, containing the objects, concepts, and other entities that are assumed to exist and the relationship among them. *Formal* refers to the fact that the expressions must be machine readable, hence natural language is excluded.

2.2 Description Logics

In this section, we introduce description logics including their syntax and semantics.

Description logics is the name of a family of knowledge representation (KR) formalisms. They emerged from earlier KR formalisms like semantic networks and frames. Their origin lies in the work of Brachman on structured inheritance networks [7]. Since then, description logics have enjoyed increasing popularity. They can essentially be understood as fragments of first-order predicate logic. They have less expressive power, but usually decidable inference problems and a user-friendly variable free syntax.

Description logics represent knowledge in terms of *objects*, *concepts*, and

roles. Concepts formally describe notions in an application domain, e.g. one could define the concept of being a father as "a man having a child" (`Father` $\equiv$ `Man` $\sqcap \exists$`hasChild`.$\top$ in DL notation). Objects are members of concepts in the application domain and roles are binary relations between objects. Objects correspond to constants, concepts to unary predicates, and roles to binary predicates in first-order logic.

In description logic systems information is stored in a *knowledge base*. It is sometimes divided in two parts: *TBox* and *ABox*. The ABox contains *assertions* about objects. It relates objects to concepts and other objects via roles. The TBox describes the *terminology* by relating concepts and roles. For some expressive description logics this clear separation does not exist. Furthermore, the notion of an *RBox*, which contains knowledge about roles, is sometimes used in expressive description logics. We will usually consider those axioms as part of the TBox in this report.

As mentioned before, DLs are a family of KR formalisms. We use the terms *description language* and *description logic* synonymously for one particular element of this family. First, we introduce the $\mathcal{ALC}$ description logic as an example language. $\mathcal{ALC}$ is a proper fragment of OWL [18] and is generally considered to be a prototypical description logic for research investigations. $\mathcal{ALC}$ stands for *attributive language with complement*. It allows to construct complex concepts from simpler ones using various language constructs. The next definition shows how such concepts can be built.

Definition 2.2.1 (Syntax of $\mathcal{ALC}$ concepts)
Let N_R be a set of *role names* and N_C be a set of *concept names* ($N_R \cap N_C = \emptyset$). The elements of N_C are also called *atomic concepts*. The set of $\mathcal{ALC}$ concepts is inductively defined as follows:

1. Each atomic concept is an $\mathcal{ALC}$ concept.
2. If C and D are $\mathcal{ALC}$ concepts and $r \in N_R$ a role, then the following are also $\mathcal{ALC}$ concepts:
   - $\top$ (top), $\bot$ (bottom)
   - $C \sqcup D$ (disjunction), $C \sqcap D$ (conjunction), $\neg C$ (negation)
   - $\forall r.C$ (value/universal restriction), $\exists r.C$ (existential restriction) $\square$

Example 2.2.2 ($\mathcal{ALC}$ concepts)
Some examples of complex concepts in $\mathcal{ALC}$ are:

- $\mathit{Man} \sqcap \exists \mathit{hasChild}.\top$

- $Man \sqcap \exists hasChild.(Rich \sqcup Beautiful)$
- $Man \sqcap \exists hasChild.\neg Adult$
- $Man \sqcap \exists hasChild.\forall hasFriend.ComputerScientist$

Other description languages are usually named according to the expressive features they support. The choice of language is usually a tradeoff between expressivity and complexity of reasoning. The description logic navigator[1] provides detailed information about the complexity of a particular language. The following is a list of commonly used letters in the description logic naming scheme along with their meaning (note that if one feature can be expressed using other ones the letter is usually omitted in the language name).

$\boxed{\mathcal{S}}$ $\mathcal{ALC}$ + transitivity: For a transitive role r, we have that $r(a, b)$ and $r(b, c)$ implies $r(a, c)$.

$\boxed{\mathcal{H}}$ subroles: $r \sqsubseteq s$ says that r is a subrole of s, i.e. $r(a, b)$ implies $s(a, b)$.

$\boxed{\mathcal{I}}$ inverse roles: r^- denotes the inverse role of r, i.e. $r^-1(a, b)$ iff $r(b, a)$.

$\boxed{\mathcal{O}}$ nominals: Sets of objects can be used to construct concepts, e.g. {MONICA} denotes the singleton set, which only contains MONICA. Nominals are useful in cases where the instances of a concept should be enumerated, e.g. the members of the European Union.

$\boxed{\mathcal{N}}$ number restrictions: Allows constructs of the form $\geq n\ r$ and $\leq n\ r$ to build concepts. This is useful if one wants to define a concept like "mother of at least three children" (Woman $\sqcap \geq 3$ hasChild).

$\boxed{\mathcal{Q}}$ qualified number restrictions: Concept constructors of the form $\geq n\ r.C$ and $\leq n\ r.C$ can be used. If C is the top concept, this is equivalent to unqualified number restrictions. This is useful to define a concept like "mother of at least three male children" (Woman $\sqcap \geq$ 3 hasChild.Male).

$\boxed{\mathcal{F}}$ functional roles: Allows to express that a role r is functional, i.e. has at most one filler, which is equivalent to the axiom $\top \sqsubseteq \leq 1\ r$.

[1] http://www.cs.manchester.ac.uk/~ezolin/dl/

$\boxed{\mathcal{R}}$ complex role inclusions: Axioms of the form $r \circ s \sqsubseteq r$ (or $r \circ s \sqsubseteq s$) state that when $r(a, b)$ and $s(b, c)$ holds, then $r(a, c)$ (or $s(a, c)$) also holds. For instance, we could use the axiom $\texttt{locatedIn} \circ \texttt{partof} \sqsubseteq \texttt{locatedIn}$ to model the part of relationship for locations. Now, if we know that Leipzig is located in Saxony and Saxony is part of Germany, we can infer that Leipzig is located in Germany.

$\boxed{\mathcal{E}}$ existential quantification: If there exists an instance b with $r(a, b)$ and b is instance of C, then a is instance of $\exists r.C$.

$\boxed{\text{D}}$ data types: Data types are used to incorporate different kinds of data, e.g. numbers or strings. This allows, for instance, to define the concept of an old person as a person of age 65 or higher.

While $\mathcal{ALC}$ is seen as a prototypical language and foundation for more expressive languages, there has also been a lot of research effort for simple languages with often tractable inference problems. Two of those languages, which are referred to within the deliverable are $\mathcal{AL}$ and $\mathcal{EL}$:

$\mathcal{AL}$ is inductively defined as follows: $\top$, $\bot$, $\exists r.\top$, A, $\neg A$ with $A \in N_C$, $r \in N_R$ are $\mathcal{AL}$ concepts. If C and D are $\mathcal{AL}$ concepts, then $C \sqcap D$ is an $\mathcal{AL}$ concept. If C is an $\mathcal{AL}$ concept and r a role, then $\forall r.C$ is an $\mathcal{AL}$ concept.

$\mathcal{EL}$ is inductively defined as follows: $\top$, A with $A \in N_C$ are $\mathcal{EL}$ concepts. If C and D are $\mathcal{EL}$ concepts and $r \in N_R$, then $C \sqcap D$ and $\exists r.C$ are $\mathcal{EL}$ concepts.

The semantics of concepts is defined by means of interpretations. See the following definition and Table 2.1 listing common concept constructors.

Definition 2.2.3 (Interpretation)
An *interpretation* $\mathcal{I}$ consists of a non-empty *interpretation domain* $\Delta^{\mathcal{I}}$ and an *interpretation function* $\cdot^{\mathcal{I}}$, which assigns to each $A \in N_C$ a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to each $r \in N_R$ a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. □

Example 2.2.4 (Interpreting Concepts)
Let the interpretation $\mathcal{I}$ be given by:

$$\Delta^{\mathcal{I}} = \{\mathit{MONICA}, \mathit{JESSICA}, \mathit{STEPHEN}\}$$
$$\mathit{Woman}^{\mathcal{I}} = \{\mathit{MONICA}, \mathit{JESSICA}\}$$
$$\mathit{hasChild}^{\mathcal{I}} = \{(\mathit{MONICA}, \mathit{STEPHEN}), (\mathit{STEPHEN}, \mathit{JESSICA})\}$$

We then have:

$$(\mathit{Woman} \sqcap \exists \mathit{hasChild}.\top)^{\mathcal{I}} = \{\mathit{MONICA}\}$$

construct	syntax	semantics
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
role	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
nominals	$\{o\}$	$\{o\}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, \lvert\{o\}\rvert^{\mathcal{I}} = 1$
top concept	$\top$	$\Delta^{\mathcal{I}}$
bottom concept	$\bot$	$\emptyset$
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
exists restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{a \mid \exists b.(a,b) \in r^{\mathcal{I}} \text{ and } b \in C^{\mathcal{I}}\}$
value restriction	$\forall r.C$	$(\forall r.C)^{\mathcal{I}} = \{a \mid \forall b.(a,b) \in r^{\mathcal{I}} \text{ implies } b \in C^{\mathcal{I}}\}$
atleast restriction	$\geq n\ r.C$	$(\geq n\ r)^{\mathcal{I}} = \{a \mid \lvert(\{b \mid (a,b) \in r^{\mathcal{I}}\}\rvert \geq n\}$
atmost restriction	$\leq n\ r.C$	$(\leq n\ r)^{\mathcal{I}} = \{a \mid \lvert(\{b \mid (a,b) \in r^{\mathcal{I}}\}\rvert \leq n\}$

Table 2.1: Syntax and semantics for concepts in $\mathcal{SHOIN}$.

In the most general case, *terminological axioms* are of the form $C \sqsubseteq D$ or $C \equiv D$, where C and D are (complex) concepts. The former axioms are called *inclusions* and the latter *equivalences*. An equivalence whose left hand side is an atomic concept is a *concept definition*. In some languages with low expressivity, like $\mathcal{AL}$, terminological axioms are restricted to definitions. We can define the semantics of terminological axioms in a straightforward way. An interpretation $\mathcal{I}$ satisfies an inclusion $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and it satisfies the equivalence $C \equiv D$ if $C^{\mathcal{I}} = D^{\mathcal{I}}$. $\mathcal{I}$ satisfies a set of terminological axioms iff it satisfies all axioms in the set. An interpretation, which satisfies a (set of) terminological axiom(s) is called a *model* of this (set of) axiom(s). Two (sets of) axioms are *equivalent* if they have the same models. A finite set $\mathcal{T}$ of terminological axioms is called a *(general) TBox*. Let N_I be the set of object names (disjoint with N_R and N_C). An *assertion* has the form $C(a)$ (*concept assertion*), $r(a,b)$ (*role assertion*), where a, b are object names, C is a concept, and r is a role. An *ABox* $\mathcal{A}$ is a finite set of assertions.

Objects are also called individuals. To allow interpreting ABoxes we extend the definition of an interpretation. In addition to mapping concepts to subsets of our domain and roles to binary relations, an interpretation has to assign to each individual name $a \in N_I$ an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. An interpretation $\mathcal{I}$ is a model of an ABox $\mathcal{A}$ (written $\mathcal{I} \models \mathcal{A}$) iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all $C(a) \in \mathcal{A}$ and $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ for all $r(a,b) \in \mathcal{A}$. An interpretation $\mathcal{I}$ is a model of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ (written $\mathcal{I} \models \mathcal{K}$) iff it is a model of $\mathcal{T}$ and $\mathcal{A}$.

Example 2.2.5 (Models of a Knowledge Base)
Let the knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be given by:

TBox $\mathcal{T}$*:*

$$\begin{aligned} \texttt{Man} &\equiv \neg\texttt{Woman} \sqcap \texttt{Person} \\ \texttt{Woman} &\sqsubseteq \texttt{Person} \\ \texttt{Mother} &\equiv \texttt{Woman} \sqcap \exists \texttt{hasChild}.\top \end{aligned}$$

ABox $\mathcal{A}$*:*

$$\begin{aligned} &\texttt{Man}(\texttt{STEPHEN}). \\ &\neg\texttt{Man}(\texttt{MONICA}). \\ &\texttt{Woman}(\texttt{JESSICA}). \\ &\texttt{hasChild}(\texttt{STEPHEN}, \texttt{JESSICA}). \end{aligned}$$

We will now look at some interpretations and determine whether or not they are a model of $\mathcal{K}$*. For all interpretations, the domain* $\{\texttt{MONICA}, \texttt{JESSICA}, \texttt{STEPHEN}\}$ *is used and all object names are interpreted in the obvious way (*STEPHEN *is interpreted as* STEPHEN *etc.).*

Let the interpretation $\mathcal{I}_1$ *be given by:*

$$\begin{aligned} \texttt{Man}^{\mathcal{I}_1} &= \{\texttt{JESSICA}, \texttt{STEPHEN}\} \\ \texttt{Woman}^{\mathcal{I}_1} &= \{\texttt{MONICA}, \texttt{JESSICA}\} \\ \texttt{Mother}^{\mathcal{I}_1} &= \emptyset \\ \texttt{Person}^{\mathcal{I}_1} &= \{\texttt{JESSICA}, \texttt{MONICA}, \texttt{STEPHEN}\} \\ \texttt{hasChild}^{\mathcal{I}_1} &= \{(\texttt{STEPHEN}, \texttt{JESSICA})\} \end{aligned}$$

Clearly this does not satisfy $\mathcal{T}$*, because the definition* $\texttt{Man} \equiv \neg\texttt{Woman} \sqcap \texttt{Person}$ *is not satisfied. We have* $\texttt{Man}^{\mathcal{I}_1} = \{\texttt{JESSICA}, \texttt{STEPHEN}\}$ *and* $(\neg\texttt{Woman} \sqcap \texttt{Person})^{\mathcal{I}_1} = \{\texttt{STEPHEN}\}$*, which are not equal. However,* $\mathcal{I}_1$ *satisfies* $\mathcal{A}$*.*

Let the interpretation $\mathcal{I}_2$ *be given by:*

$$\begin{aligned} \texttt{Man}^{\mathcal{I}_2} &= \{\texttt{STEPHEN}\} \\ \texttt{Woman}^{\mathcal{I}_2} &= \{\texttt{JESSICA}, \texttt{MONICA}\} \\ \texttt{Mother}^{\mathcal{I}_2} &= \emptyset \\ \texttt{Person}^{\mathcal{I}_2} &= \{\texttt{JESSICA}, \texttt{MONICA}, \texttt{STEPHEN}\} \\ \texttt{hasChild}^{\mathcal{I}_2} &= \emptyset \end{aligned}$$

$\mathcal{I}_2$ *satisfies* $\mathcal{T}$*, but not* $\mathcal{A}$*. We have* $\texttt{hasChild}(\texttt{STEPHEN}, \texttt{JESSICA}) \in \mathcal{A}$*, but* $(\texttt{STEPHEN}^{\mathcal{I}_2}, \texttt{JESSICA}^{\mathcal{I}_2}) \notin \texttt{hasChild}^{\mathcal{I}_2}$*.*

Let the interpretation $\mathcal{I}_3$ be given by:

$$\begin{aligned}
\texttt{Man}^{\mathcal{I}_3} &= \{\texttt{STEPHEN}\} \\
\texttt{Woman}^{\mathcal{I}_3} &= \{\texttt{JESSICA}, \texttt{MONICA}\} \\
\texttt{Mother}^{\mathcal{I}_3} &= \{\texttt{MONICA}\} \\
\texttt{Person}^{\mathcal{I}_3} &= \{\texttt{JESSICA}, \texttt{MONICA}, \texttt{STEPHEN}\} \\
\texttt{hasChild}^{\mathcal{I}_3} &= \{(\texttt{MONICA}, \texttt{STEPHEN}), (\texttt{STEPHEN}, \texttt{JESSICA})\}
\end{aligned}$$

$\mathcal{I}_3$ is a model of $\mathcal{T}$ and $\mathcal{A}$, so it is a model of $\mathcal{K}$. One may argue that nothing in our knowledge base justifies the fact that we interpret `MONICA` *as mother. However, in DLs we usually have the* open world assumption*. This means that the given knowledge is viewed as incomplete. There is nothing, which tells us that* `MONICA` *is not a mother. In databases one usually uses the* closed world assumption*, i.e. all facts, which are not explicitly stored, are assumed to be false.*

As we have described, a knowledge base can be used to represent the information we have about an application domain. Besides this *explicit* knowledge, we can also deduce *implicit* knowledge from a knowledge base. It is the aim of *inference algorithms* to extract such implicit knowledge. There are some standard reasoning tasks in description logics, which we will briefly describe.

In *terminological reasoning* we reason about concepts. The standard problems are *consistency*, *satisfiability* and *subsumption*. Intuitively, consistency checks detect whether a knowledge base contains contradictions. Satisfiability determines whether a concept can be satisfied, i.e. it is free of contradictions. Subsumption of two concepts detects whether one of the concepts is more general than the other.

Definition 2.2.6 (Consistency)
A knowledge base $\mathcal{K}$ is *consistent* iff it has a model. □

Example 2.2.7 (Consistency)
The knowledge base $\mathcal{K} = \{A_1 \equiv A_2 \sqcap \neg A_2, A_1(a)\}$ is not consistent, since A_1 is equivalent to $\bot$ and has an asserted instance a.

Definition 2.2.8 (Satisfiability)
Let C be a concept and $\mathcal{T}$ a TBox. C is *satisfiable* iff there is an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$. C is *satisfiable with respect to* $\mathcal{T}$ iff there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$. □

Example 2.2.9 (Satisfiability)
$\texttt{Man} \sqcap \texttt{Woman}$ is satisfiable. However, it is not satisfiable with respect to the TBox in Example 2.2.5.

Definition 2.2.10 (Subsumption, Equivalence)
Let C, D be concepts and $\mathcal{T}$ a TBox. *C is subsumed by D*, denoted by $C \sqsubseteq D$, iff for any model $\mathcal{I}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. *$C$ is subsumed by D with respect to $\mathcal{T}$*, denoted by $C \sqsubseteq_{\mathcal{T}} D$, iff for any model $\mathcal{I}$ of $\mathcal{T}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

C is equivalent to D (with respect to $\mathcal{T}$), denoted by $C \equiv D$ ($C \equiv_{\mathcal{T}} D$), iff $C \sqsubseteq D$ ($C \sqsubseteq_{\mathcal{T}} D$) and $D \sqsubseteq C$ ($D \sqsubseteq_{\mathcal{T}} C$).

C is strictly subsumed by D (with respect to $\mathcal{T}$), denoted by $C \sqsubset D$ ($C \sqsubset_{\mathcal{T}} D$), iff $C \sqsubseteq D$ ($C \sqsubseteq_{\mathcal{T}} D$) and not $C \equiv D$ ($C \equiv_{\mathcal{T}} D$). □

Example 2.2.11 (Subsumption)
$\texttt{Mother}$ is not subsumed by $\texttt{Woman}$. However, $\texttt{Mother}$ is subsumed by $\texttt{Woman}$ with respect to the TBox in Example 2.2.5.

Subsumption allows to build a hierarchy of atomic concepts, commonly called the *subsumption hierarchy*. Analogously, for more expressive description logics *role hierarchies* can be inferred.

In *assertional reasoning* one reasons about objects. As one relevant task for learning in DLs, the *instance check problem* is to find out whether an object is an instance of a concept, i.e. belongs to it. A *retrieval* operation finds all instances of a given concept.

Definition 2.2.12 (Instance Check)
Let $\mathcal{A}$ be an ABox, $\mathcal{T}$ a TBox, $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a knowledge base, C a concept, and $a \in N_I$ an object. *a is an instance of C with respect to $\mathcal{A}$*, denoted by $\mathcal{A} \models C(a)$, iff in any model $\mathcal{I}$ of $\mathcal{A}$ we have $a^{\mathcal{I}} \in C^{\mathcal{I}}$. *$a$ is an instance of C with respect to $\mathcal{K}$*, denoted by $\mathcal{K} \models C(a)$, iff in any model $\mathcal{I}$ of $\mathcal{K}$ we have $a^{\mathcal{I}} \in C^{\mathcal{I}}$.

To denote that a is not an instance of C with respect to $\mathcal{A}$ ($\mathcal{K}$) we write $\mathcal{A} \not\models C(a)$ ($\mathcal{K} \not\models C(a)$). □

We use the same notation for sets S of assertions of the form $C(a)$, e.g. $\mathcal{K} \models S$ means that every element in S follows from $\mathcal{K}$.

Definition 2.2.13 (Retrieval)
Let $\mathcal{A}$ be an ABox, $\mathcal{T}$ a TBox, $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a knowledge base, C a concept. The *retrieval $R_{\mathcal{A}}(C)$ of a concept C with respect to $\mathcal{A}$* is the set of all instances of C: $R_{\mathcal{A}}(C) = \{a \mid a \in N_I \text{ and } \mathcal{A} \models C(a)\}$. Similarly the *retrieval $R_{\mathcal{A}}(C)$ of a concept C with respect to $\mathcal{K}$* is $R_{\mathcal{K}}(C) = \{a \mid a \in N_I \text{ and } \mathcal{K} \models C(a)\}$. □

Example 2.2.14 (Instance Check, Retrieval)
In Example 2.2.5 we have $R_{\mathcal{K}}(\texttt{Woman}) = \{\texttt{JESSICA}, \texttt{MONICA}\}$. JESSICA *and* MONICA *are instances of* Woman*, because in any model* $\mathcal{I}$ *of* $\mathcal{K}$ *we have* $\texttt{JESSICA}^{\mathcal{I}} \in \texttt{Woman}^{\mathcal{I}}$ *and* $\texttt{MONICA}^{\mathcal{I}} \in \texttt{Woman}^{\mathcal{I}}$.

For more detailed information about description logics, we refer the interested reader to [17, 5, 14] .

2.3 OWL

After we have introduced description logics, we will now describe their relationship to OWL (Web Ontology Language). In essence OWL is based on description logics extended by several features to make it suitable as a web ontology language, e.g. using URIs/IRIs as identifiers, imports of other ontologies etc. By basing OWL-DL on description logics, it can make use of the theory developed for DLs, in particular sophisticated reasoning algorithms.

In OWL, different naming conventions are used compared to description logics. OWL *classes* correspond to concepts in description logics and *properties* correspond to roles.

OWL comes in three flavors: OWL Lite, OWL DL, and OWL Full. OWL Lite corresponds to $\mathcal{SHIF}(D)$ and OWL DL to $\mathcal{SHOIN}(D)$. OWL Full contains features not expressible in description logics, but needed to be compatible with RDFS, i.e. OWL Full can be seen as the union of RDFS and OWL DL.

The latest version OWL 2 is again split in two flavors OWL 2 DL and OWL 2 Full. OWL 2 DL corresponds to the logic $\mathcal{SROIQ}(D)$, whereas the full variant is again introduced for RDFS compatibility. In addition, three profiles were introduced: EL, QL, and RL. Each profile imposes, usually syntactical, restrictions on OWL in order to allow more efficient reasoning. OWL 2 EL is aimed at applications which require expressive property modelling and is based on the logic $\mathcal{EL}^{++}$, which guarantees polynomial reasoning time wrt. ontology size for all standard inference problems. QL is targeted at applications with massive volumes of instance data. In QL, query answering can be implemented on top of conventional relational database systems and sound and complete conjunctive query answering methods can be implemented in LOGSPACE. As in the EL profile, the standard inference problems run in polynomial time. RL is aimed at scalable applications, which however, do not want to sacrifice too much expressive power. Reasoning algorithms for it can be implemented in rule-based engines and run in polynomial time. The EL and QL languages are subsets of OWL 2 DL, whereas RL provides

OWL expression / axiom	DL syntax	Manchester syntax
Thing	$\top$	Thing
Nothing	$\bot$	Nothing
intersectionOf	$C_1 \sqcap \cdots \sqcap C_n$	C_1 and ... and C_n
unionOf	$C_1 \sqcup \cdots \sqcup C_n$	C_1 or ... or C_n
complementOf	$\neg C$	not C
oneOf	$\{x_1\} \sqcup \cdots \sqcup \{x_n\}$	$\{x_1, \ldots, x_n\}$
allValuesFrom	$\forall r.C$	r only C
someValuesFrom	$\exists r.C$	r some C
maxCardinality	$\leq n\ r$	r max n
minCardinality	$\geq n\ r$	r min n
cardinality	$\leq n\ r \sqcap \geq n\ r$	r exact n
subClassOf	$C_1 \sqsubseteq C_2$	C_1 SubClassOf: C_2
equivalentClass	$C_1 \equiv C_2$	C_1 EquivalentTo: C_2
disjointWith	$C_1 \equiv \neg C_2$	C_1 DisjointWith: C_2
sameAs	$\{x_1\} \equiv \{x_2\}$	x_1 SameAs: x_2
differentFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	x_1 DifferentFrom: x_2
domain	$\forall r.\top \sqsubseteq C$	r Domain: C
range	$\top \sqsubseteq \forall r.C$	r Range: C
subPropertyOf	$r_1 \sqsubseteq r_2$	r_1 SubPropertyOf: r_2
equivalentProperty	$r_1 \equiv r_2$	r_1 EquivalentTo: r_2
inverseOf	$r_1 \equiv r_2^-$	r_1 InverseOf: r_2
TransitiveProperty	$r^+ \sqsubseteq r$	r Characteristics: Transitive
FunctionalProperty	$\top \sqsubseteq \leq 1\ r$	r Characteristics: Functional

Table 2.2: OWL constructs in DL and Manchester OWL syntax (excerpt).

two variants where one is subset of OWL 2 Full and the other one is a subset of OWL 2 DL.

In general, OWL offers more convenience constructs than the corresponding description logics, but does not extend its expressivity. It should be noted that OWL does not make the unique name assumption, so different individuals can be mapped to the same domain element. It allows to express equality and inequality between individuals ($a = b$, $a \neq b$) using `owl:sameAs` and `owl:differentFrom`. Most algorithms for description logics already supported this before the OWL specification was created. Not making the unique names assumption is crucial in the Semantic Web, where it is often the case that many knowledge bases contain information about the same entity. In this case, a common approach is that each knowledge base uses their own URI and `owl:sameAs` is used to connect them.

Table 2.2 shows for some examples how constructs in OWL can be mapped to description logics. We can see that some features can be mapped directly to description logics, e.g. union, and others are syntactic sugar, e.g. functional properties.

OWL also has different syntactic formats, in which a knowledge base can be stored. Since it can be converted to RDF, formats like RDF/XML or Turtle can be used. There is also a special XML syntax called OWL/XML and the Manchester OWL Syntax. The latter one is popular in ontology editors. Examples are shown on the right column in Table 2.2.

Chapter 3

Ontology errors

There are several points which can be seen as a possible reason for modelling errors and problems:

Difficulty in understanding modelling: Due to the fact that OWL is based on an expressive DL one of the main causes for errors, especially semantic errors, is the difficulty that comes from modelling accurately in an expressive and complex ontology language. OWL users and developers are not likely to have a lot of experience with description logic based KR, and without adequate tool support for training and explanation, engineering ontologies can be a hard task for such users. As ontologies become larger and more complex, highly non-local interactions in the ontology (e.g., interaction between local class restrictions on properties and its global domain/range restrictions) make modeling, and analysing the effects of modeling non-trivial even for domain experts.

Interlinking of OWL Ontologies: The idea behind Web ontology development is different from traditional and more controlled ontology engineering approaches which rely on high investment, relatively large, heavily engineered, mostly monolithic ontologies. For OWL ontologies, which are based on the Web architecture (characterized as being open, distributed and scalable), the emphasis is more on utilizing this freeform nature of the Web to develop and share (preferably smaller) ontology models in a relatively ad hoc manner, allowing ontological data to be reused easily, either by linking models (using the numerous mapping properties available in OWL) or merging them (using the `owl:imports` command). However, when related domain ontologies created by separate parties are merged using `owl:imports`, the combination can result in modeling errors. This could be due to ontology

authors either having different views of the world, following alternate design paradigms, or simply, using a conflicting choice of modeling constructs. An example is when the two upper-level ontologies, CYC and SUMO are merged leading to a large number of unsatisfiable concepts due to disjointness statements present in CYC [32].

Migration to OWL: Since OWL is a relatively new standard, one can expect that existing schema/ontologies in languages pre-dating OWL such as XML, DAML, KIF etc. will be migrated to OWL, either manually or using automated translation scripts. A faulty migration process can lead to an incorrect specification of concepts or individuals in the resultant OWL version. For example, the OWL version of the Tambis ontology seen earlier contains 144 unsatisfiable classes (out of 395) due to an error in the transformation script used in the conversion process.

Ontology evolution is the timely adaptation of an ontology to changed business requirements, to trends in ontology instances and patterns of usage of the ontology-based application, as well as the consistent management/propagation of these changes to dependent elements. A modification in one part of the ontology may generate subtle inconsistencies in other parts of the same ontology, in the ontology-based instances as well as in depending ontologies and applications [25]. This variety of causes and consequences of the ontology changes makes ontology evolution a very complex operation that should be considered as both, an organizational and a technical process.

Modelling errors and problems in ontologies can roughly be divided into three levels: The first one are so called syntactic errors, which especially include violations of conventions of the language in which the ontology is modelled, e.g. the validity of XML. Another type of modelling problems are semantic errors, here we in particular think of inconsistency and incoherency. The last level summarises structural problems, where in general taxonomy errors are subsumed by. Beyond this distinction there are two additional more task-focused types: (a) if somebody is interested in getting inferences on expressive knowledge bases one could also ask for problems which can negatively affect the performance of reasoners and (b) if data should be provided as Linked Data there are also specific kinds of errors. In the following sections we will discuss the three levels and the additional types in a more detailed way.

3.1 Syntactic errors

Syntactic issues loom large in OWL for a number of reasons including the baroque exchange syntax, RDF/XML and the use of URIs/IRIs (and their abbreviations), but most of these are straightforward to detect and rectify using XML parsers and RDF validators. However, for OWL DL, there is yet another layer of syntactic structure on top of the corresponding RDF graph, i.e., a number of restrictions are imposed on the form of the graph in order for it to count as an instance of the OWL DL "species". These restrictions are quite onerous for authors and easy to violate as, in general, importing is not species safe: importing an OWL Lite document into another may result in an OWL Full document, and an OWL DL document importing either an OWL Lite or OWL DL document may become OWL Full. Even OWL Full, the superset of the rest, may become OWL DL or Lite upon an import. The WebOnt working group defined a category of OWL processor for so-called species validation, and though there were serious fears of the complexity and implementation of such validation, several implementations have emerged and appear to be reliable.

Invalid RDF/XML documents; usually caused by simple errors such as unescaped special characters, misuse of RDF/XML shortcuts, and omission of namespace. Again, such issues are relatively rare, presumably due to use of mature RDF/XML APIs for producing data and the popularity of the W3C RDF/XML validation service [1]

Unfortunatelly, incorrect use of datatypes is relatively common in the Web of Data. Firstly, datatype literals can be malformed: i.e., ill-typed literals which do not abide by the lexical syntax for their respective datatype. [15]

3.1.1 Parsing and Syntax[2]

RDF/XML and RDFa: Ambiguous Base-URI

Just like in HTML, in certain RDF syntaxes use of relative URIs is allowed. This allows use of abbreviated names in the document which will be appended onto the base URI: usually determined as the URL from which the document is retrieved. Although XML (and thus RDF/XML and RDFa) allows specification of an unambiguous base URI, oftentimes, such a base URI is unspecified.

[1] `http://www.w3.org/RDF/Validator/`

[2] This section contains material edited and adapted from from the work done in the Pedantic Web Group - `http://pedantic-web.org/fops.html` with the permissions of the authors. Please also refer to [15] when mentioning this content.

If we consider that a document can be retrieved from two different locations; e.g., http://example.org/doc.rdf and http://www.example.org/doc.rdf. This document uses relative URIs but doesn't explicitly specify a base URI. Now, an agent which accesses the document from both locations will resolve the relative URIs against different base URIs, with different resulting URIs. The agent will see the same resource—when identified by a relative URI—as two different resources with distinct URIs (one version with, and one version without the www.).

Thus, unless one is sure that his base URI is unambiguous or he does not use relative URIs, we encourage use of the xml:base construct to explicitly specify the base URI, and ultimately avoid confusion.

One other word of warning about base URIs: depending on the combination of the base URI and the relative URI being resolved against it, a parser may unexpectedly strip part of the base URI to create what it deems to be the intended full URI. For example:

- "http://example.org/dangling/" + "name" = "http://example.org/dangling/name"
- "http://example.org/dangling" + "name" = "http://example.org/name"
- "http://example.org/dangling" + "" = "http://example.org/dangling"
- "http://example.org/dangling" + "/name" = "http://example.org/name"
- "http://example.org/dangling/" + "/name" = "http://example.org/name"
- "http://example.org/dangling#" + "name" = "http://example.org/name"
- "http://example.org/dangling" + "#name" = "http://example.org/dangling#name"
- "http://example.org/dangling#" + "#name" = "http://example.org/dangling#name"
- "http://example.org/dangling#" + "/name" = "http://example.org/name"

The moral of the story here is to be careful if using relative URIs and always:

- ensure that base URI is unambiguous
- and double-check that the URIs resolve as expected.
- if using RDF/XML, be wary of the fact that rdf:ID relative names have a different means of being resolved against base URIs...

RDF/XML: rdf:ID/rdf:nodeID/rdf:about/rdf:resource

In RDF/XML, there are four constructs for identifying things: rdf:ID, rdf:nodeID, rdf:about and rdf:resource. Jumbling them up is surprisingly easy and can result in a document which although valid, represents something completely different from what one intended. We now briefly clarify the intended use of the four constructs, and then discuss some common mistakes and confusion:

- **rdf:about** : Used solely as an attribute on a "node element" to uniquely identify a resource by means of a URI. The URI can be specified in full, or as a relative URI which will be resolved against the in-scope base-URI.
- **rdf:resource** : Used solely as an attribute on a "property element" to specify a URI value for an object. Similarly to rdf:about, the URI may be given in full, or as a relative URI which will be resolved against the in-scope base URI.
- **rdf:ID** : Used as an attribute to provide unique relative XML names which will be appended onto the base URI. When used on a node element, rdf:ID="xmlname" acts roughly like rdf:about="#xmlname"; however, rdf:ID values must be unique names and must be valid XML names. Can also be used on a "property element" to identify a reified statement (valid, but rare usage).
- **rdf:nodeID** : When used on a "subject element", acts similarly to rdf:ID and provides unique names which are used to create blank-nodes instead of URIs. When used in the "property position", and allows for specifying blank-node objects.

Problems mainly arise when rdf:ID is mistakenly used instead of rdf:about, rdf:nodeID or rdf:resource; or indeed, vice-versa. Firstly, on node elements, and unlike rdf:about, rdf:ID values have a '#' prepended. Secondly, when used on node elements, rdf:ID creates URIs and rdf:nodeID creates blank nodes. Thirdly, when used on property elements, rdf:ID (unlike rdf:nodeID) identifies a reified statement, and not the object of the property—to identify an object URI, rdf:resource should be used.

Again, even though a validator may give one document the thumbs up, this is only an indication that the document can be parsed into triples, not necessarily that the document parses into the triples that one intended and with the names that one intended. One should also verify that the parsed triples are as expected, and that any relative URIs resolve as expected.

Incompatibility with Range Datatype

Properties can have a defined range which states that a value of a specyfic property (object of a triple with that property in the predicate position) must be of a certain class. Not only can a range be a class of individuals (e.g., the knows property has range Person), but it can also be a datatype class (e.g., the lastModified property has range dateTime).

To understand how problems arise, we need to look a bit deeper into the interpretation of datatypes and datatype literals. Firstly, it is important to note that the class of plain literals without language tags (literals without a datatype or language tag) can be considered equivalent to the datatype class xsd:string. Secondly, a literal cannot have both a datatype and a language tag (if you try to give a literal both a language tag and a datatype in RDF/XML, the language tag will most often be ignored). Thirdly, there are two types of XML Schema datatypes: primitive datatypes and derived datatypes where the latter are defined in terms of (derived from) a parent datatype; all derived datatypes have exactly one primitive datatype ancestor and a member of a derived datatype is also considered a member of all ancestor datatypes—to give an example [3] , nonPositiveInteger is a datatype derived from integer, which is in turn derived from decimal; a member of nonPositiveInteger is also a member of integer and decimal. Finally, all of the primitive XML Schema datatypes are disjoint from each other; this means that a literal cannot be a member of more than one primitive datatype (or, as it follows, of derived datatypes with different primitive datatype ancestors).

What is more, remember that properties can have datatypes defined as range. Now, everytime one use that property, one must ensure that he gives a value whose datatype is compatible (not disjoint) with the defined range. One common misconception is that if the range of a property; e.g., lastModified; is a certain datatype; e.g., xsd:dateTime; and a plain literal value is given for that property; e.g.,

`"2002-10-10T12:00:00Z"` ; then that plain literal will be converted into a typed literal; e.g., `"2002-10-10T12:00:00Z"^^<xsd:datetime>` This is not so, and is in fact an inconsistency since the plain literal value is considered analogously to an xsd:string which is disjoint with the property's range xsd:dateTime.

The safest option to avoid such confusion is fairly straightforward: if the range of a property that one is using is a datatype, specifically type each value for that property using that exact datatype, and ensure that the value abides by the lexical form of that datatype; e.g., every time one use lastModified, one should specify that the value is an xsd:dateTime, and ensure that the

[3] `http://www.w3.org/TR/xmlschema-2/#built-in-datatypes`

value is a lexically valid xsd:dateTime.

3.2 Semantic errors

Given a syntactically correct OWL ontology, semantic resp. logical defects are those which can be detected by an OWL reasoner. These include unsatisfiable classes and inconsistent ontologies.

Unsatisfiable classes are those which cannot be true of any possible individual, that is, they are equivalent to the empty set (or, in description logic terms, to the bottom concept, or, in OWL lingo, to `owl:Nothing`). For example, class A is unsatisfiable if it is a subclass of both, class C and $\neg C$, since it implies a direct contradiction. If the ontology contains at least one unsatisfiable class, sometimes people speak of an incoherent ontology. Unsatisfiable concepts are usually a fundamental design error, as they cannot be used to characterize any individual. Unsatisfiable concepts are also quite easy to detect for a reasoner and for a tool to display. However, determining why a concept in an ontology is unsatisfiable can be a considerable challenge even for experts in the formalism and in the domain, even for modestly sized ontologies. The problem worsens significantly as the number and complexity of axioms of the ontology grows.

Inconsistent ontologies are those which have a contradiction in the instance data, e.g., an instance of an unsatisfiable class. They are also fairly easy for a reasoner to detect, if it can process the ontology at all. In fact, in tableau reasoners, unsatisfiability testing is reduced to a consistency test by positing that there is a member of the to be tested class and doing a consistency check on the resultant knowledge base (KB). However, unlike with mere unsatisfiable classes, an inconsistent ontology is, on the face of it, very difficult for a reasoner to do further work with. Since anything at all follows from a contradiction, no other results from the reasoner (e.g., with regard to the subsumption hierarchy) are useful.

Not always but sometimes incoherency and inconsistency are related in some way, as we can see in Fig. 3.2. The ontology in (a) is incoherent because there exists an unsatisfiable class C, which is subclass of the two disjoint classes A and B, but is consistent because the only two existing instances a and b are not asserted to C. If we consider the knowledge base in example (b) we can see that it is incoherent because of the same reason as in (a), but this time is also inconsistent as a is asserted to the unsatisfiable concept C. Examples (c) and (d) are also inconsistent, but this time the error is not rooted by an unsatisfiable class, but because instance a is asserted to A and $\neg A$ (example (c)) resp. the disjoint classes A and B (example (d)).

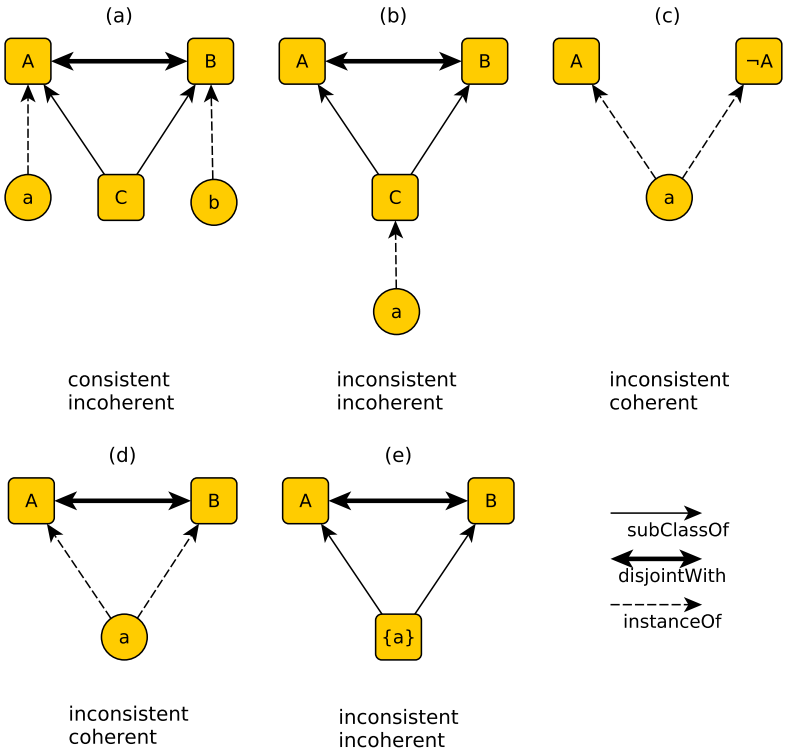

Figure 3.1: The differences/relationships between incoherency and inconsistency.

In both cases the ontology itself is coherent. In the case of (e), we haven an inconsistent and incoherent knowledge base, because the class existing exactly of the instance a is subclass of the disjoint classes A and B.

Common error patterns Although there are theoretically indefinitely many ways in which inconsistencies may arise, in [37] they have found empirically that most can be boiled down to a small number of "error patterns":

1. The inconsistence is from some local definition:

   (a) Having both a class and its complement class as super conditions.

   Example 3.2.1

   $$MeatyVegetable \sqsubseteq Vegetable$$
   $$MeatyVegetable \sqsubseteq \neg Vegetable$$

   (b) Having both universal and existential restrictions that act along the same property, whilst the filler classes are disjoint.

Example 3.2.2

$$VegetarianPizza \sqsubseteq \forall hasTopping.Vegetable$$
$$VegetarianPizza \sqsubseteq \exists hasTopping.Meat$$
$$Vegetable \sqcap Meat \sqsubseteq \bot$$

(c) Having a super condition that is asserted to be disjoint with `owl:Thing`.

Example 3.2.3

$$Pizza \sqsubseteq \neg \top$$

(d) Having a super condition that is an existential restriction that has a filler which is disjoint with the range of the restricted property.

Example 3.2.4

$$IceCreamPizza \sqsubseteq \exists hasTopppping.IceCream$$
$$Range\,(hasTopping) = PizzaTopping$$
$$IceCream \sqcap PizzaTopping \sqsubseteq \bot$$

(e) Having an universal restriction with `owl:Nothing` as the filler and a must existing restriction along property relationships.

Example 3.2.5

$$Bread \sqsubseteq \forall hasTopppping.\bot$$
$$Bread \sqsubseteq \exists hasTopppping.Meat$$

(f) Having super conditions of n existential restrictions that act along a given property with disjoint fillers, whilst there is a super condition that imposes a maximum cardinality restriction or equality cardinality restriction along the property whose cardinality is less than n.

Example 3.2.6

$$BoringPizza \sqsubseteq \leq 1 hasTopping.\top$$
$$BoringPizza \sqsubseteq \exists hasTopping.Meat$$
$$BoringPizza \sqsubseteq \exists hasTopping.Vegetable$$
$$Meat \sqcap Vegetable \sqsubseteq \bot$$

(g) Having super conditions containing conflicting cardinality restrictions.

Example 3.2.7

$$BoringFancyPizza \sqsubseteq < 2 hasTopping.\top$$
$$BoringFancyPizza \sqsubseteq > 2 hasTopping.\top$$

2. The inconsistence is propagated from other source:

(a) Having a super condition that is an existential restriction that has an inconsistent filler.

Example 3.2.8
(Assumption: $MeatyVegetable$ is inconsistent)

$$MeatyVegetablePizza \sqsubseteq \exists hasTopping.MeatyVegetable$$

(b) Having a super condition that is a hasValue restriction that has an individual that is asserted to be a member of an inconsistent class.

Example 3.2.9
(Assumption: $MeatyVegetable$ is inconsistent)

$$MeatyVegetablePizza \sqsubseteq \exists hasTopping.aMeatyVegetable$$
$$aMeatyVegetable \in MeatyVegetable$$

Explaining logical errors Finding and understanding these undesired entailments can be a difficult or impossible task without tool support. Even in ontologies with a small number of logical axioms, there can be several, non-trivial causes.

Therefore, interest in finding explanations for such entailments has increased in recent years. One of the most usual kinds of explanations are *justifications* [21]. A justification for an entailment is a minimal subset of axioms with respect to a given ontology, that is sufficient for the entailment to hold. More formally, let $\mathcal{O}$ be a given ontology with $\mathcal{O} \models \eta$, then $\mathcal{J}$ is a justification for η if $\mathcal{J} \models \eta$, and for all $\mathcal{J}' \subset \mathcal{J}$, $\mathcal{J}' \not\models \eta$. In the meantime, there is support for the detection of potentially overlapping justifications in tools like Protégé[4] and Swoop[5]. Justifications allow the user to focus on a small subset of the ontology for fixing a problem. However, even such a subset can be complex, which has spurred interest in computing *fine-grained* justifications [16] (in contrast to *regular* justifications). In particular, *laconic justifications* are those where the axioms do not contain superfluous parts and are as weak as possible. A subset of laconic justifications are *precise justifications*, which split larger axioms into several smaller axioms allowing minimally invasive repair.

A possible approach to increase the efficiency of computing justifications is module extraction [11]. Let $\mathcal{O}$ be an ontology and $\mathcal{O}' \subseteq \mathcal{O}$ a subset of axioms of $\mathcal{O}$. $\mathcal{O}'$ is a module for an axiom α with respect to $\mathcal{O}$ if: $\mathcal{O}' \models \alpha$ iff $\mathcal{O} \models \alpha$. $\mathcal{O}'$ is a module for a signature $\mathbf{S}$ if for every axiom α with $Sig(\alpha) \subseteq \mathbf{S}$, we have that $\mathcal{O}'$ is a module for α with respect to $\mathcal{O}$. Intuitively, a module is an ontology fragment, which contains all relevant information in the ontology with respect to a given signature. One possibility to extract such a module is syntactic locality [11]. [35] showed that such *locality-based modules* contain all justifications with respect to an entailment and can provide order-of-magnitude performance improvements.

For a single entailment, e.g. an unsatisfiable class, there can be many justifications. Moreover, in real ontologies, there can be several unsatisfiable classes or several reasons for inconsistency. While the approaches described above work well for small ontologies, they are not feasible if a high number of justifications or large justifications have to be computed. Due to the relations between entities in an ontology, several problems can be intertwined and are difficult to separate.

One approach [24] for handling the first problem mentioned above is to separate between root and derived unsatisfiable classes. A derived unsatisfiable class has a justification, which is a proper super set of a justification of another unsatisfiable class. Intuitively, their unsatisfiability may depend on other unsatisfiable classes in the ontology, so it can be beneficial to fix those root problems first. There are two different approaches for determining such

[4] `http://protege.stanford.edu`

[5] `http://www.mindswap.org/2004/SWOOP/`

classes: The first approach is to compute all justifications for each unsatisfiable class and then apply the definition. The second approach relies on a structural analysis of axioms and heuristics. While the first approach is computationally very expensive for larger ontologies, the second one suffers from incompleteness as it is sound, but incomplete, i.e. not all class dependencies are found, but the found ones are correct. To increase the proportion of found dependencies, the TBox can be modified in a way which preserves the subsumption hierarchy to a large extent. It was shown in [24] that this allows to draw further entailments and improves the pure syntactical analysis.

Given a justification, the problem needs to be resolved by the user, which involves the deletion or modification of axioms in it. For supporting the user by handling many justification with possible many axioms, ranking methods, which highlight the most probable causes for problems, are important. Common methods (see [22] for details) are frequency (How often does the axiom appear in justifications?), syntactic relevance (How deeply rooted is an axiom in the ontology?) and semantic relevance (How many entailments are lost or added?[6]).

3.3 Structural errors

These are defects that are not necessarily invalid, syntactically or semantically, yet are discrepancies in the KB or unanticipated results of modeling, which require the modelers' attention before use in a specific domain or application scenario. Consider the following cases:

- There may be unintended inferences (subsumption, realization relationships, etc.) discovered by the reasoner. For example, it can be inferred that "parents of at least three children" is a subclass of "parents with at least two children", even if there is no explicit assertion of that relationship. Though the reasoner can detect and report subsumptions such as this, it cannot distinguish between desirable (non)inferences and undesired ones.

- Missing type declarations can occur in a KB, such as if a resource is used in a particular manner that entails it to be of a particular type, but is not explicitly declared to be so, e.g., given the axiom `hasParent` (`John`, `Mary`) where `hasParent` is known to be an `owl:ObjectProperty`, one can infer that `John` and `Mary` both have to be of type `owl:Individual`.

[6] Since the number of entailed axioms can be infinite, it is recommended to restrict the search for that entailments to a subset of axioms as suggested in [22].

In such cases, the reasoner will infer the corresponding entailment, but the absence of this explicit information could be considered as a defect.

- In some cases, redundancies may exist in the KB, such as when an asserted axiom is entailed by another set of axioms from the KB. Here, depending on whether the redundancy is desired or not, the case could be considered as a defect.

- There may be cases of unused atomic classes or properties with no references anywhere in the KB (i.e., the term is not explicitly used in any axiom in the KB), which can be considered as extraneous data.

To make it clearer which problems and errors are considered in this chapter, we will describe a classification which was suggested in [33]. Therein the focus is aimed on the evaluation of taxonomic knowledge, regarding other perspectives likes completeness or redundancy.

This approach is based on 3 criteria which can be considered while evaluating an ontology:

Consistency is declared as there exists no possibility to get contradictory conclusions from a given set of valid definitions.

Completeness means that all which is supposed to be in the ontology is explicitly stated in it, or can be inferred.

Conciseness refers to whether in the ontology exist (a) no unnecessary or useless informations, (b) no redundancy between explicit definitions and (c) no inferred redundancies.

Taken into account these criteria, the taxonomy errors are divided into the three areas *Inconsistency*, *Incompleteness* and *Redundancy*, which are then further subdivided as we will describe in Sec. 3.2.

Inconsistency

There are mainly three types of errors that cause inconsistency and ambiguity in the ontology. These are Circulatory errors, Partition errors and Semantic inconsistency errors.

Circularity errors

They occur when a class is defined as a subclass or superclass of itself at any level of hierarchy in the ontology. They can occur with distance 0, 1

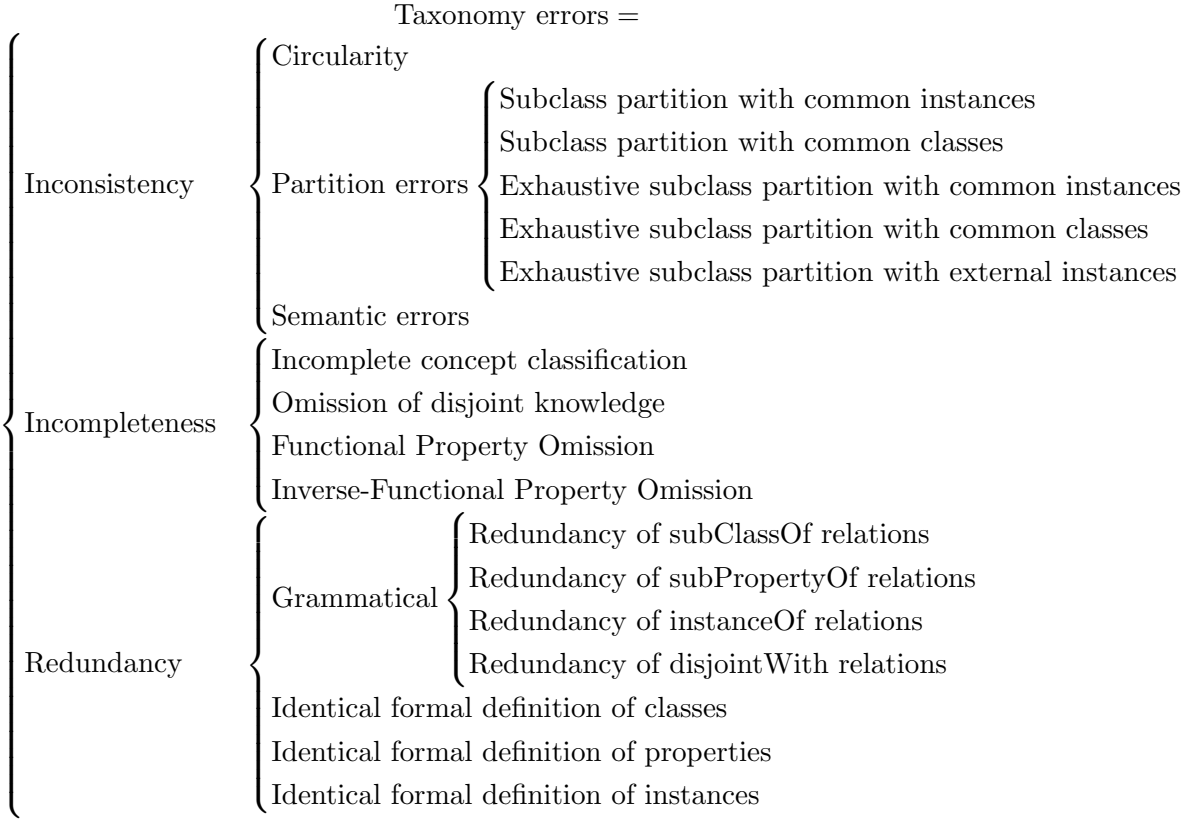

Figure 3.2: Classification of the taxonomy errors
Source: Handbook on ontologies[33] extended with [9]

or n, depending upon the number of relations involved when traversing the concept down the hierarchy of concepts until we get the same from where we started traversal. For example, circulatory error of distance 0 occurs when an ontologist models `OddNumber` concept as subclass of `NaturalNumber` and `NaturalNumber` as subclass of `OddNumber`. As OWL ontologies provide constructs to form property hierarchies, circulatory errors can also occur there in.

Partition errors

There are mainly several ways of classification depending upon the type of decomposition of superclass into subclasses. When all the features of subclasses are independently described and subclasses do not overlap with each other then it leads to disjoint decomposition. When ontologists follow the completeness constraint between the subclasses and the superclass, then

it leads to a complete or exhaustive decomposition. The other can depend on both the disjoint and exhaustive decomposition. Three types of errors are:

Semantic errors

This type of errors occurs if the developer makes incorrect statements during the classification, i.e. a class is declared as subclass of another class, to which it doesn't really belong. For instance if someone wants to create an ontology about plants and thereby makes a statement that class `Car` is subclass of the class `Rose`, which of course this doesn't reflect the real world.

Incompleteness

Sometimes ontologists made the classification of concepts but overlook some of the important information about them. Such incompleteness often creates ambiguity and lacks reasoning mechanisms. The following subsections give an overview of incompleteness errors.

Incomplete concept classification

Generally, an error of this type is made whenever concepts are classified without accounting for them all, that is, concepts existing in the domain are overlooked. An error of this type occurs if a concept classification `Transportation` is defined considering only the classes formed by `Cars` and `Planes` and overlooking, for example, the `Ships`.

Omission of disjoint knowledge

This error occurs when ontologists classify the concept into many subclasses and partitions, but omits disjoint knowledge axiom between them. For example an ontologist models the `BeachLocation`, `HistoricLocation` and `MountainLocation` as subclasses of `Location` concept, but omits to model the disjoint knowledge axiom between subclasses. Due to significant importance of disjoint axiom between classes, OWL allows to specify disjoint axioms between properties as well. So we also emphasis that ontologists should check and specify disjoint knowledge between properties, and avoid creating common instances between them.

Subclass partition omission

This type of error occurs if the developer defines a partition of a class, but omits to add an completeness constraint to the set of classes in the partition.

Exhaustive subclass partition omission

This error occurs when ontologists do not follow the completeness constraint while decomposition of concept into subclasses and partitions. For example ontologist models the `BeachLocation`, `HistoricLocation` and `MountainLocation` as disjoint subclasses of `Location` concept, but does not specify that whether or not this classification forms an exhaustive decomposition.

Functional Property Omission (FPO) for single valued property

According to Ontology Definition Metamodel, when there is only one value for a given subject then that property needs to be declared as functional. The tag Functional can be associated with both the object properties and datatype properties. For example `hasBlood_Group` as an object property between `Person` and `Blood_Group` is an example of functional object property. Every subject `Person` belongs to only one type of `BloodGroup`, so `hasBlood_Group` property should be tagged as functional so that person should be associated with one blood group. Likewise functional datatype properties allow only one range R for each domain instance D. Ignoring the functional tag allows a property to have more than one value leading to inconsistency. One of the main reasons for such inconsistency is that the ontologist has ignored that an OWL ontology by default supports multi-values for datatype property and object property.

Inverse-Functional Property Omission (IFPO) for a unique valued property

According to Ontology Definition Metamodel, inverse-functional property of the object determines the subject uniquely, i.e. it acts like a unique key in databases. This means that if we state P as an `owl:InverseFunctionalProperty`, then this restricts that for a single instance there can only be a value x, i.e. there cannot exist two different instances y and z such that both pairs (y, x) and (z, x) are valid instances of P. In OWL Full, a datatype property can be tagged as inverse-functional property because datatype property is a subclass of object property. But in OWL DL a datatype property can not be tagged as inverse-functional property because object properties and datatype properties are disjoint. An example of an inverse object property is `National_SecurityNo` that belongs to the `Person` as it uniquely identifies the `Person`. Ignoring inverse-functional tag with the property `National_SecurityNo` creates inconsistency within the ontology due to incomplete specification of concept. We consider such lack of information as

an error, because such ignorance leads machine not to infer and reason about concepts uniquely.

Redundancy

Redundancy is a type of error that occurs when we redefine expressions of the ontology that were already defined explicitly or that can be inferred from other definitions.

Redundancy of subClassOf, subPropertyOf and instanceOf relations

Redundancies of subClassOf error occur when ontologists specify classes that have more than one subClassOf relation directly or indirectly. Directly means that a subClassOf relation exist between the same source and target classes. Indirectly means that a subClassOf relations exist between a class and its indirect superclass of any level. For example ontologists specify `BeachLocation` as a subclass of `Location` and `Place`, and furthermore `Location` is defined as a subclass Of `Place`. Here indirect subClassOf relation exists between `BeachLocation` and `Place` creating redundancy. Likewise redundancy of subPropertyOf can exist while building property hierarchies. Redundancies of instanceOf relation occur when ontologists specify instance Swat as an instanceOf `Location` and `Place` classes, and it is already defined that `Location` is a subclass of `Place`. The explicit instanceOf relation between `SWAT` and `Place` creates redundancy as `SWAT` is indirect instance of `Place` and `Place` is a superclass of `Location`.

Redundancy of disjointWith relations (RDR)

Redundancy of disjointWith relation occurs when the concept is explicitly defined as disjoint with other concepts more than once. By Description Logic rules, if a concept is disjoint with any concept then it is also disjoint with its sub concepts. The one possible way of occurrence of RDR is that the concept is explicitly defined as disjoint with parent concept and also with its child concept. For an example, concept `Male` is defined as disjoint with `Female` and also with sub concepts of `Female`. This type of redundancy can occur due to direct disjointness (directly disjoint) and indirect disjointness (concept is disjoint with other because its parent is disjoint with it).

Identical formal definition of classes, properties and instances

Identical formal definition of classes, properties or instances may occur when the ontologist defines different (or same) names of two classes, properties or instances respectively, but provides the same formal definition.

3.4 Reasoning performance problems

OWL reasoners performance isn't bad, it's unpredictable. And that's a problem.

Non-experts often find predicting reasoner performance from their ontology disappointing. The connections between performance and the data are opaque in a way that is sometimes confounding and off-putting. In general all automated reasoners for expressive knowledge representation formalisms have this problem, some more so than other.

Contrast RDBMS technology. Not only is the underlying computational complexity much better, but many developers have internalized the technology such that predictions are more reliable and the connection between data, queries, and performance is more transparent. And the bad stuff is always bad, till one fix it, and the good stuff is always good till one break it. And if all else fails one can just EXPLAIN to get the way to happiness.

For serious OWL reasoner users, the analogue of EXPLAIN is either an email to the users list or a support contract, respectively. Another solution is to sniff out problems in ontologies and, ideally, automatically repair them. To achieve this efficiently the developers of the Pellet have build a tool, called Pellint, which analyses an ontology to find possible performance problems based on common error patterns.

At next we provide a very high-level description of the tableau algorithm for DLs and explain the main sources of its reasoning complexities. We refer the reader to [4] for a more detailed and accurate description of the tableau algorithms. After that we will describe which kinds of patterns are suggested to be a possible problem for existing tableau-based reasoners.

Tableau-based reasoning complexity

All the reasoning services in tableau algorithms can be reduced to consistency checking, which is done by building a completion graph. The nodes in the completion graph intuitively stand for individuals and literals. Each node is associated with its corresponding types. Property-value assertions are represented as directed edges between nodes. If we are checking the consistency of an ontology the initial completion graph is built from the asserted facts in

the ontology. If we are checking the satisfiability of a class the initial graph contains a single node whose type is that concept. The reasoner repeatedly applies the tableau expansion rules until a clash (i.e. a contradiction) is detected in the label of a node, or until a clash-free graph is found to which no more rules are applicable. In the process of tableau completion, there are two main sources of complexities: (1) non-determinism in "completing" the graph; and (2) the size of the graph built.

Non-determinism

Building the completion graph is non-deterministic due to disjunctions which are expressed with the `UnionOf` construct in OWL. The instances of the union class must be an instance of at least one of the union elements. The reasoner will do a case-by-case analysis to figure out if that is possible. A non-deterministic choice made because of a union class will have an impact on the completion graph, e.g. new edges may be added because of that choice. In the event that we made a wrong choice (which will be indicated by an inconsistency in the later stages of completion process) we have to backtrack and undo all the changes made to the graph. This effect multiplies if we have many choices to make. There are many optimization algorithms implemented in tableau-based reasoners to reduce the effects of non-determinism, but it is not hard to see how very large number of union classes will adversely affect reasoning time.

Completion graph size

The size of the completion graph depends on the size of the initial graph (i.e., the asserted instances), but also on the use of existential restrictions. Constructs like `SomeValuesFrom`, `MinCardinality`, and `ExactCardinality` will cause the tableau algorithm to create new nodes in the completion graph. Applying the tableau completion rules to new nodes will require more processing time and possibly increase the non-determinism involved because there might be new non-deterministic choices made for these new nodes. Predicting the exact size of the completion graph (without actually building the graph) is not possible, but in Pellint some heuristics and graph analysis techniques are used to compute an approximation of this size.

Patterns

There are two groups of patterns: *axiom-based* and *ontology-based*. Axiom-based patterns detect lints at the axiom level, typically at a single equivalent

classes axiom or a single subclass axiom; whereas ontology-based patterns detect lints that are established by two or more axioms in the whole ontology.

We will describe the patterns next, where the information for each pattern starts with its name and description. It is followed by example(s) of axioms demonstrating the pattern, and an explanation of why it may be problematic for reasoning.

Axiom-based Patterns

General Concept Inclusions (GCIs) A subclass axiom with a complex concept expression on the left hand side, or an equivalent axiom with two or more complex concepts.

Example 3.4.1

$$A \sqcap B \sqsubseteq C$$

or

$$C \sqcap D \equiv \exists P.E$$

Reasoning complexity:

A tableau-based reasoner deals with GCI axioms by converting them into a standard form. For example, $C \sqsubseteq D$ is converted into the axiom $\top \sqsubseteq \neg C \sqcup D$ where C and D can be arbitrary concepts. Since every individual is an instance of Thing, the reasoner then applies the converted axiom to every individual. We observe that every conversion produces a non-deterministic choice due to the OR construct, which is then applied to every individual. Hence GCI axioms are extremely expensive.

Equivalent to AllValue Restriction A named concept is equivalent to an AllValues restriction.

Example 3.4.2

$$A \equiv \forall R.C$$

Reasoning complexity:

An AllValues restriction does not require to have a property value but only restricts the values for existing property values. This means any concept not having the property value, e.g. a concept that is disjoint with the domain of the property, will satisfy the AllValues restriction

and turn out to be a subclass of the concept defined to be equivalent to the restriction. This typically leads to unintended inferences and additional inferences may eventually slow down reasoning performance.

Equivalent to Some Complement A named concept is equivalent to some complement.

Example 3.4.3

$$A \equiv \neg (C \sqcap \exists R.D)$$

Reasoning complexity:

An axiom of this pattern implies that two concepts are disjoint unions of Thing (i.e. $\top \sqsubseteq A \sqcup (C \sqcap \exists R.D)$ in the example) which adds to the general concept inclusion (GCI) axioms. This pattern typically indicates a modelling error since it forces every individual to be classified under one of the two possible definitions.

Equivalent to Top Top is equivalent to some concept or is part of an equivalent classes axiom.

Example 3.4.4

$$A \equiv \top$$

or

$$EquivalentClasses\,(A, \top, \exists R.C)$$

Reasoning complexity:

This pattern directly adds to the GCI axioms since it affects the definition of owl:Thing.

Large Cardinality Cardinality restriction is too large.

Example 3.4.5

$$A \sqsubseteq \leq 11R.\top$$

or

$$A \sqsubseteq \geq 11R.\top$$

or

$$A \sqsubseteq = 11R.\top$$

Reasoning complexity:

Min and exactly restrictions generate individuals during reasoning, which grows exponentially when these axioms interact with the others in a recursive manner. Setting the number too large on these restrictions may lead to intractable memory consumption during reasoning. On the other hand, a max restriction introduces non-determinism in choosing which individuals to merge during reasoning, which leads to intractable time complexity.

Large Disjunction Too many disjuncts in a disjunction.

Example 3.4.6

$$A \sqsubseteq C \sqcap (D_1 \sqcup D_2 \sqcup D_3 \sqcup D_4 \sqcup D_5 \sqcup D_6 \sqcup D_7 \sqcup D_8)$$

Reasoning complexity:

Disjunction is a source of non-determinism during reasoning, which leads to intractable time complexity.

Ontology-based Patterns

Concept with Equivalent and Subclass Axioms A named concept appears in an equivalent axiom and on the left-hand side of a subclass axiom.

Example 3.4.7

$$A \equiv C_1 \sqcap C_2$$
$$A \sqsubseteq \forall R.D$$

or

$$A \sqsubseteq \forall R.D$$
$$EquivalentClasses\,(A, B, C_1, C_2)$$

Reasoning complexity:

These implicitly define GCI axioms. For instance, $(C_1 \sqcap C_2) \sqsubseteq A$ is implicitly implied by both examples above.

Existential Explosion An existential restriction (some, as well as min and exactly) generates individuals in a tableau-based reasoner. Many existential restrictions interact in a complex manner and may generate an intractable number of individuals in such reasoners.

Example 3.4.8

$$
\begin{aligned}
&C \sqsubseteq \exists R.D \\
&D \sqsubseteq \exists R.E \\
&E \sqsubseteq \geq 10P.\top \\
&E \sqsubseteq \forall P.F \\
&F \sqsubseteq \exists R.G \\
&\quad \vdots
\end{aligned}
$$

Reasoning complexity:

A large number of individuals need to be maintained and kept in memory, which slows down reasoning significantly.

Too Many DifferentIndividuals Axioms Too many individuals involved in DifferentIndividuals axioms.

Example 3.4.9

$$DifferentIndividuals\,(Ind_1, Ind_2, Ind_3, Ind_4, Ind_5, Ind_6, Ind_7, Ind_8, Ind_9)$$

Reasoning complexity:

Some reasoners keep track of DifferentIndividuals for each individual separately. This means the memory consumption required to represent a DifferentIndividuals axiom is quadratically proportional to the number of individuals involved in the axiom; for `owl:differentFrom` there are only 2 individuals involved whereas for `owl:AllDifferent` there are arbitrary number of individuals involved. Using too many DifferentIndividual axioms increases the memory consumption significantly and affect reasoning performance.

3.5 Problems in Linked Data[7]

The common problems can be divided into five categories:

- the accessibility of a particular document,
- naming and dereferencability,
- interpretation of datatype literals,
- reasoning
- absence of good quality links

Accessibility

Document Not Retrievable

Simple: a document is not externally accessible on the Web... Not to dwell too much on the issue—and besides obvious causes such as the document being nonexistent—a publisher should ensure that the document is not an internal or local resource, that authentication is not required, and that the robots.txt settings do not conflict with (at least) low-volume external access.

Incorrect Content-Type

Related to the above issue of content negotiation, a server returns the media type of the returned content by means of the Content-Type field in the HTTP response header (cf. Section 14.17 of the HTTP specification) [8] . Again, the responding server should return the most specific media type which applies to the returned document format. The correct media types for various formats likely to be used around the Web of Data are:

A frequent problem that should be avoided is the use of the generic XML media types text/xml or application/xml for specific XML formats that have their own media type, such as XHTML, RDF/XML, or SPARQL results.

[7] This section contains material edited and adapted from the work done in the Pedantic Web Group - `http://pedantic-web.org/fops.html` with the permissions of the authors. Please also refer to [15] when mentioning this content.

[8] `http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17`

Format	Media type
RDF/XML	`application/rdf+xml`
Turtle	`text/turtle`
N-Triples	`text/plain`
N-Quads	`text/x-nquads`
HTML	`text/html`
XHTML	`text/html or application/xhtml+xml`
XHTML with RDFa	`application/xhtml+xml`
General JSON	`application/json`
SPARQL Query Result XML format	`application/sparql-results+xml`
SPARQL Query Result JSON format	`application/sparql-results+json`

Incorrect Content Negotiation

In practice, content negotiation is successfully used in the following scenarios:

- Redirecting users to country- or language-specific sub-sites based on their IP address.
- Delivering browser-specific versions of a site to work around browser incompatibilities, based on the browser's User-Agent header. (This is considered poor practice by Web standards advocates, but is common nonetheless.)
- Different translations of the same document are displayed based on the Accept-Language header. Browsers send different Accept-Language headers based on the settings chosen by the user in the browser or system preferences.
- Some REST-style Web APIs allow clients to choose between different data formats, such as XML and JSON, based on the client's Accept HTTP header. (The usefulness of this can be questioned, because clients are custom-built for those APIs, and that would be an easier task if the API would simply use different URIs for the different variants.)
- The Linked Data style of publishing RDF uses content negotiation to provide convenient human-readable versions of the published RDF data. Based on the Accept header, either RDF or HTML is delivered. Linked data browsers and other RDF clients have to send an appropriate Accept header to get the RDF.

Content Negotiation between Inappropriate Variants

Imagine a Web site that uses RDF to express incomplete metadata about its HTML pages. For each HTML page, say, /foo.html, there might be a corresponding RDF page at /foo.rdf that contains basic metadata (information about title, creator, creation date, and the like). But the RDF does not contain the actual main content of the HTML page. In this case, the RDF is not an appropriate alternate version of the HTML, because it does not contain the same information. Content negotiation between both variants from /foo would be inappropriate.

Another example—and one that does not involve RDF—is as follows: Imagine an important document, which is available in English, and in a Spanish translation. But the Spanish translation is not complete: the second half of the document is simply missing from the Spanish version. Again, content negotiation between both variants is inappropriate.

In general, content negotiation between different versions of the same content is only appropriate if all the variants contain the same information. Variances in format (e.g., JSON vs. XML), language, and quality (to some extent—e.g., pristine English words and a sloppy German translation), are acceptable. But if some variants give you more information then others, then content negotiation is harmful.

Incorrect interpretation of the Accept Header

Content negotiation is often presented in a simplified way: "If the client sends X in the Accept HTTP header, then the server returns format X. If the client sends Y, then the server returns Y." But this is not the whole story and if one think that it is, then he is very likely to implement content negotiation incorrectly.

Accept headers have a fairly complex syntax. In particular:

- Accept headers can include multiple media types, separated by comma. The following header would indicate that the client prefers either RDF/XML or Turtle: application/rdf+xml,text/turtle.

- Media types, such as text/html, can include additional parameters appended after a semicolon: text/html;charset=utf-8. It is often sufficient to just ignore the parameters.

- One parameter is of particular importance though: the quality parameter—also known as the q value. Clients use q values to indicate preference of some media types over others. In the following example, the

client indicates that it prefers RDF/XML, but would also accept HTML with a lower preference: application/rdf+xml;q=1.0,text/html;q=0.4. Note that a q value of 1.0 is the default and can be omitted. If a server has both RDF/XML and HTML, it should return RDF/XML, because the client has indicated a higher preference.

In the common case of negotiating between RDF and an HTML rendering thereof, commonly observed problems include:

- not recognising media types if they include a parameter—e.g., text/html;charset=utf-8 or application/rdf+xml;q=0.9;
- always sending HTML when several media types are specified in the Accept header;
- always sending HTML when both RDF and HTML are in the Accept header, even if RDF has a higher q value;
- choosing between RDF and HTML based on which appears first (or last) in the Accept header, rather than based on their q values;
- redirecting to a nonexistent URI, such as something.rdf.html, when both RDF and HTML are in the Accept header.

In particular, clients that accept both RDF/XML and HTML (e.g., browser plugins and clients that support RDFa as well as RDF/XML) run into problems because of server implementation problems...

If, for whatever reason, it is impossible to implement the full algorithm in the server environment, including q values, then an approximation will have to do. Here is a good one:

1. If no Accept header is sent by the client, assume that the client wants raw data; i.e., RDF/XML. (This is probably an unsophisticated client that has not been properly written to actually emit an appropriate Accept header, and it's much more likely that such a client is a quickly hacked data processing script than an HTML-processing Web browser.)
2. If a raw data format—such as application/rdf+xml—is mentioned, then send that format. (A client that can process HTML and RDF/XML can probably do more interesting things with the raw data, rather than its human-readable rendering.)
3. In all other cases, send HTML. (It's probably a Web browser.)

Note, however, that the existence of such heuristics is no excuse for not implementing correct handling of q values.

Content Negotiation with Missing Vary Header

Caches are essential to the efficient operation of the Web. HTTP caches sit between client and server, and store any cacheable server responses. When another client later on requests the same resource, then the cache may directly return the stored response. So the client receives a response without the origin server being hit at all. This can significantly reduce server load.

But for this to work, the cache has to know which responses are cacheable for what kinds of requests, and for how long. Servers can indicate this by using various HTTP headers in their responses.

Content negotiation and the Vary header. If a resource has multiple representations subject to content negotiation (e.g., it has an HTML representation and an RDF representation), then caches must be made aware of this. Otherwise they might return a cached HTML response to a client requesting RDF, not knowing that the server would handle these two requests differently.

To make caches aware of multiple representations, the server must include a Vary HTTP header with any response that is subject to content negotiation. The value of the Vary header is one or more names of other HTTP headers: the headers that the server uses to select a representation.

The typical case for content negotiation with RDF is that the Accept header is used to select the appropriate representation. Therefore, a Vary HTTP header like this has to be included in content-negotiated responses:

```
Vary: Accept
```

This will prevent caches from returning representations that were generated for a different Accept header, and will prevent hard-to-debug issues where a client inexplicably sees responses in an unexpected format.

Naming and Dereferencability

In RDF, we name things, give things values for named properties, define named relations to other named things and organise named things into named classes; in RDF we use URIs as names, which enables dereferencing: the URI name of a resource can be accessed, with the expectation that an RDF document is returned with some description of the named resource. Now, instead of copying and pasting all information available about all resources named in the document (or exhaustively linking to other documents using,

e.g., rdfs:seeAlso), one can simply use the dereferencable URI which an agent can resolve for more information.

There are two "recipes" for creating dereferencable URIs: one uses hash-based URIs whereas the other uses slash-based URIs. The best-practices for both have been covered extensively in many documents, such as Best Practice Recipes for Publishing RDF Vocabularies [9] and How to Publish Linked Data on the Web [10]. To summarise here—and possibly over-simplifying—dereferencable hash-based URIs are best suited to group the descriptions of a small or moderate number of related terms into one document and one location, allowing an agent to retrieve the descriptions of multiple related terms with one HTTP lookup; dereferencable slash-based URIs are best suited to provide individual documents for each of a large number of terms, such that an agent will not need to download a massive document to find the description of one term.

Redirects Other Than 303

Redirects are often used to point from the URI of a non-information resource to the document which describes it; in particular the 303 See Other redirect is recommended. Although most agents will support other redirect schemes—such as 301 Moved Permanently, or 302 Found—the 303 redirect has been agreed upon as the most suitable for accessing resource descriptions and should be used.

Malformed Datatype Literals

If a datatype-aware agent will receive an incorrect integer value, that agent will disagree. Datatype classes have what is called a lexical representation which defines the sequences of characters which are allowed in a literal of that class. The lexical representations for all datatype classes are defined in XML Schema Part 2: Datatypes Second Edition [11] .

> **integer** has a lexical representation consisting of a finite-length sequence of decimal digits (#x30-#x39) with an optional leading sign. If the sign is omitted, "+" is assumed.
> For example: -1, 0, 12678967543233, +100000.

[9] `http://www.w3.org/TR/swbp-vocab-pub/`
[10] `http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/`
[11] `http://www.w3.org/TR/xmlschema-2/`

From this, a datatype-aware agent will know that A is not a valid integer literal: in fact, asserting otherwise is an inconsistency. Although this example is fairly straightforward, many datatype classes have more complex lexical forms. In particular, the datatypes classes relating to date and time are subject to errors, the most common being dateTime:

The lexical space of dateTime consists of finite-length sequences of characters of the form: '-'? yyyy '-' mm '-' dd 'T' hh ':' mm ':' ss ('.' s+)? (zzzzzz)?, where

- '-'? yyyy is a four-or-more digit optionally negative-signed numeral that represents the year; if more than four digits, leading zeros are prohibited, and '0000' is prohibited. . . ;
- the remaining '-'s are separators between parts of the date portion;
- the first mm is a two-digit numeral that represents the month;
- dd is a two-digit numeral that represents the day;
- 'T' is a separator indicating that time-of-day follows;
- hh is a two-digit numeral that represents the hour; '24' is permitted if the minutes and seconds represented are zero, and the dateTime value so represented is the first instant of the following day (the hour property of a dateTime. . . cannot have a value greater than 23);
- ':' is a separator between parts of the time-of-day portion;
- the second mm is a two-digit numeral that represents the minute;
- ss is a two-integer-digit numeral that represents the whole seconds;
- '.' s+ (if present) represents the fractional seconds;
- zzzzzz (if present) represents the timezone (as described below).

For example, 2002-10-10T12:00:00-05:00 (noon on 10 October 2002, Central Daylight Savings Time as well as Eastern Standard Time in the U.S.) is 2002-10-10T17:00:00Z, five hours later than 2002-10-10T12:00:00Z.

The most common errors relating to dateTime include the use plain text values such as 12:32 Feb 7 2008, the omission of the mandatory seconds field,

and the omission of : delimiters.

Reasoning

Bogus Values for Inverse-Functional Properties

An inverse-functional property is a property whose value uniquely identifies a resource. Examples include properties such as ISBN codes for books, social security numbers for people, physical MAC addresses for devices, and so on. Inverse-functional properties are pretty handy on the Web: oftentimes people don't agree on what URI to use for a particular resource; e.g., a book; but as long as they give a consistent value for a consistent inverse-functional property; e.g., use the same ISBN property with the same value for that book; people don't have to agree on URIs and a reasoner will be able to conclude that it's the same book being discussed. In other words, since people don't always agree upon URIs, inverse-functional properties allow people to identify resources according to values already agreed upon (ISBNs, SSNs, MACs, etc.).

Publishers sometimes give nonsensical values for inverse-functional properties. The most common example of this is the FOAF inverse-functional property foaf:mbox_sha1sum, intended to represent an encoded version of a person's email address, and defined to uniquely identify a person. This property is commonly instantiated — particularly from social networking exporters which externalise a public FOAF profile for each of their users—and is subsequently used to match descriptions of people across different sites and different URI naming schemes. Unfortunately however, many exporters do not bother to validate user-input correctly (e.g., allow users to leave email fields blank) and hence export bogus values for foaf:mbox_sha1sum such as 08445a31a78661b5c746feff39a9db6e4e2cc5cf and da39a3ee5e6b4b0d3255bfef95601890afd80709; the former is the encoded sha1-sum of the string "mailto:" and the latter is the sha1-sum of an empty string. A quick Google [12] of the former value will reveal hundreds of thousands of results, which upon quick inspection, are mostly RDF files and values for foaf:mbox_sha1sum. Now, a reasoner will interpret any individual with this value for foaf:mbox_sha1sum as being the same person, resulting in what we call the "God Entity": an omnipresent individual with hundreds of thousands of names, locations, friends, homepages, and so on.

[12] `http://www.google.com/search?q=08445a31a78661b5c746feff39a9db6e4e2cc5cf`

Inconsistencies

Inconsistencies can occur if trying to reconcile different world-views from different data publishers. For example, an atheist will assert that God is a ImaginaryBeing whereas a theist will assert that God is a RealBeing, although ImaginaryBeing is clearly disjoint with RealBeing. Such disagreement can occur even in more concrete domains: a botanist will assert that a Tomato is a Fruit whereas a taxman will tell you that a Tomato is a Vegetable and apply a tariff accordingly. Such inconsistencies are due to genuine disagreement between publishers and—with the danger here of getting more coffee stains on lab-coats—are not a bad thing at all and probably best left unresolved.

However, most inconsistencies currently found on the Web result directly from mistakes in RDF documents or disagreement on the identification of resources, and can be resolved. Also, almost all inconsistencies are caused by resources found to be members of disjoint classes. One of the most common causes is using a URI to describe two completely different things: e.g., using a person's homepage URI to identify both the homepage and the person (clearly, a resource cannot be both a homepage and a person). Another common cause is using a property or class on the basis of its label and not verifying that its semantics are suitable. For example, the somewhat generically named foaf:img property is used to relate people to pictures they appear in, and so has its domain defined as foaf:Person (thus, every resource described with a value for foaf:img must be a foaf:Person); however, publishers commonly use this property on anything from documents to countries, leading to inconsistencies.

The need for linking, the absence of links

The dream that inspired many Semantic Web researchers is that of a web where bits of information are discovered and connected automatically because they "matter" for the task at hand, possibly coming from any web location and ultimately reused well beyond the purpose for which they were originally created. Applied to news, this vision would allow a reader to get "second and third" points of views when reading about a news article. Applied to commerce it would ideally eliminate the need for advertising: sellers and suppliers would simply "be found" for the characteristics of their offers.

Given no expected imminent breakthrough in the ways machine can understand content meant for human consumption, the idea of the Semantic Web initiative has been that of proposing that Web Site "lend a hand" to machines by encoding semantics using RDF. For years, however, RDF de-

scriptions on the Web have been made available almost exclusively by web data enthusiasts, i.e. by the Semantic Web community itself. Despite this, the community has been able to made available a remarkable amount of information, the LOD cloud, to the point that many entities, e.g. encyclopedic entities but also the people participating in the community, are often "described" (have metadata about themselves in RDF) in several dozen different independent RDF sources on the web.

The existence of descriptions alone, however, is not a sufficient condition for this data to be discovered automatically. For this reason the LOD community has been advocating the reuse of URIs of other sites as a way to create interlinks. In "How to publish linked data" [13], it is explained that to allow crawlers and agents to understand that a description is about something described also elsewhere, URIs from other sites should be used. For these URIs to be found, one should first manually select datasets from a maintained list of known datasets, then explore these to find suitable URIs to link to, and this for each entity to be linked. It is suggested that automatic methods be used when linking multiple entities [6] , but especially in this case it is necessary to know a priory which specific dataset to link to but and its schema in order to perform manual configuration of the matching algorithms, something that requires a high degree of expertise.

This complexity, together with the – arguably temporary - lack of immediate incentives for doing such task, makes it so that even among the LOD community interlinks are scarce, a quick query on Sindice, currently indexing approximately 65M semantic documents, shows that less than 4 million RDF documents (usually entity descriptions from the LOD cloud) exhibit at least 1 sameAs link.

In the last year however LOD is rapidly becoming not the only source of large amounts of RDF structured content. Thanks for the support of Google and Yahoo for RDFa encoded content for advanced snippets, it is safe to say that tens of millions of pages of database generated content have appeared, but none of these, to the best of our knowledge, is providing interlinks among descriptions on different websites.

It is safe to say that the problem of missing interlinks in RDFa descriptions - and of the very little number of interlinks also in the Semantic Web community - is "here to stay" because of the lack of perceived usefulness that they bring to the person or entity which should put them. Links on the web increase the value of a site by making it "more useful" to visitors, the same cannot be said of invisible RDF links. Also to be of use one would ex-

[13] "How to publish linked data" Bizer, C., Cyganiak, R. Heath, T 2007. `http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial`

pect to have many links from a single entry points, something which requires manual or semi assisted 1 to 1 connections. Finally, for being consistent on the Semantic Web, links would have to be bidirectional and maintained by multiple parties at the same time, something which is in essence contrary to the principles of decoupling which made the very web successful.

Furthermore there could be political and commercial reasons why a dataset provider might not have any incentive to put links, e.g. when in a dominant market position.

It is also partially recognizing these shortcomings that the EU project LATC [14] , started in 2010 is now addressing these issues with the creation of a 24/7 interlinking machine that is set to provide a core of high quality links with the intention of encouraging more and more datasets producers to join in and also use the LATC infrastructure to provide quality links.

[14] `http://latc-project.eu/`

Chapter 4

Tool Support

4.1 Tools related to syntactic errors

The Validating RDF Parser (VRP)[1] analyses, validates and processes RDF schemas and resource descriptions. This parser offers syntactic validation for checking if the input namespace conforms to the updated RDF/XML syntax proposed by W3C, and semantic validation for verifying constraints derived from RDF Schema Specification (RDFS).

RDF Validation Service[2] is based on HP-Labs Another RDF Parser (ARP17), which currently uses the version 2-alpha-1. This service supports the Last Call Working Draft specifications issued by the RDF Core Working Group, including datatypes. It offers syntactic validation for checking if the input namespace conforms to the updated RDF/XML syntax proposed by W3C. However, this service does not do any RDFS validation.

4.2 Tools related to semantic errors

There are a number of reasoners[3] which can detect logical errors in OWL ontologies. Most of them are based on the tableau-algorithm but differ from each other in the supported expressivity and optimizations. Moreover there other kinds of reasoners, using e.g. resolution or rules. Next we will give a short overview and a comparison in Tab. 4.2:

[1] Validating RDF Parser: `http://139.91.183.30:9090/RDF/VRP/`
[2] RDF Validation Service: `http://www.w3.org/RDF/Validator/`
[3] List of existing reasoners: `http://www.cs.man.ac.uk/~sattler/reasoners.html`

Pellet[1] is a free open-source Java-based reasoner for $\mathcal{SROIQ}$ with simple datatypes (i.e., for OWL 2). It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering). It supports the OWL-API, the DIG interface, and the Jena interface and comes with numerous other features.

KAON2 is a free (free for non-commercial usage) Java reasoner for $\mathcal{SHIQ}$ extended with the DL-safe fragment of SWRL. It implements a resolution-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering)[28]. It comes with its own, Java-based interface, and supports the DIG interface.

RacerPro[13] is a commercial (free trials and research licenses are available) lisp-based reasoner for SHIQ with simple datatypes (i.e., for OWL-DL with qualified number restrictions, but without nominals). It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, nRQL query answering). It supports the OWL-API and the DIG interface and comes with numerous other features.

Fact++[36] is a free (GPL/LGPL) open-source C++-based reasoner for $\mathcal{SROIQ}$ with simple datatypes (i.e., for OWL 2). It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval). It supports the OWL-API, the lisp-API and the DIG interface.

HermiT s a free (under LGPL license) Java reasoner for OWL 2/$\mathcal{SROIQ}$ with OWL 2 datatype support and support for description graphs. It implements a hypertableau-based decision procedure[29, 30, 31], uses the OWL API 3.0, and is compatible with the OWLReasoner interface of the OWL API.

CEL[2] is a free (for non-commercial use) LISP-based reasoner for $\mathcal{EL}+$. It implements a refined version of a known polynomial-time classification algorithm[3] and supports new features like module extraction and axiom pinpointing. Currently, it accepts inputs in a small extension of the KRSS syntax and supports the DIG interface.

SHER[8] is a commercial (free for academic use) Java-based reasoner for $\mathcal{SHIN}$. It is based on Pellet and uses database technology to reason

about $\mathcal{SHIN}$ TBoxes and large scale ABoxes (retrieval, conjunctive query answering)[10].

	Algorithm	Expressivity
Pellet	Tableau	$\mathcal{SROIQ}(D)$
KAON2	Resolution & Datalog	$\mathcal{SHIQ}(D)$
RacerPro	Tableau	$\mathcal{SHIQ}(D)$
Fact++	Tableau	$\mathcal{SROIQ}(D)$
HermiT	Hypertableau	$\mathcal{SROIQ}(D)$
CEL	Completion rules	$\mathcal{EL}+$
SHER	Tableau & Database	$\mathcal{SHIN}$

Table 4.1: Overview about reasoning systems, the algorithm they use and the supported expressivity.

There are a number of related tools for ontology repair:

Swoop[4] **[23]** is a Java-based ontology editor using web browser concepts. It can compute justifications for the unsatisfiability of classes and offers a repair mode. The fine-grained justification computation algorithm is, however, incomplete.

RaDON[5] **[20]** is a plugin for the NeOn toolkit. It offers a number of techniques for working with inconsistent or incoherent ontologies. It can compute justifications and, similarly to Swoop, offers a repair mode. RaDON also allows to reason with inconsistent ontologies and can handle sets of ontologies (ontology networks).

PION and DION[6] have been developed in the SEKT project to deal with inconsistencies. PION is an inconsistency tolerant reasoner, i.e. it can, unlike standard reasoners, return meaningful query answers in inconsistent ontologies. To achieve this, a four-valued paraconsistent logic is used. DION offers the possibility to compute justifications, but cannot repair inconsistent or incoherent ontologies.

Explanation Workbench[7] is a Protégé plugin for reasoner requests like class unsatisfiability or inferred subsumption relations. It can compute

[4]SWOOP: `http://www.mindswap.org/2004/SWOOP/`

[5]RaDON: `http://radon.ontoware.org/demo-codr.htm`

[6]PION: `http://wasp.cs.vu.nl/sekt/pion/`

DION: `http://wasp.cs.vu.nl/sekt/dion/`

[7]Explanation Workbench: `http://owl.cs.manchester.ac.uk/explanation/`

regular and laconic justifications [16], which contain only those axioms which are relevant for answering the particular reasoner request. This allows to make minimal changes to resolve potential problems.

RepairTab has been developed for Protégé based on [26]. Its aim is to support the user in finding and detecting errors in ontologies. Similarly to Explanation Workbench, it can compute explanations and display those parts of the involved axioms, which are responsible for an inconsistency. The main difference is that RepairTab uses a modified tableau algorithm. In addition, it shows the inferences, which can no longer be drawn after removing an axiom.and allows the user to make meaning additions to preserve those inferences, if desired.

4.3 Tools related to structural errors

The Validating RDF Parser (VRP) (described in Sec. 4.1)

RDF Validation Service (described in Sec. 4.1)

OWL Ontology Validator[8] can be used to check if an ontology conforms to a specific OWL species, since it validates an OWL ontology and reports as a result the OWL language species to which the ontology belongs: OWL Lite, OWL DL, and OWL Full. Besides, if requested, the validator returns a description of the classes, properties and individuals in the ontology in terms of the OWL Abstract Syntax.

OWL Validator[9] is based on the DAML Validator[10] (it uses a modified version of the Jena Toolkit). This tool is not a simple parser in the sense that it checks OWL ontologies not only for problems related to simple syntax errors, but also for other potential errors. The OWL Validator does not aim at perfoming full reasoning or inferencing, but only at checking these kinds of problems.

OdeVal[38] performs syntactic evaluation of RDF(S), DAML+OIL, and OWL ontologies, and evaluates their concept taxonomies from the point of view of knowledge representation using the ideas proposed in [12]. ODEval detects inconsistencies and redundancies in ontology concept taxonomies.

[8]OWL Ontology Validator: `http://www.mygrid.org.uk/OWL/Validator`
[9]OWL Validator: `http://owl.cs.manchester.ac.uk/validator/`
[10]DAML Validator: `http://www.daml.org/validator/`

			Validating RDF Parser	RDF Validation Service	OWL Ontology Validator	OWL Validator	OdeVal	Sindice Web Data Inspector
Inconsistency: Circularity Errors	At distance 0		✓	x	x	x	✓	x
	At distance 1		✓	x	x	x	✓	x
	At distance n		✓	x	x	x	✓	x
Inconsistency: Partition Errors	Common classes in disjoint decompositions	direct	x	x	x	x	✓	x
		indirect	x	x	x	x	✓	x
	Common classes in partitions		x	x	x	x	✓	x
	Common instances in disjoint decompositions	direct	x	x	x	x	✓	x
		indirect	x	x	x	x	✓	x
	Common instances in partitions		x	x	x	x	✓	x
	External classes in exhaustive decompositions		x	x	x	x	x	x
	External classes in partitions		x	x	x	x	x	x
	External instances in exhaustive decompositions		x	x	x	x	x	x
	External instances in partitions		x	x	x	x	x	x
Redundancy: Grammatical Errors	Redundancies of subclassOf relations	direct	x	x	x	x	✓	x
		indirect	x	x	x	x	✓	x
	Redundancies of instanceOf relations	direct	x	x	x	x	x	x
		indirect	x	x	x	x	x	x

Table 4.2: Feature Overview of Various Tools.

4.4 Tools related to performance errors

Pellint[11] **[27]** is an open source lint tool for Pellet which flags and (optionally) repairs modelling constructs that are known to cause performance problems. Pellint recognizes several patterns at both the axiom and ontology level as described in section 3.4.

4.5 Tools related to problems in Linked Data

W3C Markup Validation Service[12] checks syntactic correctness for web pages that contain embedded RDFa markup, provided the XHTML+RDFa doctype is used. The validator however only checks validity according to the RDFa DTD; this means that many kinds of errors, such as undeclared namespace prefixes, or invalid datatyped literals, are not detected.

Vapour[13] is a Linked Data validator which checks the dereferencability of a given URI and provides feedback on content negotiation/redirect codes and can determine whether the given URI is an information resource or a non-information resource. The system optionally checks whether the resolved document contains data about the dereferenced URI and has special features for performing checks on the dereferencability of vocabulary terms and namespaces.

URI Debugger[14] displays the request and response header for accessing a given URI; the system further allows for specifying custom User-Agent and Accept-Header fields. The content of the request is then displayed, with all URIs displayed with a link to recursively debug.

Sindice Web Data Inspector[15] is a tool to Visualize and Validate the structured data content available at a given web location (URL). The Web Data Inspector can be used to visualize and validate:

- RDF files
- HTML pages embedding microformats
- XHTML pages embedding RDFa

[11] Pellint: `http://pellet.owldl.com/pellint`
[12] W3C Markup Validation Service: `http://validator.w3.org/`
[13] Vapour: `http://validator.linkeddata.org/vapour`
[14] URI Debugger: `http://linkeddata.informatik.hu-berlin.de/uridbg/`
[15] Sindice Web Data Inspector: `http://inspector.sindice.net`

The Web Data Inspector works as a chain of extraction, validation and processing elements that create a final tabbed report. The services provided are:

Visualization:

- Rich triple visualization – see the content as sortable RDF triples
- SVG-based zoomable graph visualization
- Frame based visualization
- "Sig.ma" based visualization – a human friendly view of the main "topic"

Inspection and Validation Services:

- RDF Syntax Validation – based on the same engine hosted by the W3C
- RDFa Validator
- Pedantic Validation Service: performs reasoning and checks for common errors as observed in RDF data found on the web
- Ontology services: upon request, the inspector will perform Linked Data based ontology reasoning: ontologies are recursively fetched and used to compute the inference closure of the statements in the original data files. Inferred triples are visualized in a different colour for the above visualizers
- Ontology explorer: the chain of imports between ontologies can be explored in an interactive tree view

Chapter 5

Conclusion

As we have reported in this work there exist different types of problems and errors which can arise during the lifecycle of ontological knowledge bases and Linked Data. As these problems are not new to the community there is also a wide range of tools which were developed to tackle mostly one particular type or a subset of kind of errors, but to the best of our knowledge there is no system available which can handle all or even most of the problems. Further in the context of Linked Data, especially for the detection of logical/semantic errors, handling large amounts of data is a problem, because:

(a) Loading the whole data into standard OWL reasoners is nearly impossible.

(b) Often, not all of the data can be downloaded at once, because it is only available via Linked Data or a SPARQL endpoint. Downloading the data via SPARQL Queries or visiting all Linked Data resources is very time-consuming and puts a high load on the corresponding servers.

(c) Due to the nature of Linked Data, a network of knowledge bases needs to be taken into account when analysing semantic problems.

For this reason, we suggest to focus the work in LOD2, in particular the ORE tool, on the following aspects:

- Provide the possibility to analyse fragments of data instead of working with all the data in a knowledge base. Focus on extracting information, which is most relevant for detecting problems.

- Allow the possibility to access information via Linked Data and SPARQL in addition to loading RDF/OWL files.

- Use incremental reasoning and/or stream reasoning techniques to allow continuously adding further information to extracted fragments.
- Cover the structural problems which can be detected automatically.

Overall, our conclusion is that there is a demand for a system, which can handle most of the described problems and errors, which can automatically be detected, and is also be able to work on large knowledge bases in an efficient way. This goal will be pursued within the LOD2 project.

Bibliography

[1] Evren Sirin A, Bijan Parsia A, Bernardo Cuenca Grau A, Aditya Kalyanpur A, and Yarden Katz A. Abstract pellet: A practical OWL-DL reasoner, 2008.

[2] F. Baader, C. Lutz, and B. Suntisrivaraporn. CEL—a polynomial-time reasoner for life science ontologies. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 287–291. Springer-Verlag, 2006.

[3] F. Baader, C. Lutz, and B. Suntisrivaraporn. Efficient reasoning in $\mathcal{EL}^+$. In *Proceedings of the 2006 International Workshop on Description Logics (DL2006)*, CEUR-WS, 2006. To appear.

[4] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[5] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2007.

[6] Christian Bizer, Julius Volz, Georgi Kobilarov, and Martin Gaedke. Silk - a link discovery framework for the web of data. In *18th International World Wide Web Conference*, April 2009.

[7] Ronald J. Brachman. A structural paradigm for representing knowledge. Technical Report BBN Report 3605, Bolt, Beraneck and Newman, Inc., Cambridge, MA, 1978.

[8] Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Edith Schonberg, and Kavitha Srinivas. Scalable highly expressive reasoner (sher). *J. Web Sem.*, 7(4):357–361, 2009.

[9] Muhammad Fahad and Muhammad Abdul Qadir. A framework for ontology evaluation. In Peter W. Eklund and Ollivier Haemmerlé, editors, *ICCS Supplement*, volume 354 of *CEUR Workshop Proceedings*, pages 149–158, 2008.

[10] Achille Fokoue, Aaron Kershenbaum, Li Ma, Edith Schonberg, and Kavitha Srinivas. The summary abox: Cutting ontologies down to size. *The Semantic Web - ISWC 2006*, pages 343–356, 2006.

[11] Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Modular reuse of ontologies: Theory and practice. *J. Artif. Intell. Res. (JAIR)*, 31:273–318, 2008.

[12] Asunción Gómez-Pérez. Evaluating ontologies: Cases of study. *IEEE Intelligent Systems*, 16(3):391–409, 2001.

[13] V. Haarslev and R. Möller. Racer system description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*, pages 701–705. Springer-Verlag, 2001.

[14] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. CRC Press/Chapman & Hall, 2009.

[15] Aidan Hogan, Andreas Harth, Alexandre Passant, Stefan Decker, and Axel Polleres. Weaving the pedantic web. In *In Proceedings of the Linked Data on the Web WWW2010 Workshop (LDOW 2010)*, 2010.

[16] Matthew Horridge, Bijan Parsia, and Ulrike Sattler. Laconic and precise justifications in OWL. In *The Semantic Web - ISWC 2008*, volume 5318 of *LNCS*, pages 323–338. Springer, 2008.

[17] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible SROIQ. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 57–67. AAAI Press, 2006.

[18] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[19] Prateek Jain, Pascal Hitzler, Peter Yeh, Kunal Verma, and Amit Sheth. Linked data is merely more data. 2010.

[20] Qiu Ji, Peter Haase, Guilin Qi, Pascal Hitzler, and Steffen Stadtmüller. Radon - repair and diagnosis in ontology networks. In *ESWC 2009*, volume 5554 of *LNCS*, pages 863–867. Springer, 2009.

[21] Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding all justifications of OWL DL entailments. In *ISWC 2007*, volume 4825 of *LNCS*, pages 267–280, Berlin, Heidelberg, 2007. Springer.

[22] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and Bernardo Cuenca Grau. Repairing unsatisfiable concepts in owl ontologies. In *ESWC 2006*, volume 4011 of *LNCS*, pages 170–184, 2006.

[23] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo Cuenca Grau, and James Hendler. Swoop: A web ontology editing browser. *Journal of Web Semantics*, 4(2):144–153, 2006.

[24] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging unsatisfiable classes in OWL ontologies. *Journal of Web Semantics*, 3(4):268–293, 2005.

[25] M. Klein and D. Fensel. Ontology versioning for the Semantic Web. In *Proc. of the First Int. Semantic Web Working Symposium (SWWS)*, pages 75–91, 2001.

[26] Joey Sik Chun Lam, Derek H. Sleeman, Jeff Z. Pan, and Wamberto Weber Vasconcelos. A fine-grained approach to resolving unsatisfiable ontologies. *J. Data Semantics*, 10:62–95, 2008.

[27] Harris Lin and Evren Sirin. Pellint - a performance lint tool for pellet. In *OWLED 2008*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[28] B Motik. Reasoning in description logics using resolution and deductive databases. *PhD theis, University Karlsruhe, Germany*, 2006.

[29] Boris Motik, Rob Shearer, and Ian Horrocks. A Hypertableau Calculus for SHIQ. In Diego Calvanese, Enriso Franconi, Volker Haarslev, Domenico Lembo, Boris Motik, Sergio Tessaris, and Anny-Yasmin Turhan, editors, *Proc. of the 20th Int. Workshop on Description Logics (DL 2007)*, pages 419–426, Brixen/Bressanone, Italy, June 8–10 2007. Bozen/Bolzano University Press.

[30] Boris Motik, Rob Shearer, and Ian Horrocks. Optimized Reasoning in Description Logics using Hypertableaux. In Frank Pfenning, editor, *Proc. of the 21st Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNAI*, pages 67–83, Bremen, Germany, July 17–20 2007. Springer.

[31] Boris Motik, Rob Shearer, and Ian Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.

[32] S. Schlobach. Debugging and semantic clarification by pinpointing. In *Proceedings of ESWC 2005*, 2005.

[33] Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. Springer, Berlin, 1. edition, 2004.

[34] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25(1-2):161 – 197, 1998.

[35] Boontawee Suntisrivaraporn, Guilin Qi, Qiu Ji, and Peter Haase. A modularization-based approach to finding all justifications for OWL DL entailments. In *ASWC 2008*, volume 5367 of *LNCS*, pages 1–15. Springer, 2008.

[36] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[37] Hai Wang, Matthew Horridge, Alan L. Rector, Nick Drummond, and Julian Seidenberg. Debugging owl-dl ontologies: A heuristic approach. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 745–757. Springer, 2005.

[38] Óscar Corcho, Asunción Gómez-Pérez, Rafael González-Cabero, and María del Carmen Suárez-Figueroa. Odeval: A tool for evaluating rdf(s), daml+oil and owl concept taxonomies. In Max Bramer and Vladan Devedzic, editors, *AIAI*, pages 369–382. Kluwer, 2004.