DESCRIPTION LOGICS: REASONING SUPPORT FOR THE SEMANTIC WEB

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2004

By
Jeff Z. Pan
School of Computer Science

Contents

List of Tables

List of Figures

List of Examples

Abstract

DL-based ontologies play a key role in the Semantic Web. They can be used to describe the intended meaning of Web resources and can exploit powerful Description Logic (DL) reasoning tools, so as to facilitate machine understandability of Web resources. Their wider acceptance, however, has been hindered by (i) the semantic incompatibilities between OWL, the W3C Semantic Web standard ontology language that is based on Description Logics, and RDF(S), the W3C proposed foundation for the Semantic Web, as well as by (ii) the lack of support for customised datatypes and datatype predicates in the OWL standard.

This thesis proposes a novel modification of RDF(S) as a firm semantic foundation for many of the latest DL-based Semantic Web ontology languages. It also proposes two decidable extensions of OWL that support customised datatypes and datatype predicates. Furthermore, it presents a DL reasoning framework to support a wide range of decidable DLs that provide customised datatypes and datatype predicates.

In our framework, datatype groups provide a general formalism to unify existing ontology-related datatype and predicate formalisms. Based on datatype groups, datatype expressions can be used to represent customised datatypes and datatype predicates. The framework provides decision procedures for a wide range of decidable DLs that support datatype expressions, including those that are closely related to OWL and its two novel extensions mentioned above. A remarkable feature of the proposed framework is its flexibility: the hybrid reasoner is highly extensible to support new datatypes and datatype predicates, new forms of datatype expressions and new decidable Description Logics.

This thesis constitutes an advance in the direction of Description Logic reasoning support for the Semantic Web. It clarifies the vision of the Semantic Web and particularly the key role that Description Logics play in the Semantic Web; it should, therefore, be of value to both communities. It is hoped that this work can provide a firm foundation for further investigations of DL reasoning support for the Semantic Web and, in general, ontology applications.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Acronyms

$\mathcal{AL}$ = Attributive Language, a Description Logic that provides atomic concept, universal concept ($\top$), bottom concept ($\bot$), atomic negation, intersection ($\sqcap$), value restriction ($\forall R.\mathsf{C}$) and limited exist restriction ($\exists R.\top$)

$\mathcal{ALC}$ = $\mathcal{AL}$ extended with full concept negation ($\mathcal{C}$)

$\mathcal{ALC}(\mathcal{D})$ = $\mathcal{ALC}$ extended with an admissible concrete domain ($\mathcal{D}$)

DAML+OIL = the result of merging DAML-ONT and OIL

DAML-ONT = an early result of the DARPA Agent Markup Language (DAML) programme

DL = Description Logic

HTML = HyperText Markup Language

$\mathcal{L}(\mathcal{G})$ = a $\mathcal{G}$-combined DL, where $\mathcal{L}$ is a $\mathcal{G}$-combinable DL, and $\mathcal{G}$ is a conforming datatype group

OIL = Ontology Inference Layer

OWL = Web Ontology Language, a W3C recommendation

OWL-E = OWL extended with datatype Expressions

OWL-Eu = an unary restriction of OWL-E

RDF = Resource Description Framework

RDFS = RDF Schema

RDFS(FA) = RDFS with Fixed metamodeling Architecture

$\mathcal{S}$ = $\mathcal{ALC}_{R^+}$, i.e., $\mathcal{ALC}$ extended with transitive role axioms ($_{\mathcal{R}^+}$)

$\mathcal{SH}$ = $\mathcal{S}$ extended with role hierarchy ($\mathcal{H}$)

$\mathcal{SH}\boldsymbol{f}$ = $\mathcal{SH}$ extended with functional role axioms (*f*)

$\mathcal{SHI}$ = $\mathcal{SH}$ extended with inverse roles ($\mathcal{I}$)

$\mathcal{SHIO}$ = $\mathcal{SH}$ extended with nominals ($\mathcal{O}$) and inverse roles ($\mathcal{I}$)

$\mathcal{SHIQ}$ = $\mathcal{SHQ}$ extended with inverse roles ($\mathcal{I}$)

$\mathcal{SHOQ}$ = $\mathcal{SHQ}$ extended with nominals ($\mathcal{O}$)

$\mathcal{SHOQ}(\mathbf{D})$ = $\mathcal{SHOQ}$ extended with a conforming universal concrete domain (**D**)

$\mathcal{SHOIQ}$ = $\mathcal{SHQ}$ extended with nominals ($\mathcal{O}$) and inverse roles ($\mathcal{I}$)

$\mathcal{SHOIQ}(\mathcal{G})$ = $\mathcal{SHOIQ}$ extended with a conforming datatype group ($\mathcal{G}$)

$\mathcal{SHQ}$ = $\mathcal{SH}$ extended with qualified number restrictions ($\mathcal{Q}$)

$\mathcal{SI}$ = $\mathcal{S}$ extended with inverse roles ($\mathcal{I}$)

SW = Semantic Web

XML = Extensible Markup Language

W3C = World Wide Web Consortium

Acknowledgements

I am primarily indebted to all the people who helped and inspired me during the preparation of this thesis: Ian Horrocks, Ulrike Sattler, Carsten Lutz, Peter Patel-Schneider, Enrico Franconi, Alan Rector, Peter Aczel, Carole Goble, Norman Paton, Sean Bechhofer, Alvaro Fernandes, David Rydeheard, Ning Zhang, Ella Tian, Jos de Bruijn, Kevin Garwood, Daniele Turi, Mike Bada, Phil Lord, Hai Wang, Dmitry Tsarkov, Matthew Horridge, Simon Harper and many present and past members of Information Management Group (IMG), in particular those who gave up their valuable time to read various parts of my drafts with incredible care and to offer comments that have led to many improvements.

I want to express my heartfelt thanks to my supervisor Ian Horrocks. Without his support, advice and inspiration, the work presented in this thesis would not have been possible. I would also like to thank Enrico Franconi, for introducing Ian as my supervisor, and Ulrike Sattler, for serving as a supervisor when Ian is not around.

This work has been supported by a grant from the Overseas Research Students Award Scheme.

To my parents

Chapter 1

Introduction

In *Realising the Full Potential of the Web* [15], Tim Berners-Lee identifies two major objectives that the Web should fulfil. The first goal is to enable people to work together by allowing them to share knowledge. The second goal is to incorporate tools that can help people analyse and manage the information they share in a meaningful way. This vision has become known as the *Semantic Web* (SW) [16].

The Web's provision to allow people to write online content for other people is an appeal that has changed the computer world. This same feature that is responsible for fostering the first goal of the Semantic Web, however, hinders the second objective. Much of the content on the existing Web, the so-called *syntactic Web*, is human but not machine readable. Furthermore, there is great variance in the quality, timeliness and relevance [15] of Web resources (i.e., Web pages as well as a wide range of Web accessible data and services) that makes it difficult for programs to evaluate the worth of a resource.

The vision of the Semantic Web is to augment the syntactic Web so that resources are more easily interpreted by programs (or 'intelligent agents'). The enhancements will be achieved through the *semantic markups* which are machine-understandable annotations associated with Web resources.

1.1 Heading for the Semantic Web

Encoding semantic markups will necessitate the Semantic Web adopting an annotation language. To this end, the W3C (World Wide Web Consortium) community has developed a recommendation called Resource Description Framework (RDF) [81]. The development of RDF is an attempt to support effective creation, exchange and use of

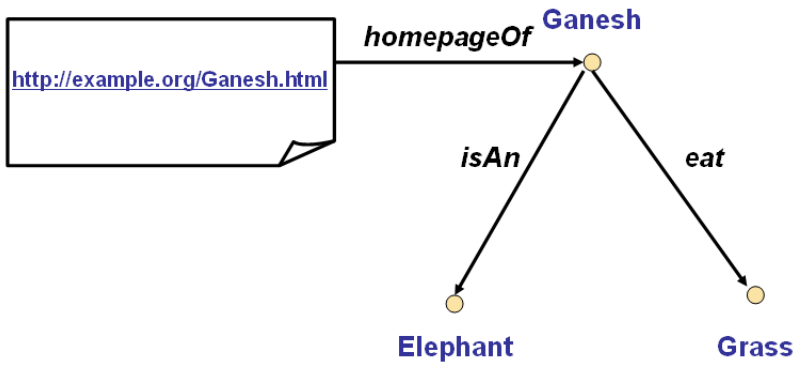

Figure 1.1: RDF annotations in a directed labeled graph

annotations on the Web.

Example 1.1 Annotating Web Resources in RDF

As shown in Figure 1.1, we can associate RDF annotation[1] to `http://example.org/Ganesh.html` and state that it is the homepage of the resource Ganesh, which is an elephant and eats grasses.

We invite the reader to note that the above RDF annotations are different from HTML [125] mark-ups in that they describe the contents of Web resources, instead of the presentations of Web pages. ♢

Annotations alone do not establish semantics of what is being marked-up. For example, the annotations presented in Figure 1.1 does not explain what elephants mean. In response to this need for more explicit meaning, ontologies [46, 142] have been proposed to provide shared and precisely defined terms and constraints to describe the meaning of resources through annotations — such annotations are called *machine-understandable annotations*.

An *ontology* typically consists of a hierarchical description of important concepts in a domain, along with descriptions of the properties of each concept, and constraints about these concepts and properties. Here is an example of an ontology.

Example 1.2 An Elephant Ontology

An elephant ontology might contain concepts, such as animals, plants, elephants, adult elephants (elephants with their ages greater than 20) and herbivores (animals that eat only plants or parts of plants), as well as constraints that elephants are a kind of animal, and that adult elephants eat only plants. These constraints allow the concept

[1] See Section 3.1.1 for precise definitions of RDF syntax.

'adult elephants' to be unambiguously interpreted, or understood, as a specialisation of the concept 'herbivores' by, e.g., an animal feeding agent.

Note that when we define complex concepts, such as 'adult elephants', we can refer to not only abstract entities, such as the concept 'elephants', but also concrete entities, such as 'greater than 20', which are called *datatypes*.[2] Both concepts and datatypes are very useful in ontologies; therefore, the DL reasoning framework to be presented in this thesis support both of them. ♢

Besides playing an important role in the Semantic Web, ontologies are popular through other applications, such as information retrieval and extraction [48, 13, 96, 145, 80, 137, 30], information integration [14, 97, 92], enterprise integration [47, 143], electronic commerce [83, 95, 84] and medical and bioinformatics terminology systems [127, 122, 43, 134, 136, 44, 133]. Standard SW ontology languages can benefit users of these fields by improving inter-operability of ontologies and by enabling them to exploit the reasoning services for these languages.

The advent of RDF Schema (RDFS) [22] represented an early attempt at a SW ontology language based on RDF. RDF and RDFS, or simply RDF(S), are intended to provide the foundation for the Semantic Web [94, Sec. 7.1]. As the constructors that RDFS provides for constructing ontologies are very primitive, more expressive SW ontology languages have subsequently been developed, such as OIL [57], DAML+OIL [68] and OWL [12], which are all based on Description Logics.

1.2 Description Logics and the Semantic Web

Description Logics (DLs) [7] are a family of class-based knowledge representation formalisms, equipped with well-defined model-theoretic semantics [6]. They were first developed to provide formal, declarative meaning to semantic networks [123] and frames [99], and to show how such structured representations can be equipped with efficient reasoning tools. The basic notions of Description Logics are classes, i.e., unary predicates that are interpreted as sets of objects, and properties, i.e., binary predicates that are interpreted as sets of pairs.

Description Logics are characterised by the constructors that they provide to build complex class and property descriptions from atomic ones. For example, 'elephants

[2] Following the XML Schema type system, 'greater than 20' can be seen as a customised, or derived, datatype of the base datatype $integer$; cf. Section 3.3.1.

with their ages greater than 20' can be described by the following DL class description:[3]

$$\mathsf{Elephant} \sqcap \exists age.\ >_{20},$$

where Elephant is an atomic class, age is an atomic datatype property, $>_{20}$ is a customised datatype (treated as a unary datatype predicate) and $\sqcap$,$\exists$ are class constructors. As shown above, datatypes and predicates (such as $=, >, +$) defined over them can be used in the constructions of class descriptions. Unlike classes, datatypes and datatype predicates have obvious (fixed) extensions; e.g., the extension of $>_{20}$ is all the integers that are greater than 20. Due to the differences between classes and datatypes, there are two kinds of properties: (i) object properties, which relate objects to objects, and (ii) datatype properties, which relate objects to data values, which are instances of datatypes.

Class and property descriptions can be used in axioms in DL knowledge bases. DL *Axioms* are statements that describe (i) relations between class (property) descriptions, (ii) characteristics of properties, such as asserting a property is transitive, or (iii) instance-of relations between (pairs of) individuals and classes (properties). We can use DL axioms to represent concepts and constraints in an ontology. For example, we can define the class AdultElephant with the following DL axiom

$$\mathsf{AdultElephant} \equiv \mathsf{Elephant} \sqcap \exists age.\ >_{20};$$

we can represent the constraint 'Elephant are a kind of Animal':

$$\mathsf{Elephant} \sqsubseteq \mathsf{Animal};$$

we can also assert that the object `Ganesh` is an instance of the class description 'Elephants who are older than 25 years old':

$$\texttt{Ganesh} : (\mathsf{Elephant} \sqcap \exists age.\ >_{25}).$$

In general, we can represent an ontology with (part of) a DL knowledge base.

Description Logics have distinguished logical properties. They emphasise on the decidability of key reasoning problems, such as class satisfiability and knowledge base satisfiability. They provide decidable reasoning services [39], such as tableaux algorithms (cf. Section 2.2), that deduce implicit knowledge from the explicitly represented

[3] Precise definitions of syntax and semantics are presented in Chapter 2.

knowledge. For example, given the above axioms, a DL reasoner (with datatype support) should be able to infer that `Ganesh` is an AdultElephant. Highly optimised DL reasoners (such as FaCT [59], Racer [51] and DLP [115]) have showed that tableaux algorithms for expressive DLs lead to a good practical performance of the system even on (some) large knowledge bases.

High quality ontologies are pivotal for the Semantic Web. Their construction, integration, and evolution crucially depend on the availability of a well-defined semantics and powerful reasoning tools (cf. Section 3.1.3). Description Logics address *both* these ontology needs; therefore, they are ideal logical underpinnings for SW ontology languages (Baader et al. [9]). Unsurprisingly, the SW ontology languages OIL, DAML+OIL and OWL use DL-style model-theoretic semantics. This has been recognised as an essential feature in these languages, since it allows ontologies, and annotations using vocabulary and constraints defined by ontologies, to be shared and exchanged without disputes as to their precise meaning.

DLs and insights from DL research had a strong influence on the design of these Web ontology languages. The influence is not only on the formalisations of the semantics, but also on the choice of language constructors, and the integration of datatypes and data values. OIL, DAML+OIL and OWL thus can be viewed as expressive DLs with Web syntax.

Among these SW ontology languages, OWL is particularly important. OWL has been adopted as the standard (W3C recommendation) for expressing ontologies in the Semantic Web. There are three sub-languages of OWL: OWL Lite, OWL DL and OWL Full. In this thesis, when we mention ‘OWL’ we usually mean ‘OWL DL’ because OWL Lite is simply a sub-language of OWL DL, while OWL Full can be seen as an unsuccessful attempt at integrating RDF(S) and OWL DL (cf. Section 3.2.2).

1.3 The Two Issues

The wider acceptance of DL-based ontologies in the Semantic Web has been hindered, however, by the following issues.

Issue 1: Semantic Mismatch Between RDF(S) and OWL

Semantic Web ontology languages are supposed to be extensions and thus compatible with RDF(S) [16]. There are, however, at least three known problems in extending the

RDF(S) semantics [53] with OWL constructors [116, 117, 74], i.e., the problems of 'too few entailments', 'contradiction classes' and 'size of the universe', all of which stem from some unusual characteristics of RDF(S) (cf. Section 3.2.2). In short, the intended foundation of the Semantic Web and DL-based SW ontology languages are not compatible with each other.

Here comes our worrying vision: the annotations of Web resources are represented in RDF(S), while the ontologies of various SW application are represented in DL-based Web ontology languages; between them is their semantic mismatch, which discourages existing RDF(S) users from adopting OWL, the newer standard [21].

Issue 2: No Customised Datatypes and Datatype Predicates

Although OWL is rather expressive, it has a very serious limitation; i.e., it does not support customised datatypes and datatype predicates. It has been pointed out that many potential users will not adopt OWL unless this limitation is overcome [126]. This is because it is often necessary to enable users to define their own datatypes and datatype predicates for their applications. In what follows, we give some examples to illustrate the usefulness of customised datatypes and datatype predicates in various SW and ontology applications.

Example 1.3 Semantic Web Service: Matchmaking

Matchmaking is a process that takes a service requirement and a group of service advertisements as input, and returns all the advertisements that may potentially satisfy the requirement. In a computer sales ontology, a service requirement may ask for a PC with memory size greater than 512Mb, unit price less than 700 pounds and delivery date earlier than 15/03/2004.

Here 'greater than 512', 'less than 700' and 'earlier than 15/03/2004' are customised datatypes of base datatypes integer, integer and date, respectively. ♢

Example 1.4 Ontology Merging: Unit Mapping[4]

Suppose we need to merge two map ontologies A and B: A describes distance in miles, while B describes distance in kilometres. When we merge A and B, we need to set up a bridging constraint to make sure that the data value of the distance in miles (in ontology A) should be equal to 1.6 times the data value of distance (between the

[4]Take the length units for example: there are so many of them (actually there are more than one hundred length units according to `http://www.chemie.de/`) that it is very likely that different ontologies might use different units.

same pairs of locations) in kilometres; otherwise, there will be data inconsistencies in the merged ontology.

Here, 'multiplication' is a datatype predicate defined over the rationals, while 'multiply by 1.6' is a customised datatype predicate. ♢

Example 1.5 Electronic Commerce: A 'No Shipping Fee' Rule

Electronic shops may need to classify items according to their sizes, and to reason that an item for which the sum of height, length and width is no greater than 15cm belongs to a class in their ontology, called 'small-items'. Then they can have a rule saying that for 'small-items' no shipping costs are charged. Accordingly, the billing system will charge no shipping fees for all the instances of the 'small-items' class.

Here 'greater than 15' is a customised datatype, 'sum' is a datatype predicate, while 'sum no greater than 15' is a customised datatype predicate. ♢

The usefulness of customised datatypes and datatype predicates in SW and ontology applications means that it is necessary to provide DL reasoning support for some extension of OWL that supports them.

To sum up, the SW standards RDF(S) and OWL are results of restricted techniques available when they were designed as well as political compromises within the corresponding W3C working groups. We will provide more details about the two issues in Chapter 3.

1.4 Objectives

The aim of the thesis is, therefore, to provide further techniques to solve the two issues presented in the last section. This aim is further developed into the following objectives:

1. To propose a novel modification of RDF(S) as a firm semantic foundation for the latest DL-based SW ontology languages.

2. To propose some extensions of OWL, so as to support customised datatypes and datatype predicates.

3. To provide a DL reasoning framework, which (i) supports customised datatypes

and datatype predicates, (ii) integrates a family of decidable (including very expressive) DLs with customised datatypes and datatype predicates, and (iii) provides decision procedures and flexible reasoning services for some members of this family that are closely related to OWL and the proposed extensions.

To fulfill these objectives, this thesis makes the following main contributions:

- the design of *RDFS(FA)*,[5] a sub-language of RDFS with DL-style model-theoretic semantics, which provides a firm foundation for using DL reasoning in the Semantic Web and thus solidifies RDF(S)'s proposed role as the foundation of the Semantic Web (Chapter 4);
- the *datatype group approach*, which specifies a formalism to unify datatypes and datatype predicates, and to provide a wide range of *decidable* Description Logics (including very expressive ones) integrated with customised datatypes and datatype predicates (Chapter 5);
- the design of *OWL-E*, a decidable extension of OWL that provides customised datatypes and datatype predicates based on datatype groups, and its unary restriction *OWL-Eu*, which is a much smaller extension of OWL (Chapter 6);
- the design of practical *tableaux algorithms* for a wide range of DLs that are combined with arbitrary conforming datatype groups, including those of a family of DLs that are closely related to OWL, DAML+OIL, OWL-Eu and OWL-E (Chapter 6);
- a flexible *framework architecture* to support decidable Description Logic reasoning services for Description Logics integrated with datatype groups. (Chapter 7).

We invite the reader to note that although the above contributions are made in the context of the Semantic Web, they can be applied to general ontology applications.

By fulfilling these objectives, this thesis shows that Description Logics can provide clear semantics, decision procedures and flexible reasoning services for SW ontology languages, including those that provide customised datatypes and datatype predicates.

[5] RDFS with Fixed metamodeling Architecture.

1.5 Reader's Guide

The remainder of the thesis is organised as follows.

Chapter 2 Introduces the basic DL formalisms, reasoning services, reasoning techniques and, most importantly, the existing approaches to integrating Description Logics with datatype predicates.

Chapter 3 Explains the importance of DL-based ontologies in the Semantic Web, presents RDF(S), OWL, and their datatype formalisms, and clarifies the two issues that the rest of the thesis is going to tackle.

Chapter 4 Presents RDFS(FA) as a strong connection between OWL and RDF(S), defining its DL style model theoretic-semantics and RDF style axioms, explaining how it solves the problem of layering OWL on top of RDF(S), and therefore establishing a firm foundation for using DL reasoning in the Semantic Web and clarifying the vision of the Semantic Web.

Chapter 5 Proposes a general formalism to unify existing ontology-related data- type and predicate formalisms and investigates, in the unified formalism, a family of decidable DLs that provide customised datatypes and datatype predicates.

Chapter 6 Proposes decidable extensions of OWL, i.e., OWL-E and OWL-Eu, that support customised datatypes and datatype predicates, and designs practical decision procedures for a family of Description Logics that are closely related to OWL and the proposed extensions.

Chapter 7 Introduces a flexible architecture for the framework, which is based on an extension of the current general DL interface DIG/1.1, for providing DL inferencing services for OWL and OWL-E.

Chapter 8 Describes a prototype implementation of the tableaux algorithm for the $\mathcal{SHIQ}(\mathcal{G})$ DL based on the FaCT system, along with two simple type checkers.

Chapter 9 Reviews the work presented and the extent to which the stated objectives have been met. The significance of the major results is summarised, and perspectives for further research are sketched.

Some results in this thesis have previously been published: some aspects of RDFS(FA) in Chapter 4 appeared in [108, 110, 113, 112]; an early version of the datatype group

approach presented in Chapter 5 has been published in [114]; a brief introduction of OWL-E presented in Chapter 6 appeared in [107]; finally, the $\mathcal{SHOQ}(\mathbf{D_n})$ DL, which is closely related to the family of DLs presented in Chapter 6, and its reasoning support for the Web ontology languages were published in [106, 111, 109].

Chapter 2

Description Logics

Chapter Aims

- To introduce the basic formalisms of Description Logics (DLs).
- To introduce basic DL reasoning techniques.
- To describe the existing approaches to integrating description languages with datatype predicates and to discuss their limitations.

Chapter Plan

2.1 Foundations

Description Logics (DLs) [7] are a family of logic-based knowledge representation formalisms designed to represent and reason about the knowledge of an application domain in a structured and well-understood way. They are based on a common family of languages, called *description languages*, which provide a set of constructors to build concept (class) and role (property) descriptions. Such descriptions can be used in axioms and assertions of DL knowledge bases and can be reasoned about w.r.t. DL knowledge bases by DL systems.

2.1.1 Description Languages

The foundations of description languages are concept and role *descriptions* (or concepts and roles for short). Intuitively, a *concept* represents a class of objects sharing some common characteristics, while a *role* represents a binary relationship between objects, or between objects and data values. We do not cover Description Logics integrated with datatype predicates in this section but will introduce them in Section 2.3.

A description language consists of an alphabet of distinct concept names ($\mathbf{C}$), role names ($\mathbf{R}$) and individual (object) names ($\mathbf{I}$); together with a set of constructors to construct concept and role descriptions. For a description language $\mathcal{L}$, we call $\mathbf{Cdsc}(\mathcal{L})$ and $\mathbf{Rdsc}(\mathcal{L})$ the set of concepts and roles of $\mathcal{L}$, respectively.

Description Logics have a model theoretic semantics, which is defined in terms of interpretations. An *interpretation* (written as $\mathcal{I}$) consists of a *domain* (written as $\Delta^{\mathcal{I}}$) and an *interpretation function* (written as $\cdot^{\mathcal{I}}$), where the domain is a nonempty set of objects and the interpretation function maps each individual name $\mathtt{a} \in \mathbf{I}$ to an element $\mathtt{a}^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, each concept name $\mathsf{CN} \in \mathbf{C}$ to a subset $\mathsf{CN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, and each role name $RN \in \mathbf{R}$ to a binary relation $RN^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function can be extended to give semantics to concept and role descriptions (see, e.g., Table 2.1). In the rest of the thesis, we will adopt the above notations (i.e., font styles) of individual names ($\mathtt{a}$), class names (CN) and role names (RN).

Description languages are distinguished by the constructors they provide; each language is named according to a convention introduced in [131], where the language $\mathcal{AL}$ (acronym for 'attributive language') was introduced as a minimal language that is of practical interest. The other languages of this family are extensions of $\mathcal{AL}$. Each additional constructor is associated with a special capital letter (e.g., $\mathcal{C}$ for general negation and $\mathcal{U}$ for disjunction, see Table 2.1).

In modern DLs, the language $\mathcal{S}$ is often used as a minimal language, which has previously been called $\mathcal{ALC}_{R^+}$ according to the above convention; we call this langauge $\mathcal{S}$ because it relates to the propositional (multi) modal logic $\mathbf{S4}$ [129], and because we want to avoid names becoming too cumbersome when adding letters to represent additional features [67]. Well known $\mathcal{S}$-family languages include $\mathcal{SI}$ [64], $\mathcal{SH}$, $\mathcal{SHI}$ [63], $\mathcal{SH}f$ [58, 138], $\mathcal{SHIQ}$ [76, 67] and $\mathcal{SHOQ}(\mathbf{D})$ [75],[1] etc.

Now we show how we build $\mathcal{S}$-roles and $\mathcal{S}$-concepts from role names and concept names. $\mathcal{S}$-roles are simply role names; i.e., $\mathcal{S}$ provides no constructor to build $\mathcal{S}$-roles. On the other hand, according to Definition 2.1, $\mathcal{S}$ provides quite a few constructors to

[1] $\mathcal{SH}f$, $\mathcal{SHIQ}$ and $\mathcal{SHOQ}(\mathbf{D})$ are pronounced as 'chef', 'shick' and 'shock D', respectively.

build $\mathcal{S}$-concepts, viz. concept descriptions of $\mathcal{S}$. This is common in many DLs, and that is why description languages are sometimes also called *concept languages*.

Definition 2.1. **($\mathcal{S}$-concepts)** Let $\mathsf{CN} \in \mathbf{C}$ be an atomic concept name, $R \in \mathbf{Rdsc}(\mathcal{S})$ an $\mathcal{S}$-role and $C, D \in \mathbf{Cdsc}(\mathcal{S})$ $\mathcal{S}$-concepts. Valid $\mathcal{S}$-concepts are defined by the abstract syntax:

$$C ::= \top \mid \bot \mid \mathsf{CN} \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C$$

The semantics of $\mathcal{S}$-concepts is given in Table 2.1 ◇

Constructor	Syntax	Semantics
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
concept name	CN	$\mathsf{CN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
general negation ($\mathcal{C}$)	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction ($\mathcal{U}$)	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
exists restriction ($\mathcal{E}$)	$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
value restriction	$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$

Table 2.1: Semantics of $\mathcal{S}$-concepts

Example 2.1 Building an $\mathcal{S}$-Concept Let $\mathsf{Plant} \in \mathbf{C}$ be a concept name, eat, $partOf \in \mathbf{Rdsc}(\mathcal{S})$ $\mathcal{S}$-roles,

$$\forall eat.(\mathsf{Plant} \sqcup \exists partOf.\mathsf{Plant})$$

is an $\mathcal{S}$-concept, following the abstract syntax in Definition 2.1:

1. Plant is an $\mathcal{S}$-concept;
2. $\exists partOf.\mathsf{Plant}$ is an $\mathcal{S}$-concept;
3. hence $\mathsf{Plant} \sqcup \exists partOf.\mathsf{Plant}$ is an $\mathcal{S}$-concept;
4. finally, $\forall eat.(\mathsf{Plant} \sqcup \exists partOf.\mathsf{Plant})$ is an $\mathcal{S}$-concept. ♢

Table 2.1 shows how the interpretation function is extended to give semantics to $\mathcal{S}$-concepts. An interpretation $\mathcal{I}$ is said to be a model of a concept C, or $\mathcal{I}$ models C, if the interpretation of C in $\mathcal{I}$ (viz. $C^{\mathcal{I}}$) is not empty.

Example 2.2 Interpretation of an $\mathcal{S}$-Concept $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is a model of $\forall eat.(\mathsf{Plant} \sqcup \exists partOf.\mathsf{Plant})$ where

$$\Delta^{\mathcal{I}} = \{\texttt{Ganesh, Bokhara, Balavan,grass1,stone1}\}$$

and the interpretation function $\cdot^{\mathcal{I}}$ is defined by:

$$\begin{aligned} \mathsf{Plant}^{\mathcal{I}} &= \{\texttt{grass1}\} \\ eat^{\mathcal{I}} &= \{\langle \texttt{Ganesh,grass1}\rangle, \langle \texttt{Bokhara,stone1}\rangle\} \\ partOf^{\mathcal{I}} &= \emptyset. \end{aligned}$$

According to Table 2.1, we have

$$\begin{aligned} (\exists partOf.\mathsf{Plant})^{\mathcal{I}} &= \emptyset \\ (\mathsf{Plant} \sqcup \exists partOf.\mathsf{Plant})^{\mathcal{I}} &= \{\texttt{grass1}\} \\ (\forall eat.(\mathsf{Plant} \sqcup \exists partOf.\mathsf{Plant}))^{\mathcal{I}} &= \{\texttt{Ganesh, Balavan,grass1,stone1}\} \end{aligned}$$

Note that `Balavan,grass1,stone1` do not relate to anything via the role eat, according to the semantics of the value restriction (see Table 2.1), all of them are instances of the concept $\forall eat.(\mathsf{Plant} \sqcup \exists partOf.\mathsf{Plant})$. ♢

In addition to the constructors presented in Table 2.1, $\mathcal{S}$ provides transitive role axioms, which will be introduced in the next section.

2.1.2 Knowledge Base

Typically, a DL knowledge base is composed of two distinct parts: the *intensional knowledge* (TBox and RBox), i.e., general knowledge about the problem domain, and *extensional knowledge* (ABox), i.e., knowledge about a specific situation.

TBox

Intuitively, a T(erminological)Box is a set of statements about how concepts are related to each other. For example,

$$\exists hasStudent.\mathsf{PhDStudent} \sqsubseteq \mathsf{AcademicStaff} \sqcap \mathsf{FullTimeStaff},$$

says only full time AcademicStaff can have PhDStudents.

Formally, a *TBox* is defined as follows.

Definition 2.2. (**TBox**) Let $\mathcal{L}$ be a Description Logic, $C, D \in \mathbf{Cdsc}(\mathcal{L})$ $\mathcal{L}$-concepts, a TBox $\mathcal{T}$ is a finite, possibly empty, set of statements of the form $C \sqsubseteq D$, called *concept*

inclusions. $C \equiv D$, called a *concept equivalence*, is an abbreviation for $C \sqsubseteq D$ and $D \sqsubseteq C$. Statements in $\mathcal{T}$ are called *terminological axioms*.

An interpretation $\mathcal{I}$ *satisfies* a concept inclusion $C \sqsubseteq D$, or $\mathcal{I}$ *models* $C \sqsubseteq D$ (written as $\mathcal{I} \models C \sqsubseteq D$), if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and it satisfies a concept equivalence $C \equiv D$ (written as $\mathcal{I} \models C \equiv D$), if $C^{\mathcal{I}} = D^{\mathcal{I}}$. An interpretation $\mathcal{I}$ *satisfies* a TBox $\mathcal{T}$ (written as $\mathcal{I} \models \mathcal{T}$), iff it satisfies all the terminological axioms in $\mathcal{T}$. $\diamond$

Reasoning in the presence of a TBox is much harder than that without a TBox [103], especially when terminological cycles are allowed. A *terminological cycle* in a TBox is a recursive concept inclusion, e.g., $\mathsf{Person} \sqsubseteq \forall hadParent.\mathsf{Person}$, or one or more mutually recursive concept inclusions, e.g.,

$$\{\mathsf{Person} \sqsubseteq \exists hasParent.\mathsf{Mother}, \mathsf{Mother} \sqsubseteq \exists hasChild.\mathsf{Person}\}.$$

Note that, however, for concept names $\mathsf{A}, \mathsf{B} \in \mathbf{C}$, $\{\mathsf{A} \sqsubseteq \mathsf{B}, \mathsf{B} \sqsubseteq \mathsf{A}\}$ is *not* a terminological cycle, since it is equivalent to a concept equivalence $\mathsf{A} \equiv \mathsf{B}$.

It can be shown that a knowledge base without such cycles and containing only *unique introductions*, i.e., terminological axioms with only concept names appearing on the left hand side and, for each concept name CN, there is at most one axiom in $\mathcal{T}$ of which CN appears on the left side, can be transformed into an equivalent one with an empty terminology by, roughly speaking, substituting all the introduced concept names in concept descriptions with their unique equivalence introductions; this transformation is called *unfolding* [138].

Although terminological cycles in a TBox present computational, algorithmic and even semantic problems [101],[2] experiences with terminological knowledge representation systems in applications show that terminological cycles are used regularly [79].

It has already been shown that we can handle general terminologies, or TBoxes (possibly) with terminological cycles, theoretically and practically. For DLs with the disjunction ($\mathcal{U}$) and general negation($\mathcal{C}$) constructors, general terminologies can be transformed into (possibly cyclical) terminologies with only introductions [24], which can be further transposed into meta constraints [58] in tableaux algorithms (cf. Section 2.2). If such DLs also provide, e.g., transitive closure, or both transitive role

[2]Following [101], the semantics we have studied so far is called *descriptive* semantics. Even though it can produce counter-intuitive results when the terminology contains cycles [101, 58], while the so-called *fixpoint* semantics will not [101, 34], the descriptive semantics is adopted in this thesis because of its wide acceptance as the most appropriate one [24, 39], and because existence of fixpoint cannot be guaranteed for the expressive languages studied here.

axioms and role inclusion axioms (see Definition 2.3), general terminologies can be 'internalised' [65] (see Theorem 2.8), so that reasoning w.r.t. the TBox can be reduced to pure concept reasoning. The extensive use of the disjunctive constructor in the above processes inevitably leads to intractability, which can be overcome, to some extent, by the optimisation technique called *absorption* that reduces the number of concept inclusion axioms by absorbing them into inclusion definitions whenever possible [58]. As a result of the above theoretical and empirical investigations, all modern DL systems provide unrestricted concept inclusion axioms.

RBox

Intuitively, an R(ole)Box is a set of statements about the characteristics of roles. An RBox can include statements asserting that a role is functional (e.g., since one can have at most one father, we can hence say the role $hasFather$ is functional) or transitive (e.g., the role $partOf$ is transitive), or is in inclusions relationships (e.g., $isComponent \sqsubseteq partOf$). An *RBox* is formally defined as follows.

Definition 2.3. (**RBox**) Let $\mathcal{L}$ be a Description Logic, $RN, SN \in \mathbf{R}$ role names, $R_1, R_2 \in \mathbf{Rdsc}(\mathcal{L})$ $\mathcal{L}$-roles, an RBox $\mathcal{R}$ is a finite, possibly empty, set of statements of the form:

- $\mathsf{Func}(RN)$ or $RN \in \mathbf{F}$, where $\mathbf{F} \subseteq \mathbf{R}$ is a set of *functional roles*, or
- $\mathsf{Trans}(SN)$ or $SN \in \mathbf{R}_+$, where $\mathbf{R}_+ \subseteq \mathbf{R}$ is a set of *transitive roles*, or
- $R_1 \sqsubseteq R_2$, called *role inclusions*; $R_1 \equiv R_2$, called *role equivalences*, are an abbreviation for $R_1 \sqsubseteq R_2$ and $R_2 \sqsubseteq R_1$.

Statements in $\mathcal{R}$ are called *role axioms*. The kinds of role axioms that can appear in $\mathcal{R}$ depend on the expressiveness of $\mathcal{L}$.

An interpretation $\mathcal{I}$ *satisfies* a functional role axiom $RN \in \mathbf{F}$ (written as $\mathcal{I} \models RN \in \mathbf{F}$), if, for all $x \in \Delta^{\mathcal{I}}$, $\sharp\{y \in \Delta^{\mathcal{I}} \mid \langle x, y\rangle \in RN^{\mathcal{I}}\} \leq 1$ ($\sharp$ denotes cardinality). An interpretation $\mathcal{I}$ *satisfies* a transitive role axiom $SN \in \mathbf{R}_+$ (written as $\mathcal{I} \models SN \in \mathbf{R}_+$), if, for all $x, y, z \in \Delta^{\mathcal{I}}$, $\{\langle x, y\rangle, \langle y, z\rangle\} \subseteq SN^{\mathcal{I}} \rightarrow \langle x, z\rangle \in SN^{\mathcal{I}}$. An interpretation $\mathcal{I}$ *satisfies* a role inclusion $R_1 \sqsubseteq R_2$ (written as $\mathcal{I} \models R_1 \sqsubseteq R_2$), if $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$, and it satisfies a role equivalence $R_1 \equiv R_2$ (written as $\mathcal{I} \models R_1 \equiv R_2$), if $R_1^{\mathcal{I}} = R_2^{\mathcal{I}}$. An interpretation $\mathcal{I}$ *satisfies* an RBox $\mathcal{R}$ (written as $\mathcal{I} \models \mathcal{R}$), iff it satisfies all the role axioms in $\mathcal{R}$. $\diamond$

Many DLs, e.g., $\mathcal{ALC}$, do not provide any role axioms at all. The RBox is, therefore, usually disregarded as a part of a DL knowledge base.[3] For $\mathcal{S}$-family DLs, however, the RBox is a very important component in a DL knowledge base, since $\mathcal{S}$ itself provides transitive role axioms.

Like constructors, each kind of role axiom is associated with a special letter, i.e., *f* for functional role axioms, $_{\mathcal{R}^+}$ for transitive role axioms and $\mathcal{H}$ for role inclusion axioms. We can extend the $\mathcal{S}$ DL, which is also called $\mathcal{ALC}_{R^+}$, with role inclusion axioms to have the $\mathcal{SH}$ DL, which can be further extended to $\mathcal{SH}f$ by also providing functional role axioms.

Interestingly, there are some restrictions and interactions among the three kinds of role axioms. Most importantly, the set of functional roles ($\mathbf{F}$) and the set of transitive roles ($\mathbf{R}_+$) should be disjoint. The reason for this restriction is that it is still an open problem whether DLs with transitive functional roles are decidable or not.[4] With the presence of role inclusions, e.g., for $RN, SN \in \mathbf{R}$, if $RN \sqsubseteq SN$ and SN is functional, then obviously RN must be functional as well; RN can not, therefore, be a transitive role.

ABox

The third component of a DL knowledge base is the A(ssertional)Box, or *world description*, where one describes a specific state of affairs, w.r.t. some individuals, of an application domain in terms of concepts and roles. An *ABox* is formally defined as follows.

Definition 2.4. (**ABox**) Let $\mathcal{L}$ be a Description Logic, $\mathtt{a}, \mathtt{b} \in \mathbf{I}$ individual names, $C \in \mathbf{Cdsc}(\mathcal{L})$ an $\mathcal{L}$-concept and $R \in \mathbf{Rdsc}(\mathcal{L})$ an $\mathcal{L}$-role. An ABox $\mathcal{A}$ is a finite, possibly empty, set of statements of the form $\mathtt{a} : C$, called *concept assertions*, or $\langle \mathtt{a}, \mathtt{b} \rangle : R$, called *role assertions*. Statements in $\mathcal{A}$ are called *assertions* (or *individual axioms*).

An interpretation $\mathcal{I}$ *satisfies* a concept assertion $\mathtt{a} : C$ (written as $\mathcal{I} \models \mathtt{a} : C$) if $\mathtt{a}^{\mathcal{I}} \in C^{\mathcal{I}}$, and it satisfies a role assertion $\langle \mathtt{a}, \mathtt{b} \rangle : R$ (written as $\mathcal{I} \models \langle \mathtt{a}, \mathtt{b} \rangle : R$) if $\langle \mathtt{a}^{\mathcal{I}}, \mathtt{b}^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$. An interpretation $\mathcal{I}$ *satisfies* an ABox $\mathcal{A}$ (written as $\mathcal{I} \models \mathcal{A}$), iff it satisfies all the assertions in $\mathcal{A}$. $\diamond$

[3] Sometimes role axioms are regarded as parts of the TBox, which is not quite proper, since when we internalise a TBox, we do not internalise any role axioms.

[4] Allowing transitive roles in number restriction leads to undecidability [66], and a functional role $RN \in \mathbf{F}$ can be encoded by a concept inclusion using a number restriction, i.e., $\top \sqsubseteq \leqslant 1 RN.\top$.

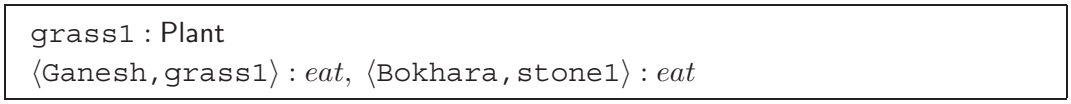

Figure 2.1: A world description (ABox)

Figure 2.1 shows an example of an ABox. The interpretation $\mathcal{I}$ given in Example 2.2 on page 29 is a model of the above ABox.

There are two common assumptions about ABoxes. The first one is the so-called *unique name assumption* (UNA), i.e., if $\mathtt{a},\mathtt{b} \in \mathbf{I}$ are distinct individual names, then $\mathtt{a}^{\mathcal{I}} \neq \mathtt{b}^{\mathcal{I}}$. Without the UNA, we may need an explicit assertion to assure that $\mathtt{a},\mathtt{b}$ are different individuals. Note that we do not need a new form of assertion though, as we can use, e.g., $\mathtt{a} : (=1\, indentity), \mathtt{b} : (=2\, indentity)$, where $indentity$ is a new role name that is not used in the knowledge base, and $(=n\, indentity)$ is a concept, each instance of which relates to exactly n different individuals via the role $indentity$. Even though it is not necessary, a special form of assertion may, however, be useful and convenient.[5]

The second assumption is called the *open world assumption*, i.e., one cannot assume that the knowledge in the knowledge base is complete. This is inherent in the fact that an ABox (or a knowledge base in general) may have many models, only some aspects of which are constrained by the assertions. For example, the role assertion $\langle \mathtt{Ian},\mathtt{Jeff} \rangle : hasPhDStudent$ expresses that Ian has a PhD student Jeff in all models; in some of these models, Jeff is the only one PhD student of Ian, while in others, Ian may have some other PhD students.

Interestingly, individual names can be used not only in an ABox, but also in the description language, where they are used to form concepts called *nominals*, i.e., concepts interpreted as sets consisting of exactly one individual. Nominal is also used to refer to the singleton set constructor (indicated by $\mathcal{O}$ in the name of a description language), written as $\{\mathtt{a}\}$, where $\mathtt{a} \in \mathbf{I}$ is an individual name. As one would expect, such a nominal is interpreted as $\{\mathtt{a}\}^{\mathcal{I}} = \{\mathtt{a}^{\mathcal{I}}\}$.

Knowledge Base

Having discussed all its three components, let us now formally define a DL *knowledge base* as follows.

[5] For example, the OWL language to be introduced in Chapter 3 provides such a form of assertion.

Definition 2.5. (**Knowledge Base**) A knowledge base Σ is a triple $\langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, where $\mathcal{T}$ is a TBox, $\mathcal{R}$ is an RBox, and $\mathcal{A}$ is an ABox.

An interpretation $\mathcal{I}$ *satisfies* a knowledge base Σ, written as $\mathcal{I} \models \Sigma$, iff it satisfies $\mathcal{T}$, $\mathcal{R}$ and $\mathcal{A}$; Σ is *satisfiable* (*unsatisfiable*), written as $\Sigma \not\models \bot$ ($\Sigma \models \bot$), iff there exists (does not exist) such an interpretation $\mathcal{I}$ that satisfies Σ.

Given a terminological axiom, a role axiom, or an assertion φ, Σ *entails* φ, written as $\Sigma \models \varphi$, iff for all models $\mathcal{I}$ of Σ we have $\mathcal{I} \models \varphi$. A knowledge base Σ *entails* a knowledge base Σ', written as $\Sigma \models \Sigma'$, iff for all models $\mathcal{I}$ of Σ we have $\mathcal{I} \models \Sigma'$. Two knowledge bases Σ and Σ' are *equivalent*, written as $\Sigma \equiv \Sigma'$, iff $\Sigma \models \Sigma'$ and $\Sigma' \models \Sigma$. $\diamond$

Knowledge base entailment and equivalence are very powerful tools: the former one will be used to describe DL reasoning services in the next section, and the latter one shows how to modify a knowledge base without affecting its logical meaning.

2.1.3 Reasoning Services

A DL system not only stores axioms and assertions, but also offers *services* that *reason* about them. Typically, reasoning with a DL knowledge base is a process of discovering implicit knowledge entailed by the knowledge base. Reasoning services can be roughly categorised as basic services, which involve the checking of the truth value for a statement, and complex services, which are built upon basic ones. Let $\mathcal{L}$ be a Description Logic, Σ a knowledge base, $C, D \in \mathbf{Cdsc}(\mathcal{L})$ $\mathcal{L}$-concepts and $\mathtt{a} \in \mathbf{I}$ an individual name, principle basic reasoning services include:

Knowledge Base Satisfiability is the problem of checking if $\Sigma \not\models \bot$ holds, i.e., whether there exists a model $\mathcal{I}$ of Σ.

Concept Satisfiability is the problem of checking if $\Sigma \not\models C \equiv \bot$ holds, i.e., whether there exists a model $\mathcal{I}$ of Σ in which $C^{\mathcal{I}} \neq \emptyset$.

Subsumption is the problem of verifying if $\Sigma \models C \sqsubseteq D$ holds, i.e., whether in every model $\mathcal{I}$ of Σ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

Instance Checking is the problem of verifying if $\Sigma \models \mathtt{a} : C$ holds, i.e., whether in every model $\mathcal{I}$ of Σ we have $\mathtt{a}^{\mathcal{I}} \in C^{\mathcal{I}}$.

The above principle basic reasoning services are not independent of each other. If $\mathcal{L}$ is closed under negation, i.e., the complement of any $\mathcal{L}$-concept is also an $\mathcal{L}$-concept, then all the basic reasoning services are reducible to knowledge base satisfiability [128].

In the thesis, we mainly consider concept satisfiability and subsumption services. Given that the DLs to be investigated are closed under negation, these two services are reducible to each other (C is subsumed by D, iff $C \sqcap \neg D$ is unsatisfiable; C is unsatisfiable, iff C is subsumed by $\bot$),[6] and the undecidability of these services would imply that none of the above basic reasoning services are computable.

These two reasoning services can be transformed into reasoning with an empty TBox and ABox [2], which is usually accomplished in two steps:

1. to eliminate the ABox;

2. to eliminate the TBox.

The first step is based on an important characteristic of DLs, i.e., TBox reasoning is usually not influenced by ABox reasoning. More precisely, we have the following theorem (with a slightly different presentation).

Theorem 2.6. ***(Nebel [102], Schaerf [128])*** *If $\mathcal{L}$ is a DL that does not provide the nominal constructor,[7] and $\Sigma = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ is a satisfiable knowledge base, then for every pair of $\mathcal{L}$-concepts $C, D \in \mathbf{Cdsc}(\mathcal{L})$, we have*

$$\langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle \models C \sqsubseteq D \text{ iff } \langle \mathcal{T}, \mathcal{R}, \{\} \rangle \models C \sqsubseteq D.$$

For DLs providing the nominal constructor, the above theorem does not apply. We can, however, encode the assertions in the ABox into concept inclusions in the TBox [128, 139].

Theorem 2.7. ***(Tobies [139, Lemma 5.3])*** *If $\mathcal{L}$ is a DL that provides the nominal constructor, knowledge base satisfiability can be polynomially reduced to satisfiability of TBoxes and RBoxes.*

For the second step, as mentioned in Section 2.1.2, we can either unfold or internalise a TBox. The following theorem addresses the internalisation of a TBox.

[6] This is easy to understand, if one recalls that, for set A, B, we have $A \subseteq B$ iff $A \setminus B = \emptyset$.

[7] Nominal is the singleton set constructor; cf. page 33.

Theorem 2.8. *(**Horrocks et al. [65]**) Let $\mathcal{L}$ be a DL providing general negation, transitive role and role inclusion axioms, $\mathcal{T}$ a TBox and $C, D \in \mathbf{Cdsc}(\mathcal{L})$ $\mathcal{L}$-concepts, $C_{\mathcal{T}} := \bigsqcap_{C_i \sqsubseteq D_i \in \mathcal{T}} \neg C_i \sqcup D_i$ and U be a transitive role with $R \sqsubseteq U$ for each role R that occurs in $\mathcal{T}$, C, or D.*

C is satisfiable w.r.t. $\mathcal{T}$ iff $C \sqcap C_{\mathcal{T}} \sqcap \forall U.C_{\mathcal{T}}$ is satisfiable. D subsumes C w.r.t. $\mathcal{T}$ (written as $C \sqsubseteq_{\mathcal{T}} D$) iff $C \sqcap \neg D \sqcap C_{\mathcal{T}} \sqcap \forall U.C_{\mathcal{T}}$ is unsatisfiable.

The most common complex services include classification and retrieval. Classification is a problem of putting a new concept in the proper place in a taxonomic hierarchy of *concept names*; this can be done by subsumption checking between each named concept in the hierarchy and the new concept. The location of the new concept, let us call it C, in the hierarchy will be between the most specific named concepts that subsume C and the most general named concepts that C subsumes. Retrieval (or query answering) is a problem of determining the set of individuals that instantiate a given concept; this can be done by instance checking between each named individual and the given concept.

After introducing some basic and complex reasoning services, we now briefly describe the complexity of reasoning services, which has been one of the main issues in DL research. Initially, the emphasis was on the reasoning services of *tractable* DLs, with an upper bound of polynomial complexity [19]. Unfortunately, only very primitive DLs are tractable in this sense, e.g., the satisfiability of $\mathcal{ALC}$-concepts w.r.t. general TBox is already EXPTIME-complete (Schild [129, 130], Lutz [88]). Interestingly, although the theoretical complexity results are discouraging, empirical analysis have shown that worse-case intractability rarely occur in practice [103, 54, 135, 61]; even some simple optimisation techniques could lead to significant improvement in the empirical performance of a DL system (Baader et al. [4]). More recently the FaCT (Horrocks [59]), RACER(Haarslev and Möller [51]), and DLP (Patel-Schneider [115]) systems have demonstrated that, even with expressive DLs, highly optimised implementations can provide acceptable performance in realistic applications. In other words, thoughtful optimisation techniques (Horrocks [58], Horrocks and Patel-Schneider [62], Horrocks and Sattler [76], Horrocks [61]) have moved the boundaries of 'tractability' to somewhere very close to EXPTIME-hard, or worse (Donini [38]).

2.2 Reasoning Algorithms

In this section, we will briefly explain how to provide DL reasoning services for concept satisfiability and subsumption problems.

There are (at least) two options here: to reuse existing first order logic algorithms, or to design new DL algorithms. For the first option, since most DLs[8] are within the two variable fragment of first-order predicate logic, we can reduce the two DL problems to known inference problems of first-order logics, e.g., $\mathcal{L}^2$ and $\mathcal{C}^2$, to provide DL reasoning services. The complexity of decision procedures obtained this way, however, is usually higher than necessary (Baader and Nutt [7]). This approach, therefore, is often used for obtaining upper bound complexity results of the two DL reasoning problems, rather than providing reasoning services in DL systems.[9]

For the second option, in the early days people used so-called structural subsumption algorithms, i.e., algorithms that compare the syntactic structure of concepts, to solve the concept subsumption problem of some rather primitive DLs. These algorithms, however, are not complete for DLs with (full) negation and disjunction, e.g., the $\mathcal{S}$-family DLs. For such languages, the so-called *tableaux algorithms* (first by Schmidt-Schauß and Smolka [131]) have turned out to be very useful to solve the concept satisfiability and subsumption problems [131, 56, 55, 3, 52].

Tableaux algorithms test the satisfiability of a concept D by trying to construct a model (witness) for D. A model of D is usually represented by a *completion tree* $\mathbf{T}$: nodes in $\mathbf{T}$ represent individuals in the model; each node x is labeled with $\mathcal{L}$(x), a set of sub-concepts[10] of D; each edge $\langle x, y\rangle$ is labeled with $\mathcal{L}(\langle x, y\rangle)$, a set of role names in D. A tableaux algorithm starts from the root node x_0 labeled with $\mathcal{L}(x_0) = \{D\}$. $\mathbf{T}$ is then expanded by repeatedly applying the completion rules (see Figure 2.2) that either extend the node labels or add new leaf nodes; such completion rules correspond to the logical constructors as well as role axioms provided by a particular description language.

The algorithm terminates either when $\mathbf{T}$ is *complete* (no further completion rules can be applied or when an obvious contradiction, or *clash* (see Figure 2.3), has been revealed). Non-determinism is dealt with by searching different possible expansions: the concept is unsatisfiable if every expansion leads to a contradiction and is satisfiable

[8] With a few exceptions like DLs introduced by Calvanese et al. [25], Lutz [86].

[9] It is worth pointing out that Hustadt et al. [77] have presented an EXPTIME resolution-based decision procedure to decide the $\mathcal{SHIQ}^-$-knowledge base satisfiability problem.

[10] A is a sub-concept of a concept description B, if A appears in B; e.g., C, D and $\exists R.$D are all sub-concepts of C $\sqcap$ $\exists R.$D.

$\sqcap$-rule: if 1.$C_1 \sqcap C_2 \in \mathcal{L}(x)$, and
2.$\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$,
then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-rule: if 1.$C_1 \sqcup C_2 \in \mathcal{L}(x)$, and
2.$\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$,
then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$
$\exists$-rule: if 1.$\exists R.C \in \mathcal{L}(x)$, and
2.there is no y s.t. $R \in \mathcal{L}(\langle x, y\rangle)$ and $C \in \mathcal{L}(y)$
then create a new node y with $\mathcal{L}(\langle x, y\rangle) = \{R\}$ and $\mathcal{L}(y) = \{C\}$
$\forall$-rule: if 1.$\forall R.C \in \mathcal{L}(x)$, and
2.there is some y s.t. $R \in \mathcal{L}(\langle x, y\rangle)$ and $C \notin \mathcal{L}(y)$,
then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$

Figure 2.2: The completion rules for $\mathcal{ALC}$

A node x of a completion tree $\mathbf{T}$ contains a *clash* if (at least) one of the following conditions holds:
(1) $\bot \in \mathcal{L}(x)$;
(2) for some $\mathsf{A} \in \mathbf{C}$, $\{\mathsf{A}, \neg\mathsf{A}\} \subseteq \mathcal{L}(x)$.

Figure 2.3: The clash conditions of $\mathcal{ALC}$

if any possible expansion leads to the discovery of a complete clash-free tree (Horrocks [58]).

For expressive DLs, e.g., the $\mathcal{S}$-family DLs, a technique called *blocking* [24] is used to ensure termination of the tableaux algorithms. To explain blocking, we shall introduce the terms of *predecessor* and *ancestor*: the predecessor of a node y is the node x if $\langle x, y\rangle$ is an edge in $\mathbf{T}$; ancestor is the transitive closure of predecessor. A node y is said to be *blocked* if there is some ancestor node x s.t. $\mathcal{L}(y) \subseteq \mathcal{L}(x)$; in this case, x is called the *blocking node*. The use of blocking (in the case of $\mathcal{S}$) is that once a node is blocked, the completion rules no longer apply on it. Figure 2.4 shows the completion rules for $\mathcal{S}$, i.e., $\mathcal{ALC}_{R^+}$, which also provides transitive role axioms and thus requires blocking. Note that different DLs might use different kinds of blocking techniques.

In Chapter 6, we will describes the tableaux algorithms for a family of new Description Logics that are designed to provide reasoning services for OWL, DAML+OIL and OWL-E, which is a decidable extension of OWL that provides customised datatypes and predicates. In these tableaux algorithms, models can not only be represented by trees, but also forests; the blocking techniques used in them are a bit more complex

$\sqcap$-rule:	if 1.$C_1 \sqcap C_2 \in \mathcal{L}(x)$ and x is not blocked, and 2.$\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$, then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-rule:	if 1.$C_1 \sqcup C_2 \in \mathcal{L}(x)$ and x is not blocked, and 2.$\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$, then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$
$\exists$-rule:	if 1.$\exists R.C \in \mathcal{L}(x)$ and x is not blocked, and 2.there is no y s.t. $R \in \mathcal{L}(\langle x, y\rangle)$ and $C \in \mathcal{L}(y)$ then create a new node y with $\mathcal{L}(\langle x, y\rangle) = \{R\}$ and $\mathcal{L}(y) = \{C\}$
$\forall$-rule:	if 1.$\forall R.C \in \mathcal{L}(x)$ and x is not blocked, and 2.there is some y s.t. $R \in \mathcal{L}(\langle x, y\rangle)$ and $C \notin \mathcal{L}(y)$, then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$
$\forall_+$-rule:	if 1.$\forall R.C \in \mathcal{L}(x)$ and x is not blocked, and 2.$R \in \mathbf{R}_+$, and 3.there is some y s.t. $R \in \mathcal{L}(\langle x, y\rangle)$ and $\forall R.C \notin \mathcal{L}(y)$, then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\forall R.C\}$

Figure 2.4: The completion rules for $\mathcal{S}$

than the one we introduced in this section.

2.3 Description Logics and Datatype Predicates

A major limitation of many DLs, such as the $\mathcal{S}$ DL, is that datatypes (such as integers and strings), data values and datatype predicates cannot be adequately represented. Without them, we have to use concepts to represent datatype constraints. For example, if we want to describe Elephants younger than 10 years old, we have to use concepts, such as YoungerThan10:

$$\mathsf{Elephant} \sqcap \mathsf{YoungerThan10}.$$

The approach, however, is *not* satisfactory at all, since it does not really describe what we meant. We can make the following assertion for an instance `Ganesh` of the above concept:

$$\texttt{Ganesh} : \mathsf{23YearsOld},$$

and it does not imply any contradiction.

2.3.1 The Concrete Domain Approach

It was Baader and Hanschke [8] who first presented a rigorous treatment of datatype predicates within so called 'concrete domains'.

Intuitively, a concrete domain provides a domain and a set of *n-ary* predicates (such as '$<$') with obvious (fixed) extensions over the domain. A *concrete domain* is formally defined as followed:

Definition 2.9. (**Concrete Domain**) (**Lutz [88]**) A concrete domain $\mathcal{D}$ consists of a pair ($\Delta_\mathcal{D}$,$\Phi_\mathcal{D}$), where $\Delta_\mathcal{D}$ is the domain of $\mathcal{D}$ and $\Phi_\mathcal{D}$ is a set of predicate names. Each predicate name p is associated with an arity n, and an n-ary predicate $p^\mathcal{D} \subseteq \Delta_\mathcal{D}^n$. Let **V** be a set of variables. A predicate conjunction of the form

$$c = \bigwedge_{j=1}^{k} p_j(v_1^{(j)}, \ldots, v_{n_j}^{(j)}), \tag{2.1}$$

where p_j is an n_j-ary predicate and the $v_i^{(j)}$ are variables from **V**, is called *satisfiable* iff there exists a function δ mapping the variables in c to data values in $\Delta_\mathcal{D}$ s.t. $(\delta(v_1^{(j)}), \ldots, \delta(v_{n_j}^{(j)})) \in p_j^\mathcal{D}$ for $1 \leq j \leq k$. Such a function δ is called a *solution* for c. A concrete domain $\mathcal{D}$ is called *admissible* iff

1. $\Phi_\mathcal{D}$ contains a name $\top_\mathcal{D}$ for $\Delta_\mathcal{D}$ and if $p \in \Phi_\mathcal{D}$, then $\overline{p} \in \Phi_\mathcal{D}$, and
2. the satisfiability problem for finite conjunctions of predicates is decidable.

By $\overline{p}$, we denote the name for the negation of the predicate p; i.e., if the arity of p is n, then $\overline{p}^\mathcal{D} = \Delta_\mathcal{D}^n \setminus p^\mathcal{D}$. ◇

To illustrate concrete domains, we present some numerical concrete domains as follows. As for non-numerical concrete domains, readers are referred to, e.g., the temporal and spacial concrete domains discussed in Lutz [88, 89].

Example 2.3 The Concrete Domain $\mathcal{N} = (\Delta_\mathcal{N}, \Phi_\mathcal{N})$, where $\Delta_\mathcal{N}$ is the set of non-negative integers and $\Phi_\mathcal{N} := \{\geq, \geq_n\}$.[11] $\mathcal{N}$ is not admissible since $\Phi_\mathcal{N}$ does not satisfy condition 1 of admissibility in Definition 2.9; in order to make it admissible, we would have to extend $\Phi_\mathcal{N}$ to $\{\top_\mathcal{N}, \bot_\mathcal{N}\} \cup \{<, \geq, <_n, \geq_n\}$. $\mathcal{N}$ is admissible as there exists a polynomial time algorithm to decide the satisfiability problem of predicate conjunctions over $\mathcal{N}$ (Lutz [88, Sec. 2.4.1]).

[11] Note that $\geq$ is a binary predicate and $\geq_n$ is a unary predicate.

It can be argued that the concrete domain approach is not user-friendly in that it disallows negated predicates used in predicate conjunctions (2.1). For example, it disallows users to use 'not less than' ($\overline{<}$) and forces them to use 'greater than or equal to' ($\geq$). ♢

Example 2.4 The Concrete Domain $\mathbb{Q}=(\Delta_{\mathbb{Q}}, \Phi_{\mathbb{Q}})$, where $\Delta_{\mathbb{Q}}$ is the set of rational numbers and $\Phi_{\mathbb{Q}}$ is $\{\top_{\mathbb{Q}}, \bot_{\mathbb{Q}}\} \cup \{<_n, \geq_n, =_n, \neq_n, >_n, \leq_n, <, \geq, =, \neq, >, \leq, +, \overline{+}, int, \overline{int}\}$. Note that $+$ is a ternary predicate

$$+^Q = \{\langle t_1, t_2, t_3\rangle \in \mathbb{Q}^3 \mid t_1 + t_2 = t_3\},$$

`int` is a unary predicate

$$\texttt{int}^Q = \mathbb{Z} \text{ (where } \mathbb{Z} \text{ denotes the integers)}$$

and $\overline{+}$, $\overline{\texttt{int}}$ are the negations of $+$ and `int`, respectively.

It has been proved that $\mathbb{Q}$ is admissible (Lutz [90, p.29-30]). If we extend $\mathbb{Q}$, however, with the unary ternary predicate $*$ where

$$*^Q = \{\langle t_1, t_2, t_3\rangle \in \mathbb{Q}^3 \mid t_1 \cdot t_2 = t_3\},$$

and its negation $\overline{*}$, $\mathbb{Q} +$ '$*$' becomes undecidable because it can easily be proved by using a reduction of Hilbert's 10-th problem (Lutz [89, Sec. 4]).

Again, it can be argued that the concrete domain approach is not user-friendly in that it does not support predicates with unfixed arities, such as the predicate

$$+'^Q = \{\langle t_1, \ldots, t_n\rangle \in \mathbb{Q}^n \mid t_1 + \ldots + t_{n-1} = t_n\},$$

which can easily represent the constraint $x_1 + x_2 + x_3 = x_4$ as $+'(x_1, x_2, x_3, x_4)$, while for the + predicate there is no way to represent the above constraint without the help of concept languages. ♢

In the concrete domain approach, there is a very useful result about the combination of admissible concrete domains. That is, if $\mathcal{D}_1$ and $\mathcal{D}_2$ are admissible concrete domains with $\Delta_{\mathcal{D}1}$ being disjoint with $\Delta_{\mathcal{D}2}$, then a new concrete domain $\mathcal{D}_1 \oplus \mathcal{D}_2$ is also admissible [8]. This indicates that it will not be a restriction that we integrate only one concrete domain into a description language in the next section.

The $\mathcal{ALC}(\mathcal{D})$ DL

Baader and Hanschke [8] showed that we can integrate an arbitrary admissible concrete domain $\mathcal{D}$ into the $\mathcal{ALC}$ DL and give the $\mathcal{ALC}(\mathcal{D})$ DL, with the abstract domain ($\Delta^{\mathcal{I}}$) being disjoint with the domain of $\mathcal{D}$ ($\Delta_{\mathcal{D}}$). The combination of a concrete domain into

DLs is achieved by adding datatype-related constructors and *features*, or functional roles, in the description languages, and allowing the use of predicates defined in the concrete domain with these constructors and features. For example, Elephants younger than 10 years old can be described using the concept

$$\mathsf{Elephant} \sqcap \exists age.\ <_{10},$$

where age is a feature, $<_{10}$ is a unary integer predicate, and $\exists age.\ <_{10}$ is an application of the datatype-related constructor called a *predicate exists restriction*.

The $\mathcal{ALC}(\mathcal{D})$*-roles and feature chains* are formally defined as follows.

Definition 2.10. **($\mathcal{ALC}(\mathcal{D})$-roles)** Let $RN \in \mathbf{R}$ be a role name, $f_1, \ldots, f_n \in \mathbf{F}$ functional role names, $\mathcal{D}$ a concrete domain and $R \in \mathbf{Rdsc}(\mathcal{ALC}(\mathcal{D}))$ an $\mathcal{ALC}(\mathcal{D})$-role, valid $\mathcal{ALC}(\mathcal{D})$-roles are defined by the abstract syntax:

$$R ::= RN \mid f_1 \circ \cdots \circ f_n,$$

where $R = f_1 \circ \cdots \circ f_n$ is called a chain of functional roles, or a *feature chain*. The interpretation function is extended to give semantics to feature chains as follows:

$$\begin{aligned}(f_1 \circ \cdots \circ f_n)^{\mathcal{I}} = \quad & \{\langle a_0, a_n\rangle \in \Delta^{\mathcal{I}} \times (\Delta^{\mathcal{I}} \cup \Delta_{\mathcal{D}}) \mid \exists a_1, \ldots, a_{n-1} \in \Delta^{\mathcal{I}}. \\ & f_1^{\mathcal{I}}(a_0) = a_1 \wedge \ldots \wedge f_n^{\mathcal{I}}(a_{n-1}) = a_n\}.\end{aligned}$$

◇

Obviously, the above definition indicates that $\mathcal{ALC}(\mathcal{D})$ provides functional role axioms. Note that in $\mathcal{ALC}(\mathcal{D})$, a feature chain connects an individual to *either* an individual *or* a data value.[12]

Now we formally define the $\mathcal{ALC}(\mathcal{D})$-concepts as follows.

Definition 2.11. **($\mathcal{ALC}(\mathcal{D})$-concepts)** Let $\mathsf{CN} \in \mathbf{C}$ be an atomic concept name, $R \in \mathbf{Rdsc}(\mathcal{ALC}(\mathcal{D}))$ a role name, $u_1, \ldots, u_n \in \mathbf{Rdsc}(\mathcal{ALC}(\mathcal{D}))$ $\mathcal{ALC}(\mathcal{D})$ feature chains, $C, D \in \mathbf{Cdsc}(\mathcal{ALC}(\mathcal{D}))$ $\mathcal{ALC}(\mathcal{D})$-concepts, $\mathcal{D}$ an admissible concrete domain and $p \in \Phi_{\mathcal{D}}$ a n-ary predicate name, valid $\mathcal{ALC}(\mathcal{D})$-concepts are defined by the abstract syntax:

$$C ::= \top \mid \bot \mid \mathsf{CN} \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C \mid \exists u_1, \ldots, u_n.p,$$

[12] Lutz [88] gives a slightly different definition on $\mathcal{ALC}(\mathcal{D})$ feature chains, where feature chains are disjointly divided into abstract feature chains and concrete feature chains, which allows a clearer algorithmic treatment and clearer proofs.

where $\exists u_1, \ldots, u_n.p$ is called *predicate exists restriction*.

The interpretation function can be extended to give semantics to predicate exists restriction as follows.

$$(\exists u_1, \ldots, u_n.p)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists t_1, \ldots, t_n \in \Delta_{\mathcal{D}}. \\ u_1^{\mathcal{I}}(x) = t_1 \wedge \ldots \wedge u_n^{\mathcal{I}}(x) = t_n \wedge \langle t_1, \ldots, t_n \rangle \in p^{\mathcal{D}}\}.$$

The semantics of the rest of the $\mathcal{ALC}(\mathcal{D})$ constructors can be found in Table 2.1 $\diamond$

Example 2.5 An $\mathcal{ALC}(\mathcal{D})$-Concept The concept Elephant $\sqcap \exists age.\ <_{10}$ that we gave earlier is an $\mathcal{ALC}(\mathcal{D})$-concept. Here we show why it is enough to express 'Elephant younger than 10 years old'.

The predicate exists restriction requires, informally speaking, each instance of the concept must have an age, and the age should satisfy $<_{10}$. Additionally, age is a functional role (feature). Therefore, there is only one age for each instance of the concept, and this age must be less than 10. This means that if there is an individual `Ganesh` that is an instance of the above concept, the following assertion

$$\texttt{Ganesh} : \exists age.\ =_{23}$$

will imply a contradiction, since $\langle =_{23} \rangle^{\mathcal{D}} \cap \langle <_{10} \rangle^{\mathcal{D}} = \emptyset$. $\diamondsuit$

Example 2.6 Feature Chains

With feature chains one can, for example, express the 'young wife' concept, i.e., all women whose husbands are even older than their fathers, as follows: Woman $\sqcap$ $\exists hasFather \circ age, hasHusband \circ age.\ <$. $\diamondsuit$

Although $\mathcal{ALC}(\mathcal{D})$ (with a admissible $\mathcal{D}$, similarly hereinafter) has been proved decidable (Baader and Hanschke [8]), Lutz [87] proved that reasoning with $\mathcal{ALC}(\mathcal{D})$ and general TBoxes is usually undecidable, even with some simple admissible concrete domains. The undecidability comes from the fact that feature chains are so expressive that one can reduce the general, undecidable PCP (Post Correspondence Problems) to $\mathcal{ALC}(A)$-concept satisfiability, where A is any arithmetic concrete domain (Lutz [88, Sec. 6.1]). The above undecidability result implies that for any DL $\mathcal{L}$ that is expressive enough to internalise a TBox, e.g., all $\mathcal{S}$-family languages that provide role inclusion axioms, $\mathcal{L}(\mathcal{D})$ is usually undecidable.

2.3.2 The Type System Approach

In order to integrate expressive DLs with datatypes (unary predicates) defined by external type systems (e.g., the XML Schema type system), Horrocks and Sattler [75] proposed the so called 'type system approach', which can be seen as a restricted form of the concrete domain approach, where a *universal concrete domain* **D** is treated as a concrete domain[13] with datatypes being unary predicates over **D**. The satisfiability problem (2.2) considered in a universal concrete domain is, therefore, much easier than that of the concrete domain approach (2.1).

Definition 2.12. (**Universal Concrete Domain**) A universal concrete domain **D** consists of a pair ($\Delta_{\mathbf{D}}$,$\Phi_{\mathbf{D}}$), where $\Phi_{\mathbf{D}}$ is a set of datatype (unary datatype predicate) names and $\Delta_{\mathbf{D}}$ is the domain of all these datatypes. Each datatype name $d \in \Phi_{\mathbf{D}}$ is associated with a datatype $d^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}$ defined in the associated type system.

Let **V** be a set of variables, a datatype conjunction of the form

$$c_1 = \bigwedge_{j=1}^{k} d_j(v^{(j)}), \tag{2.2}$$

where d_j is a (possibly negated) datatype from $\Phi_{\mathbf{D}}$ and the $v^{(j)}$ are variables from **V**, is called *satisfiable* iff there exists a function δ mapping the variables in c_1 to data values in $\Delta_{\mathbf{D}}$ s.t. $\delta(v^{(j)}) \in d_j^{\mathbf{D}}$ for $1 \leq j \leq k$. Such a function δ is called a *solution* for c_1. By $\overline{d}$, we denote the name for the negation of the datatype d, and $\overline{d}^{\mathbf{D}} = \Delta_{\mathbf{D}} \setminus d^{\mathbf{D}}$.

A set of datatypes $\Phi_{\mathbf{D}}$ is called *conforming* iff the satisfiability problem for finite (possibly negated) datatype conjunctions over $\Phi_{\mathbf{D}}$ is decidable. ◇

In the type system approach, datatype conjunctions (2.2) allow the use of negated datatypes. On the one hand, it is more user-friendly than the concrete domain approach; on the other hand, however, it can be counter-intuitive.

Example 2.7 Full Negation Can be Counter-Intuitive Let us consider the following universal concrete domain $\mathbf{D} = (\Delta_{\mathbf{D}}, \Phi_{\mathbf{D}})$, where $\Delta_{\mathbf{D}} = >_{15}^{\mathbf{D}} \cup string^{\mathbf{D}}$ and $\Phi_{\mathbf{D}} = \{>_{15}, string\}$. 'Not greater than 15' can then be represented as $\overline{>_{15}}$, which is interpreted as $\overline{>_{15}}^{\mathbf{D}} = \Delta_{\mathbf{D}} \setminus >_{15}^{\mathbf{D}} = string^{\mathbf{D}}$. In other words, any string is not greater than 15, which can be counter-intuitive. ♢

[13] In [75]'s notation, **D** refers to $\Phi_{\mathbf{D}}$ the set of datatypes. We call it $\Phi_{\mathbf{D}}$ as it corresponds to $\Phi_{\mathcal{D}}$ in Definition 2.9.

In the type system approach, datatypes are considered to be sufficiently structured by type systems, which typically define a set of primitive datatypes and provide derivation mechanisms to define new datatypes from existing ones. [75] does not further explain, however, how the derivation mechanism of a type system affects $\Phi_{\mathbf{D}}$, viz. how one defines the extensions of datatype names in $\Phi_{\mathbf{D}}$ based on the derivation mechanism of a type system.

The $\mathcal{SHOQ}(\mathbf{D})$ DL

Horrocks and Sattler [75] integrated an arbitrary conforming universal concrete domain (**D**) as well as nominals ($\mathcal{O}$) into the $\mathcal{SHQ}$ DL to give the $\mathcal{SHOQ}(\mathbf{D})$ DL, where the domain of $\mathbf{D}$, i.e., *the datatype domain* $\Delta_{\mathbf{D}}$, is disjoint with *the object domain* $\Delta^{\mathcal{I}}$ of an interpretation $\mathcal{I}$.

Let us define some notation first. Let $\mathcal{L}$ be a Description Logic, $\mathbf{R}_A$ and $\mathbf{R}_{\mathbf{D}}$ countable sets of abstract role names and concrete role names, such that $\mathbf{R}_A$ is disjoint with $\mathbf{R}_{\mathbf{D}}$ and $\mathbf{R} = \mathbf{R}_A \cup \mathbf{R}_{\mathbf{D}}$, $\mathbf{Rdsc}_A(\mathcal{L})$ and $\mathbf{Rdsc}_{\mathbf{D}}(\mathcal{L})$ a set of abstract role descriptions and a set of concrete role descriptions (in the following simply called *abstract roles* and *concrete roles*) of $\mathcal{L}$, respectively.

An $\mathcal{SHOQ}(\mathbf{D})$-role is either an abstract role name, or a datatype role name. Each abstract role $R \in \mathbf{Rdsc}_A(\mathcal{SHOQ}(\mathbf{D}))$ is interpreted as a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each concrete role $T \in \mathbf{Rdsc}_{\mathbf{D}}(\mathcal{SHOQ}(\mathbf{D}))$ is interpreted as a subset of $\Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}$. The problematic feature chains are not allowed in $\mathcal{SHOQ}(\mathbf{D})$, in order to make $\mathcal{SHOQ}(\mathbf{D})$ decidable. We cannot, thus, use $\mathcal{SHOQ}(\mathbf{D})$ to express the 'young wife' concept mentioned in Section 2.3.1 on page 42.

Now we define $\mathcal{SHOQ}(\mathbf{D})$-concepts as follows.

Definition 2.13. **($\mathcal{SHOQ}(\mathbf{D})$-concepts)** Let $\mathsf{CN} \in \mathbf{C}$ be a concept name, $R \in \mathbf{Rdsc}_A(\mathcal{SHOQ}(\mathbf{D}))$ an abstract $\mathcal{SHOQ}(\mathbf{D})$-role, $T \in \mathbf{Rdsc}_{\mathbf{D}}(\mathcal{SHOQ}(\mathbf{D}))$ a concrete $\mathcal{SHOQ}(\mathbf{D})$-role, $C, D \in \mathbf{Cdsc}(\mathcal{SHOQ}(\mathbf{D}))$ $\mathcal{SHOQ}(\mathbf{D})$-concepts, $o \in \mathbf{I}$ an individual, $\mathbf{D}$ a universal concrete domain, $d \in \Phi_{\mathbf{D}}$ a datatype, $n, m \in \mathbb{N}$ non-negative integers, valid $\mathcal{SHOQ}(\mathbf{D})$-concepts are defined by the abstract syntax:

$$\begin{aligned} C \ ::= \ & \top \mid \bot \mid \mathsf{CN} \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \{o\} \\ & \exists R.C \mid \forall R.C \mid \geqslant nR.C \mid \leqslant nR.C \mid \exists T.d \mid \forall T.d \end{aligned}$$

The semantics of $\mathcal{SHOQ}(\mathbf{D})$-concepts is given in Table 2.2 ($\sharp$ denotes cardinality). $\diamond$

Constructor Name	Syntax	Semantics
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
concept name	CN	$\mathsf{CN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
general negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
nominals	$\{\mathsf{o}\}$	$\{\mathsf{o}\}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, \sharp\{\mathsf{o}\}^{\mathcal{I}} = 1$
exist restriction	$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y.\langle x,y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
value restriction	$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y.\langle x,y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
atleast restriction	$\geqslant m R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \mid \langle x,y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq m\}$
atmost restriction	$\leqslant m R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \mid \langle x,y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq m\}$
datatype exists restriction	$\exists T.d$	$\{x \in \Delta^{\mathcal{I}} \mid \exists t.\langle x,t\rangle \in T^{\mathcal{I}} \wedge t \in d^{\mathbf{D}}\}$
datatype value restriction	$\forall T.d$	$\{x \in \Delta^{\mathcal{I}} \mid \forall t.\langle x,t\rangle \in T^{\mathcal{I}} \rightarrow t \in d^{\mathbf{D}}\}$

Table 2.2: Semantics of $\mathcal{SHOQ}(\mathbf{D})$-concepts

Example 2.8 An $\mathcal{SHOQ}(\mathbf{D})$-Concept With $\mathcal{SHOQ}(\mathbf{D})$, we can construct more expressive concepts than $\mathcal{ALC}(\mathcal{D})$, e.g., 'Items that are cheaper than 100 pounds and have at least two providers from Germany, France, or Italy':

$\mathsf{Item} \sqcap \geqslant 2 providerFrom.(\{\texttt{Germany}\} \sqcup \{\texttt{France}\} \sqcup \{\texttt{Italy}\}) \sqcap \exists priceInPounds. <_{100},$

where {Germany},{`France`} and {`Italy`} are nominals, which are interpreted as singleton sets (see Table 2.2). Note that

$$\geqslant 2 providerFrom.(\{Germany\} \sqcup \{France\} \sqcup \{Italy\})$$

is *not* a datatype constraint, but a qualified number restriction, which requires that each instance of the above concept should be related by the abstract role $providerFrom$ to at least two out of the following three individuals: `Germany`, `France`, `Italy`.

$\exists priceInPounds. <_{100}$ is a datatype exists restriction, where $<_{100}$ is a datatype defined in a type system. Note that $priceInPounds$ is a role instead of a feature, so an instance of the above concept can be related to more than one value by $priceInPounds$; in other words, some items may have two prices. This may look a bit odd in some contexts. ♢

The above example indicates that the lack of functional concrete role axioms, or in general, the lack of datatype qualified number restrictions makes $\mathcal{SHOQ}(\mathbf{D})$ less useful in some situations. Extensions of the $\mathcal{SHOQ}(\mathbf{D})$ DL with datatype qualified number restrictions will be discussed in Chapter 6; such extensions are required in

order to provide DL reasoning support for the Web ontology languages DAML+OIL and OWL (cf. Chapter 3).

Formally, an $\mathcal{SHOQ}(\mathbf{D})$-RBox is defined as follows.

Definition 2.14. **($\mathcal{SHOQ}(\mathbf{D})$ RBox)** Let $R_1, R_2 \in \mathbf{Rdsc}_A(\mathcal{SHOQ}(\mathbf{D}))$, $T_1, T_2 \in \mathbf{Rdsc}_{\mathbf{D}}(\mathcal{SHOQ}(\mathbf{D}))$, $SN, RN \in \mathbf{R}_A$, an $\mathcal{SHOQ}(\mathbf{D})$ RBox $\mathcal{R}$ is a finite, possibly empty, set of role axioms:

- functional role axioms $\mathsf{Func}(SN)$;
- transitive role axioms $\mathsf{Trans}(RN)$;
- abstract role inclusion axioms $R_1 \sqsubseteq R_2$;
- concrete role inclusion axioms $T_1 \sqsubseteq T_2$.

For a set of role inclusion axioms $\mathcal{R}_{Inc}$, $\mathcal{R}^+_{Inc} := (\mathcal{R}_{Inc}, \sqsubseteq^*)$ is called a *role hierarchy*, where $\sqsubseteq^*$ is the transitive-reflexive closure of $\sqsubseteq$ over $\mathcal{R}_{Inc}$. Given $\sqsubseteq^*$, the set of roles $R^{\downarrow} = \{S \in \mathbf{R} \mid S \sqsubseteq^* R\}$ defines the *sub-roles* of a role R. A role R is called a *super-role* of a role S if $S \in R^{\downarrow}$. A role R is called *simple* if, for each role S, $S \sqsubseteq^* R$ implies $\mathsf{Trans}(S) \notin \mathcal{R}$. ◇

There are two remarks on Definition 2.14. Firstly, only simple roles, viz. roles such that none of their sub-roles are transitive, can be used in number restrictions, since allowing transitive roles in number restrictions leads to undecidability [66]. Secondly, a concrete role T can not be a transitive role; thus, it is always a simple role.

The $\mathcal{SHOQ}(\mathbf{D})$-concept satisfiability and subsumption problems w.r.t. a TBox are proved decidable [75]. As nominals and datatypes are widely used in ontologies, and ontologies play an important role in the Semantic Web, $\mathcal{SHOQ}(\mathbf{D})$ is expected to be very useful in the Semantic Web.

2.3.3 Limitations of Existing Approaches

There are several limitations of the existing approaches that we introduced in this section:

Datatypes and Data Values Strictly speaking, the existing approaches mainly focus on datatype predicates,[14] but do not adequately represent datatypes and data values.

[14] In the type system approach, the datatypes are actually treated as unary predicates.

For example, if one wants represent 'Balavan is 1 year old', $\langle \texttt{Balavan}, 1\rangle : age$ is not enough, as '1' is a symbol can represent different data values of different datatypes (e.g., '1' can be used to represent integer 1 and boolean value $true$). A more proper way to represent the above assertion is $\langle \texttt{Balavan}, L2V(integer)(\text{“1”})\rangle : age$, where $L2V(integer)$ is the lexical-to-value mapping for the $integer$ datatype. Note that the type system approach treats datatypes as unary predicates, and it does not consider the lexical spaces and lexical-to-value mappings of datatypes (cf. Section 3.3.2).

Predicates with Unfixed Arities It is not user-friendly, for the concrete domain approach, to disallow the use of predicates with unfixed arities and to force users to represent the constraint $x_1 + x_2 + x_3 = x_4$ with the help of concept languages as follows

$$\exists x_1, x_2, x_{12}. + \sqcap \exists x_{12}, x_3, x_4.+,$$

instead of using the more natural predicate $+'$ (cf. Example 2.4 on page 41).

Negation It is not user-friendly to disallow the use of negated predicates in predicate conjunctions (2.1) in the concrete domain approach (cf. Example 2.3 on page 40). Although the type system approach allows the use of (full) negation in datatype conjunctions (2.2), it can be counter-intuitive (cf. Example 2.7 on page 44).

Customised Datatype Predicates Neither of the two approaches provide any formalism to construct customised predicates, such as 'sum no greater than 15' presented in Example 1.5 on page 22. This is because the concrete domain approach assumes that datatype constraints are sufficiently structured by $\Phi_{\mathcal{D}}$. Baader and Hanschke [8] claim that they do not want to define new classes of elements of $\Delta_{\mathcal{D}}$ (customised datatypes) or new relations between elements of $\Delta_{\mathcal{D}}$ (customised predicates) with the help of the concept language.

Chapter Achievements

- Description Logics are characterised by the use of various constructors to build concepts and roles, an emphasis on the decidability of key reasoning problems and by the provision of (empirically) tractable reasoning services.
- The concrete domain approach provides a rigorous treatment about datatype predicates, while the type system approach provides the use of unary datatype predicates defined by external type systems. Both approaches have some limitations, e.g., they do not provide any formalism to construct customised datatype predicates.

Chapter 3

The Semantic Web

Chapter Aims

- To explain why DL-based ontologies are important in the Semantic Web.
- To introduce RDF(S), OWL and their datatype formalisms.
- To clarify the two issues that the thesis is going to tackle.

Chapter Plan

3.1 Annotations and Meaning

As described in Chapter 1, the vision of the Semantic Web is to make Web resources (not just HTML pages, but a wide range of Web accessible data and services) more understandable to machines. Machine-understandable annotations are, therefore, introduced to describe the content and functions of Web resources.

3.1.1 RDF

Resource Description Framework (RDF) [81] as a W3C recommendation provides a standard *syntax* to create, exchange and use annotations in the Semantic Web. It is

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix ex:   <http://example.org/#>
@prefix elp:  <http://example.org/Animal#>

elp:Ganesh ex:mytitle "A resource called Ganesh" ;
           ex:mycreator "Pat Gregory" ;
           ex:mypublisher _:b1 .
_:b1 elp:name "Elephant United" .
```

Figure 3.1: RDF statements

built upon earlier developments such as the Dublin Core (see Section 3.1.2) and the Platform for Internet Content Selectivity (PICS) [121] content rating initiative.

An RDF statement (or RDF triple) is of the form:

[subject property object .] (3.1)

RDF-annotated resources (i.e., subjects) are usually named by Uniform Resource Identifier references. Uniform Resource Identifiers (URIs) are short strings that identify Web resources [45]. Uniform Resource Locators (URLs) are a particular type of URIs, i.e., those have network locations. A *URI reference* (or URIref) is a URI, together with an optional fragment identifier at the end. For example, the URI reference `http://www.example.org/Elephant#Ganesh` consists of the URI `http://www.example.org/Elephant` and (separated by the # character) the fragment identifier `Ganesh`. As a convention, *name spaces*, which are sources where multiple resources are from, are (usually) URIs with the # character. For example, `http://www.example.org/Elephant#` is a name space. Resources without URIrefs are called *blank nodes*; a blank node indicates the existence of a resource, without explicitly mentioning the URIref of that resource. A *blank node identifier*, which is a local identifier, can be used to allow several RDF statements to reference the same blank node. RDF annotates Web resources in terms of named properties. Values of named properties (i.e., objects) can be URIrefs of Web resources or literals, viz. representations of data values (such as integers and strings). A set of RDF statements is call an *RDF graph*.

To represent RDF statements in a machine-processable way, RDF defines a specific eXtensible Markup Language (XML) syntax, referred to as RDF/XML [94]. As RDF/XML is verbose, in this thesis, we use the Notation 3 (or N3) syntax of RDF,

where each RDF statement is of the form (3.1). Figure 3.1 shows an RDF graph in N3 syntax, where the '@prefix' introduces shorthand identifications (such as 'ex:') of XML namespaces and a semicolon ';' introduces another property of the same subject. In these statements, the annotated resource is elp:Ganesh, which is annotated with three properties ex:mytitle, ex:mycreator and ex:mypublisher. Note that `_:b1` is a blank node identifier.

Given that RDF provides only a standard syntax for annotations, how do we provide meaning to Web resources through annotations? The meaning comes either from some external agreements, e.g., Dublin Core, or from ontologies.

3.1.2 Dublin Core

One way of giving meaning to annotations is to provide some external agreement on the meaning of a set of information properties. For example, the Dublin Core Metadata Element Set [32] provides 15 'core' information properties, such as 'Title', 'Creator', 'Date', with descriptive semantic definitions (in natural language). One can use these information properties in, e.g., RDF or META tags of HTML.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix dc:   <http://purl.org/dc/elements/1.1/>
@prefix elp:  <http://example.org/Animal#>

elp:Ganesh dc:title "A resource called Ganesh" ;
           dc:creator "Pat Gregory" ;
           dc:publisher _:b1 .
_:b1 elp:name "Elephant United" .
```

Figure 3.2: Dublin Core properties in RDF statements

If we replace the properties ex:mytitle, ex:mycreator and ex:mypublisher used in Figure 3.1 with dc:title, dc:creator and dc:publisher as shown in Figure 3.2, Dublin Core compatible intelligent agents can then understand that the title of the Web resource is 'A resource called Ganesh', and the creator is Pat Gregory. This is not possible for the RDF statements in Figure 3.1 because, in general, users may use arbitrary names for the title, creator and publisher properties, etc.

The limitation of the 'external agreement' approach is its inflexibility, i.e., only a limited range of pre-defined information properties can be expressed.

3.1.3 Ontology

An alternative approach is to use ontologies to specify the meaning of Web resources. *Ontology* is a term borrowed from philosophy that refers to the science of describing the kinds of entities in the world and how they are related. In computer science, ontology is, in general, a 'representation of a shared conceptualisation' of a specific domain [46, 142]. It provides a shared and common *vocabulary*, including important concepts, properties and their definitions, and *constraints*, sometimes referred to as background assumptions regarding the intended meaning of the vocabulary, used in a domain that can be communicated between people and heterogeneous, distributed application systems.

The ontology approach is more flexible than the external agreement approach because users can customise vocabulary and constraints in ontologies. For example, applications in different domains can use different ontologies. Typically, ontologies can be used to specify the meaning of Web resources (through annotations) by asserting resources as instances of some important concepts and/or asserting resources relating to resources by some important properties defined in ontologies.

Ontologies can be expressed in Description Logics. An ontology usually corresponds to a TBox and an RBox (see Section 2.1.2) in Description Logics. Vocabulary in an ontology can be expressed by named concepts and roles, and concept definitions can be expressed by equivalence introductions. Background assumptions can be represented by general concept and role axioms. Sometimes, an ontology corresponds to a DL knowledge base. For example, in the OWL Web ontology language to be introduced in Section 3.2.2, an ontology also contains instances of important concepts and relationships among these instances, which can be represented by DL assertions.

The following example shows an ontology written in $\mathcal{SHOQ}(\mathbf{D})$ axioms and assertions (cf. Section 2.3.2 on page 45).

Example 3.1 A DL-based Ontology

A simple animal ontology may consist of three distinct parts. The first part is a set of important concepts and properties, which may include:[1]

- concepts Animal, Plant, Cow, Sheep and Elephant;
- properties eat, $partOf$, age, $liveIn$ and $weight$;

[1] As mentioned earlier, this thesis adopts the notations (i.e., font styles) of individual names (a), class names (CN) and role names (RN) introduced in Section 2.1 on page 26.

- a defined concept Herbivore, whose members are exactly those Animals s.t. everything they eat is either a Plant or is a $partOf$ a Plant: Herbivore≡Animal⊓ $\forall eat.$(Plant ⊔ $\exists partOf$.Plant);
- a defined concept AdultElephant, whose members are exactly those Elephants whose ages are greater than 20 years: AdultElephant≡Elephant⊓ $\exists age.\ >_{20}$.

The second part of the elephant ontology is composed of background assumptions of the domain and may include:

- Cow, Sheep and Elephant are Animals: Cow⊑Animal, Sheep⊑Animal, Elephant⊑Animal;
- Cows are Herbivores: Cow⊑Herbivore;
- Elephants $liveIn$ some Habitat: Elephant⊑$\exists liveIn$.Habitat;
- AdultElephants $weigh$ at least 2000 kg: AdultElephant⊑$\exists weigh.\ >_{2000}$;
- no individual can be both a Herbivore and a Carnivore: Herbivore⊑¬Carnivore;
- the property $partOf$ is transitive: Trans($partOf$).

The third part of the elephant ontology is about instances and their inter-relationships, and may include:

- `Ganesh` is an Elephant: `Ganesh`:Elephant;
- `south-sahara` is a Habitat: `south-sahara`:Habitat;
- `Ganesh` the Elephant $liveIn$ `south-sahara`: $\langle$`Ganesh`, `south-sahara`$\rangle$:$liveIn$.

♢

DL-based Ontologies can exploit powerful Description Logic reasoning tools, so as to facilitate machine understanding of Web resources. DL reasoning support can be very helpful to ensure the quality of ontologies, which is pivotal to the Semantic Web, in different development phases:

- Ontology design: DL Reasoning can be used to test whether concepts are non-contradictory and to derive implied relations. In particular, they can test whether the concepts in the ontology have their intended meaning or consequences. For example, in the animal ontology discussed in Example 3.1, one might want to

test whether elephants can be carnivores by adding a new background assumption Elephant$\sqsubseteq$Carnivore (Elephants are Carnivores). A DL reasoner should report that the concept Elephant is satisfiable (could have instances), because no relationship between Elephants and Herbivores has been specified; in other words, the report suggests that this important background knowledge is *missing* in the ontology.

- Ontology integration: Since it is not reasonable to assume that there will be a single ontology for the whole Web, interoperability and integration of different ontologies are also important issues. Integration can, for example, be supported by asserting inter-ontology relationships and testing for consistency and computing the integrated concept hierarchy. For example, one might want to integrate the above animal ontology with a mad cow ontology, asserting that the concepts Cow in the two ontologies are the same concept. If the following background assumptions are in the mad cow ontology:

  - MadCows are Cows that eat SheepBrains: MadCow$\sqsubseteq$Cow$\sqcap \exists eat$.SheepBrain,
  - SheepBrains are $partOf$ Sheep: SheepBrain$\sqsubseteq\exists partOf$.Sheep,

  a DL reasoner should report a contradiction because on the one hand, MadCows $eat\ partOf$ Sheep, thus MadCows are Carnivores, while on the other hand, MadCows are Cows, hence they cannot be Carnivores.

- Ontology deployment: Reasoning may also be used when an ontology is deployed, i.e., when a Web page is already annotated with its concepts and properties. For example, consider a Web page about `Ganesh` the elephant, with the following annotation [`Ganesh` age 23 .][2] (or $\langle$`Ganesh`$, 23\rangle$:age in DL syntax) According to the above animal ontology, a DL reasoner should conclude that Ganesh is an AdultElephant. If one asks for all the Web pages about AdultElephants (as defined by the above elephant ontology), the Web page about Ganesh should be one of the (possibly huge number of) pages returned. In the deployment phase, as one would expect, requirements on the efficiency of reasoning are much more stringent than in the design and integration phases (Baader et al. [9]).

[2]Strictly speaking, we should use the symbol for 23 here, i.e., the typed literal "23"^^xsd:integer, instead of 23 itself. For typed literals, cf. Definition 3.7 on page 71.

The importance of DL-based ontologies in the Semantic Web has inspired the development of several DL-based ontology languages specifically designed for this purpose. These include OIL, DAML+OIL and OWL [42, 68, 12], all of which provide a DL-style model theoretic semantics. The design of these languages suggests that Description Logics are an important component of knowledge representation in the Semantic Web context.

We end this section by a brief comparison of the two approaches to providing meaning to Web resources through annotations. The external agreement approach is simple and easy to use. The ontology approach is more expressive and flexible, in that users can define their vocabulary and constraints for their application domains, but requires that ontology languages should have a well-defined semantics and should have powerful reasoning support.

Note that it is not proper to use the information properties defined in Dublin Core as abstract properties (i.e., abstract roles, see Section 2.3.2) in ontologies. Otherwise, there can be unexpected restrictions or implications on the information properties. For example, if one uses $\mathrm{dc{:}author}$ as an abstract property in an ontology and there is a constraint in the ontology that an author should be a person, then it disallows anything but persons, such as organisations, to be authors. In Chapter 4, we suggest that information properties can be used as annotation properties in an ontology.

3.2 Web Ontology Languages

The main Web ontology language considered in this thesis is OWL [12], which is a W3C recommendation for expressing ontologies in the Semantic Web. The design of OWL was mainly motivated by OIL and DAML+OIL, as well as several other pre-existing languages including RDFS, SHOE and DAML-ONT.

The W3C recommendation RDFS (RDF Schema) [22] can be recognised as a simple SW ontology language. RDF and RDFS, also referred to as *RDF(S)*, are expected to be the foundation of the SW languages. There are, however, semantic problems layering OWL on top of RDF(S). In this section, we introduce RDFS and its model theory first (Section 3.1.1) and then describe the layering problems when we introduce OWL (Section 3.2.2). We will discuss datatype components of RDF(S) and OWL in Section 3.3.

3.2.1 RDFS

Syntax

Following W3C's 'one small step at a time' strategy, RDFS can be seen as a first try to support expressing simple ontologies with RDF syntax. In RDFS, predefined Web resources rdfs:Class, rdfs:Resource and rdf:Property can be used to define classes (concepts), resources and properties (roles), respectively.

Unlike Dublin Core, RDFS does not predefine information properties but a set of meta-properties that can be used to represent background assumptions in ontologies:

- rdf:type: the instance-of relationship,
- rdfs:subClassOf: the property that models the subsumption hierarchy between classes,
- rdfs:subPropertyOf: the property that models the subsumption hierarchy between properties,
- rdfs:domain: the property that constrains all instances of a particular property to describe instances of a particular class,
- rdfs:range: the property that constrains all instances of a particular property to have values that are instances of a particular class.

```
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
@prefix elp:   <http://example.org/Animal#>

elp:Animal rdf:type rdfs:Class .
elp:Habitat rdf:type rdfs:Class .
elp:Elephant rdf:type rdfs:Class ; rdfs:subClassOf elp:Animal .
elp:liveIn rdf:type rdf:Property ;
           rdfs:domain elp:Animal ; rdfs:range elp:Habitat .

elp:south-sahara rdf:type elp:Habitat .
elp:Ganesh rdf:type elp:Elephant ; elp:liveIn elp:south-sahara .
```

Figure 3.3: An RDFS ontology

RDFS statements are simply RDF triples; viz. RDFS provides no syntactic restrictions on RDF triples. Figure 3.3 shows a fragment of an animal ontology (cf. Example 3.1 on page 53) in RDFS. This fragment defines three classes, i.e., elp:Animal, elp:Habitat and elp:Elephant (which is rdfs:subClassOf elp:Animal), and a property *elp:liveIn*, the rdfs:domain and rdfs:range of which are elp:Animal and elp:Habitat, respectively. In addition, it states that the resource `elp:Ganesh` is an instance of elp:Elephant, and that it *elp:liveIn*s an elp:Habitat called `elp:south-sahara`.

At a glance, RDFS is a simple ontology langauge that supports only class and property hierarchies, as well as domain and range constraints for properties. According to the RDF Model Theory, however, it is more complicated than that.

RDF Model Theory

Initially RDF and RDFS [82] had neither a formal model theory nor any formal meaning at all. This led to disagreements about the meaning of parts of RDFS; e.g., whether multiple range and domain constraints on a single property should be interpreted conjunctively or disjunctively. Recently, however, a RDF model theory (RDF MT) [53] has been proposed, which provides semantics not only for RDFS ontologies, but also for RDF triples. It became a W3C recommendation in February 2004 and has since been called *RDF Semantics*.

Firstly, the semantics of triples (such as $[\mathrm{s\ p\ o\ .}]$) is given in terms of simple interpretations. To simplify presentation, in this thesis we do not cover blank nodes, which are identified by local identifiers instead of URIrefs.

Definition 3.1. (**Simple Interpretation**) Given a set of URI references **V**, a *simple interpretation* I of **V** in the RDF model theory is defined by

- an non-empty set **IR** of resources, called the *domain* (or *universe*) of I,
- a set **IP**, called the *set of properties* in I,
- a mapping $IEXT$, called the *extension function*, from **IP** to the powerset of $\mathbf{IR} \times \mathbf{IR}$,
- a mapping IS from URIrefs in **V** to $\mathbf{IR} \cup \mathbf{IP}$.

Given a triple $[\mathrm{s\ p\ o\ .}]$, $\mathrm{I}([\mathrm{s\ p\ o\ .}]) = true$ if s,p,o $\in \mathbf{V}$, $IS(\mathrm{p}) \in \mathbf{IP}$, and $\langle IS(\mathrm{s}), IS(\mathrm{o})\rangle \in IEXT(IS(\mathrm{p}))$; otherwise, $\mathrm{I}([\mathrm{s\ p\ o\ .}]) = false$.

Given a set of triples S, $\mathrm{I(S)} = false$ if $\mathrm{I}([\mathrm{s\ p\ o\ .}]) = false$ for some triple $[\mathrm{s\ p\ o\ .}]$ in S, otherwise $\mathrm{I(S)} = true$. I *satisfies* S, written as $\mathrm{I} \models \mathrm{S}$ if $\mathrm{I(S)} = true$; in this case, we say I is a simple interpretation of S. ◇

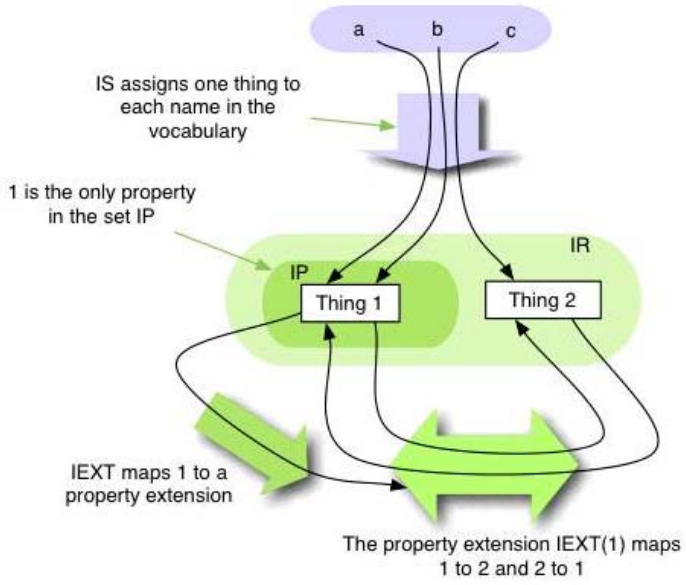

Figure 3.4: A simple interpretation of **V** = {a,b,c} (from [53])

Note that Definition 3.1 does not specify the relationship between **IP** and **IR**, i.e., **IP** may or may not be disjoint with **IR**. Figure 3.4 presents a simple interpretation I of **V** = {a,b,c}, where the URIref b is simply interpreted as a property because $IS(\mathrm{b}) = 1 \in \mathbf{IP}$, and $IEXT(IS(\mathrm{b}))$, the extension of $IS(\mathrm{b})$, is a set of pairs of resources that are in **IR**, i.e., $\{\langle 1,2\rangle,\langle 2,1\rangle\}$. Since $\langle IS(\mathrm{a}), IS(\mathrm{c})\rangle \in IEXT(IS(\mathrm{b}))$, $\mathrm{I}([\mathrm{a\ b\ c\ .}]) = true$; hence, we can conclude that I satisfies $[\mathrm{a\ b\ c\ .}]$.

Secondly, the semantics of RDF triples is given in terms of RDF-Interpretations.

Definition 3.2. (**RDF-Interpretation**) Given a set of URI references **V** and the set **rdfV**, called the *RDF vocabulary*, of URI references in the rdf: namespace, an RDF-interpretation of **V** is a simple interpretation I of **V** ∪ **rdfV** that satisfies:

1. for p∈ **V**∪**rdfV**, $IS(\mathrm{p}) \in \mathbf{IP}$ iff $\langle IS(\mathrm{p}), IS(\mathrm{rdf{:}Property})\rangle \in IEXT(IS(\mathrm{rdf{:}type}))$,
2. all the RDF axiomatic statements.[3] ◇

Condition 1 of Definition 3.2 implies that each member of **IP** is a resource in **IR**, due to the definition of $IEXT$ in Definition 3.1; in other words, RDF-interpretations require **IP** to be a subset of **IR**. RDF axiomatic statements mentioned in Condition 2 are RDF statements about RDF built-in vocabularies in **rdfV**; e.g., $[\mathrm{rdf{:}type\ rdf{:}type}$

[3]Readers are referred to [53] for the list of the RDF axiomatic statements.

$\mathrm{rdf{:}\,Property}$.] is an RDF axiomatic statement. According to Definition 3.2, any RDF-interpretation I should satisfy [$\mathrm{rdf{:}\,type\ rdf{:}\,type\ rdf{:}\,Property}$.], viz. $IS(\mathrm{rdf{:}\,type})$ should be in $\mathbf{IP}$.

Finally, the semantics of RDFS statements written in RDF triples is given in terms of RDFS-Interpretations.

Definition 3.3. (**RDFS-Interpretation**) Given $\mathbf{rdfV}$, a set of URI references $\mathbf{V}$ and the set $\mathbf{rdfsV}$, called the *RDFS vocabulary*, of URI references in the rdfs: namespace, an RDFS-interpretation I of $\mathbf{V}$ is an RDF-interpretation of $\mathbf{V} \cup \mathbf{rdfV} \cup \mathbf{rdfsV}$ which introduces

- a set $\mathbf{IC}$, called the set of classes in I, and
- a mapping $ICEXT$ (called the *class extension function*) from $\mathbf{IC}$ to the set of subsets of $\mathbf{IR}$,

and satisfies the following conditions (let x,y,u,v be URIrefs in $\mathbf{V} \cup \mathbf{rdfV} \cup \mathbf{rdfsV}$)[4]

1. $IS(\mathrm{x}) \in ICEXT(IS(\mathrm{y}))$ iff $\langle IS(\mathrm{x}), IS(\mathrm{y})\rangle \in IEXT(IS(\mathrm{rdf{:}\,type}))$,
2. $\mathbf{IC}$ = $ICEXT(IS(\mathrm{rdfs{:}\,Class}))$ and $\mathbf{IR}$ = $ICEXT(IS(\mathrm{rdfs{:}\,Resource}))$,
3. if $\langle IS(\mathrm{x}), IS(\mathrm{y})\rangle \in IEXT(IS(\mathrm{rdfs{:}\,domain}))$ and $\langle IS(\mathrm{u}), IS(\mathrm{v})\rangle \in IEXT(IS(\mathrm{x}))$, then $IS(\mathrm{u}) \in ICEXT(IS(\mathrm{y}))$,
4. if $\langle IS(\mathrm{x}), IS(\mathrm{y})\rangle \in IEXT(IS(\mathrm{rdfs{:}\,range}))$ and $\langle IS(\mathrm{u}), IS(\mathrm{v})\rangle \in IEXT(IS(\mathrm{x}))$, then $IS(\mathrm{v}) \in ICEXT(IS(\mathrm{y}))$,
5. $IEXT(IS(\mathrm{rdfs{:}\,subPropertyOf}))$ is transitive and reflexive on $\mathbf{IP}$,
6. if $\langle IS(\mathrm{x}), IS(\mathrm{y})\rangle \in IEXT(IS(\mathrm{rdfs{:}\,subPropertyOf}))$, then $IS(\mathrm{x})$,$IS(\mathrm{y}) \in \mathbf{IP}$ and $IEXT(IS(\mathrm{x})) \subseteq IEXT(IS(\mathrm{y}))$,
7. $IEXT(IS(\mathrm{rdfs{:}\,subClassOf}))$ is transitive and reflexive on $\mathbf{IC}$,
8. if $\langle IS(\mathrm{x}), IS(\mathrm{y})\rangle \in IEXT(IS(\mathrm{rdfs{:}\,subClassOf}))$, then $IS(\mathrm{x})$,$IS(\mathrm{y}) \in \mathbf{IC}$ and $ICEXT(IS(\mathrm{x})) \subseteq ICEXT(IS(\mathrm{y}))$,
9. if $IS(\mathrm{x}) \in \mathbf{IC}$, then $\langle IS(\mathrm{x}), IS(\mathrm{rdfs{:}\,Resource})\rangle \in IEXT(IS(\mathrm{rdfs{:}\,subClassOf}))$,

[4]We only focus on the core RDFS primitives, i.e., the RDFS predefined meta-properties introduced on page 57.

and satisfies all the RDFS axiomatic statements.[5] ◇

Condition 1 indicates that a 'class' is not a strictly necessary but convenient semantic construct [53] because the class extension function $ICEXT$ is simply a 'syntactic sugar' and is defined in terms of $IEXT$. Handling classes in this way can be counter-intuitive (cf. Proposition 3.4). Condition 2 to 8 are about RDFS meta-properties rdfs:domain, rdfs:range, rdfs:subPropertyOf and rdfs:subClassOf. Condition 9 ensures that all classes are sub-classes of rdfs:Resource.

Proposition 3.4. *The RDFS statements* [rdfs: Resource rdf: type rdfs: Class .] *and* [rdfs: Class rdfs: subClassOf rdfs: Resource .] *are always true in all RDFS-interpretations.*

Proof. For [rdfs: Resource rdf: type rdfs: Class .]:

1. According to the definition of IS and Definition 3.2, for any resource x, we have $IS(\mathrm{x}) \in$ **IR**. Due to **IR** = $ICEXT(IS(\text{rdfs: Resource}))$ and Condition 1 in Definition 3.3, $\langle IS(\mathrm{x}), IS(\text{rdfs: Resource})\rangle \in IEXT(IS(\text{rdf: type}))$. Since rdf: Property is a built-in resource, we have $\langle IS(\text{rdf: Property}), IS(\text{rdfs: Resource})\rangle \in IEXT(IS(\text{rdf: type}))$.

2. Due to [rdf: type rdfs: range rdfs: Class.] (an RDFS axiomatic statement), $\langle IS(\text{rdf: Property}), IS(\text{rdfs: Resource})\rangle \in IEXT(IS(\text{rdf: type}))$ and Condition 4 in Definition 3.3, we have $IS(\text{rdfs: Resource}) \in ICEXT(IS(\text{rdfs: Class}))$. Therefore, for any RDFS-interpretation I, we have I $\models$ [rdfs: Resource rdf: type rdfs: Class .].

For [rdfs: Class rdfs: subClassOf rdfs: Resource .]: According to the definition of **IC**, every class is its member, including $IS(\text{rdfs: Class})$, viz.$IS(\text{rdfs: Class}) \in$ **IC**. Due to Condition 9 of Definition 3.3, $\langle IS(\text{rdfs: Class}), IS(\text{rdfs: Resource})\rangle \in IEXT(IS(\text{rdfs: subClassOf}))$; hence, for any RDFS-interpretation I, we have I $\models$ [rdfs: Class rdfs: subClassOf rdfs: Resource.] □

The two RDFS statements in Proposition 3.4 suggest a strange situation for rdfs:Class and rdfs:Resource as discussed in [108]: on the one hand, rdfs:Resource is an instance of rdfs:Class; on the other hand, rdfs:Class is a sub-class of rdfs:Resource. Hence is rdfs:Resource an instance of its sub-class? Users find this counter-intuitive and thus

[5] Again, readers are referred to [53] for a list of the RDFS axiomatic statements, which includes, e.g., [rdf: type rdfs: range rdfs: Class .].

hard to understand — this is why we say that RDF(S) is more complicated than it appears.

Having described the semantics, we now briefly discuss reasoning in RDF(S). Since RDF(S) does not support the negation constructor, it is impossible, if we do not consider datatypes, to express contradictions. Consistency checking, therefore, is not needed for RDF(S). Instead, entailment is the key inference problem in RDF(S), which can be defined on the basis of interpretations.

Definition 3.5. (**RDF Entailments**) Given two sets of RDF statements S_1 and S_2, S_1 *simply entails* (*RDF-entails, RDFS-entails*) S_2 if all the simple interpretations (RDF-interpretations, RDFS-interpretations, respectively) of S_1 also satisfy S_2. ◇

Limitations of RDF(S)

RDF(S) has the following limitations:

1. The expressiveness of RDF(S) is limited [73]; e.g., we cannot express 'An Elephant is different from a Cow', because negation is not expressible in RDF(S).

2. Some valid RDF(S) statements can be counter-intuitive and hard to understand (such as the two statements presented in Proposition 3.4 on page 61). [104, 23, 108] argue that this is because vocabulary can play dual (or multiple) roles in RDFS-interpretations.; e.g., $\mathrm{rdfs{:}Resource}$ is a super-class and an instance of $\mathrm{rdfs{:}Class}$.

3. RDF(S) provides very few restrictions on its syntax, and no restrictions at all on the use of URIrefs in $\mathbf{rdfV} \cup \mathbf{rdfsV}$ [113]. For example, RDF(S) does not disallow statements like:

   $\mathtt{a}\ \mathrm{rdfs{:}Class}\ \mathtt{b}\ .$
   $\mathrm{rdfs{:}Resource}\ \ \mathrm{rdfs{:}subClassOf}\ \ \mathrm{rdf{:}type}\,.$

   where in the first statement $\mathrm{rdfs{:}Class}$ is used as a property, and in the second statement $\mathrm{rdfs{:}Resource}$ is asserted as a sub-class of $\mathrm{rdf{:}type}$.

4. RDF statements are not *only* standard syntax for annotations, but have built-in semantics [74], i.e., rdf-interpretations; therefore, any langauge extensions of RDF(S) *must* be compatible with such built-in semantics.

5. RDF(S) does not distinguish URI references of classes and properties, including the built-in ones of RDF(S) and its subsequent languages, both of which are interpreted as resources in the domain of interpretations [74].

Limitation 1 suggests it is both necessary and desirable to layer more expressive ontology languages on top of RDF(S). Limitation 2 indicates that it is desirable to have a more intuitive sub-language of RDF(S). Due to Limitations 3-5, there are problems layering OWL on top of RDF(S) (see Section 3.2.2), hence it is crucial to have an OWL-compatible sub-language of RDF(S).

3.2.2 OWL

The OWL language facilitates greater machine understandability of Web resources than that supported by RDFS by providing more expressive vocabulary (classes and properties) and constraints (axioms) along with a formal semantics.

OWL has three increasingly expressive sub-languages: OWL Lite, OWL DL and OWL Full. *OWL Lite* and *OWL DL*[6] are, like DAML+OIL (which is equivalent to the $\mathcal{SHOIQ}(\mathbf{D}^+)$ DL), basically very expressive description logics; they are almost[7] equivalent to the $\mathcal{SHIF}(\mathbf{D}^+)$ and $\mathcal{SHOIN}(\mathbf{D}^+)$ DLs. Therefore, they can exploit existing DL research, e.g., to have well-defined semantics and well studied formal properties, in particular the decidability and complexity of key reasoning services: OWL Lite and OWL DL are both decidable, and the complexity of the ontology entailment problems of OWL Lite and OWL DL is EXPTIME-complete and NEXPTIME-complete, respectively [69]. *OWL Full* is clearly undecidable, as it does not impose restrictions on the use of transitive properties, but presents an attempt at complete integration with RDF(S).

In this section, we will first introduce the syntax and semantics of OWL DL (and therefore OWL Lite), and then briefly describe the problems we encounter when we extend RDF MT to support OWL constructors.

Syntax

OWL DL provides an abstract syntax and an RDF/XML syntax, as well as a mapping from the abstract syntax to the RDF/XML syntax [118].

[6] 'DL' for Description Logic.

[7] They also support annotation properties, which Description Logics do not.

The abstract syntax is heavily influenced by frames in general and by the design of OIL in particular. An OWL ontology in the abstract syntax may contain a sequence of annotations, axioms and facts (i.e., individual axioms). The abstract syntax for OWL class descriptions, property descriptions and axioms (and the corresponding DL syntax) are listed in Tables 3.1, 3.2 (on page 65) and 3.3 (on page 66). Readers are referred to [118] for full details on the abstract syntax. Figure 3.5 on page 64 presents a fragment of the animal ontology (cf. Example 3.1) in the OWL abstract syntax. Note that OWL does not provide customised datatypes (such as $>_{20}$), so AdultElephant is not expressible in OWL.

```
Namespace(elp=<http://example.org/Animal#>)
Ontology(elp:ontology
  Class(elp:Animal)
  Class(elp:Plant)
  Class(elp:Cow partial exp:Animal elp:Herbivore)
  Class(elp:Sheep partial exp:Animal elp:Herbivore)
  Class(elp:Elephant partial exp:Animal elp:Herbivore
        restriction(elp:liveIn someValueFrom(elp:Habitat)))
  Class(elp:Habitat)
  Class(elp:Carnivore)
  Class(elp:Herbivore complete elp:Animal
        restriction(elp:eat allValuesFrom(
              unionOf(elp:Plant restriction(elp:partOf
                    someValueFrom(elp:Plant))))))
  ObjectProperty(elp:partOf Transitive)
  ObjectProperty(elp:eat)
  ObjectProperty(elp:liveIn)

  DisjointClasses(elp:Herbivore elp:Carnivore)

  Individual(elp:Ganesh type(elp:Elephant)
         value(elp:liveIn
               Individual(elp:south-sahara type(elp:Habitat))))
)
```

Figure 3.5: An example ontology in the OWL abstract syntax

The OWL RDF/XML syntax is the exchange syntax for OWL DL. RDF is known (cf. the 'Limitation of RDF(S)' section on page 3.2.1) to have too few restrictions to provide a well-formed syntax for OWL. Therefore, the RDF/XML syntax form of an OWL DL ontology is *valid*, iff it can be translated (according to the mapping rules

Abstract Syntax	DL Syntax	Semantics
ObjectProperty(R)	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
ObjectProperty(S inverseOf(R))	R^-	$(R^-)^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$

Table 3.1: OWL object property descriptions

provided in [118]) from the abstract syntax form of the ontology.

Abstract Syntax	DL Syntax	Semantics
Class(A)	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Class(owl:Thing)	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
Class(owl:Nothing)	$\bot$	$\bot^{\mathcal{I}} = \emptyset$
intersectionOf($C_1, C_2, \ldots$)	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
unionOf($C_1, C_2, \ldots$)	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
complementOf(C)	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
oneOf($o_1, o_2, \ldots$)	$\{o_1\} \sqcup \{o_2\}$	$(\{o_1\} \sqcup \{o_2\})^{\mathcal{I}} = \{o_1^{\mathcal{I}}, o_2^{\mathcal{I}}\}$
restriction(R someValuesFrom(C))	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
restriction(R allValuesFrom(C))	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
restriction(R hasValue(o))	$\exists R.\{o\}$	$(\exists R.\{o\})^{\mathcal{I}} = \{x \mid \langle x, o^{\mathcal{I}}\rangle \in R^{\mathcal{I}}\}$
restriction(R minCardinality(m))	$\geqslant mR$	$(\geqslant mR)^{\mathcal{I}} = \{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \geq m\}$
restriction(R maxCardinality(m))	$\leqslant mR$	$(\leqslant mR)^{\mathcal{I}} = \{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \leq m\}$

Table 3.2: OWL class descriptions

Direct Semantics

As OWL DL is basically a Description Logic, its model-theoretic semantics is very similar to the semantics provided for Description Logics (see Section 2.1), except that, in OWL DL, symbols for classes and properties, etc. are URI references instead of the usual names (strings), and that the model-theoretic semantics includes semantics for annotation properties and ontology properties (cf. [118]). Tables 3.1, 3.2 and 3.3 present the semantics of object property descriptions, concept descriptions, as well as OWL DL axioms and facts, respectively.

OWL Full and OWL DL

OWL Full is an attempt at complete integration with RDF(S). Syntactically, it differs from OWL DL in that it allows arbitrary RDF/XML forms of OWL ontologies, not only those translated from the abstract syntax form of the ontology.

OWL provides an RDF MT-compatible semantics, which is an extension of RDF MT with additional vocabulary. In fact, two version of this semantics are provided

Abstract Syntax	DL Syntax	Semantics
Class(A partial C_1 ... C_n)	$A \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$	$A^{\mathcal{I}} \subseteq C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$
Class(A complete C_1 ... C_n)	$A \equiv C_1 \sqcap \ldots \sqcap C_n$	$A^{\mathcal{I}} = C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$
EnumeratedClass(A o_1 ... o_n)	$A \equiv \{o_1\} \sqcup \ldots \sqcup \{o_n\}$	$A^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}}\}$
SubClassOf(C_1, C_2)	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
EquivalentClasses(C_1 ... C_n)	$C_1 \equiv \ldots \equiv C_n$	$C_1^{\mathcal{I}} = \ldots = C_n^{\mathcal{I}}$
DisjointClasses(C_1 ... C_n)	$C_i \sqsubseteq \neg C_j$, $(1 \leq i < j \leq n)$	$C_1^{\mathcal{I}} \cap C_n^{\mathcal{I}} = \emptyset$, $(1 \leq i < j \leq n)$
SubPropertyOf(R_1, R_2)	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
EquivalentProperties(R_1 ... R_n)	$R_1 \equiv \ldots \equiv R_n$	$R_1^{\mathcal{I}} = \ldots = R_n^{\mathcal{I}}$
ObjectProperty(R super(R_1) ... super(R_n)	$R \sqsubseteq R_i$	$R^{\mathcal{I}} \subseteq R_i^{\mathcal{I}}$
domain(C_1) ... domain(C_k)	$\geqslant 1R \sqsubseteq C_i$	$R^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
range(C_1) ... range(C_h)	$\top \sqsubseteq \forall R.C_i$	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times C_i^{\mathcal{I}}$
[Symmetric]	$R \equiv R^-$	$R^{\mathcal{I}} = (R^-)^{\mathcal{I}}$
[Functional]	$\mathsf{Func}(R)$	$\{\langle x, y\rangle \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \leq 1\}$
[InverseFunctional]	$\mathsf{Func}(R^-)$	$\{\langle x, y\rangle \mid \sharp\{y.\langle x, y\rangle \in (R^-)^{\mathcal{I}}\} \leq 1\}$
[Transitive])	$\mathsf{Trans}(R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$
AnnotationProperty(R)		
Individual(o type(C_1) ... type(C_n)	$o : C_i, 1 \leq i \leq n$	$o^{\mathcal{I}} \in C_i^{\mathcal{I}}, 1 \leq i \leq n$
value(R_1, o_1) ... value(R_n, o_n)	$\langle o, o_i\rangle : R_i, 1 \leq i \leq n$	$\langle o^{\mathcal{I}}, \phi_i^{\mathcal{I}}\rangle \in R^{\mathcal{I}}, 1 \leq i \leq n$
SameIndividual(o_1 ... o_n)	$o_1 = \ldots = o_n$	$o_1^{\mathcal{I}} = \ldots = o_n^{\mathcal{I}}$
DifferentIndividuals(o_1 ... o_n)	$o_i \neq o_j, 1 \leq i < j \leq n$	$o_i^{\mathcal{I}} \neq o_j^{\mathcal{I}}, 1 \leq i < j \leq n$

Table 3.3: OWL axioms and facts

in [118]: one for OWL DL and the other for OWL Full. In order to (try to) make the RDF MT-compatible semantics for OWL DL equivalent to the direct semantics, the domain of discourse is divided into several disjoint parts. In particular, the interpretations of classes, properties, individuals and OWL/RDF vocabulary are strictly separated. Given such a separation, there is a direct correspondence between RDF MT models and standard first-order models. Note that classes and properties, unsurprisingly, cannot be treated as ordinary resources as they are in RDF MT.

As far as the RDF MT-compatible semantics for OWL Full is concerned, the above disjointness restriction is not required. It has yet to be proved, however, that this semantics gives a coherent meaning to OWL Full. There exist at least three known problems that the RDF-compatible semantics for OWL Full needs to solve, but a proven solution has yet to be given.

- **Too Few Entailments [116]** Consider the following question: does the following individual axiom

  ```
  Individual(ex:John
      type(intersectionOf(ex:Student ex:Employee ex:European)))
  ```

  entail the individual axiom

  ```
  Individual(ex:John
      type(intersectionOf(ex:Student ex:European)))?
  ```

In OWL DL, the answer is simply 'yes', since intersectionOf(ex:Student ex:Employee ex:European) is a sub-class of intersectionOf(ex:Student ex:European). Since in RDF(S) every class is a resource, OWL Full needs to make sure of the existence of the resource intersectionOf(ex:Student ex:European) in every possible interpretation; otherwise, the answer will be 'no' which leads to a disagreement between OWL DL and OWL Full.

In general, OWL Full introduces so called *comprehension principles* to add all the missing resources into the domain for all the OWL class descriptions (see Table 3.2 on page 65). This is surely a very difficult task. It has yet to be proved that the proper resources are all added into the universe, no more and no less, and that the added resources will not bring any side-effects.

- **Contradiction Classes [116, 117, 74]** In OWL Full, it is possible to construct a class the instances of which have no rdf:type relationship linked to:

  _:c owl:onProperty rdf:type; owl:allValuesFrom _:d .
  _:d owl:complementOf _:e .
  _:e owl:oneOf _:l
  _:l rdf:first _:c; rdf:rest rdf:nil.

  The above triples require that rdf:type relates member of the class _:c to anything but _:c. It is impossible for one to determine the membership of _:c. If an object is an instance of _:c, then it is not; but if it is not then it is — this is a contradiction class. Note that it is not a valid OWL DL class, as OWL DL disallows using rdf:type as an object property.

  With naive comprehension principles, resources of contradiction classes would be added to all possible OWL Full interpretations, which thus have ill-defined class memberships. To avoid the problem, the comprehension principles must also consider avoiding contradiction classes. Unsurprisingly, devising such comprehension principles took a considerable amount of effort [74], and no proof has ever shown that all possible contradiction classes are excluded in the comprehension principles of OWL Full.

- **Size of the Universe [70]** Consider the following question: is it possible that there is only one object in an interpretation of the following OWL ontology?

  ```
  Individual(elp:Ganesh type(elp:Elephant))
  DisjointClasses(elp:Elephant elp:Plant)
  ```

In OWL DL, classes are not objects, so the answer is 'yes': The only object in the domain is the interpretation of `elp:Ganesh`, the elp:Elephant class thus has one instance, i.e., the interpretation of `elp:Ganesh`, and the elp:Plant class has no instances.

In OWL Full, since classes are also objects, besides elp:Ganesh, the classes elp:Elephant and elp:Plant should both be mapped to the only one object in the universe. This is not possible because the interpretation of `elp:Ganesh` is an instance of elp:Elephant, but not an instance of elp:Plant; hence, elp:Elephant and elp:Plant should be different, i.e., there should be at least two objects in the universe. As the above axioms are valid OWL DL axioms, this example show s that OWL Full disagrees with OWL DL on valid OWL DL ontologies.

Furthermore, this example shows that the interpretation of OWL Full has different features than the interpretation of standard First Order Logic (FOL) model theoretic semantics. This raises the question as to whether it is possible to layer FOL languages on top of RDF(S).

Consequently, there is a serious mismatch between the two versions of RDF MT-compatible semantics. Even for two *OWL DL* ontologies $\mathcal{O}_1$ and $\mathcal{O}_2$, $\mathcal{O}_1$ OWL Full entails $\mathcal{O}_2$ does *not* imply that $\mathcal{O}_1$ OWL DL entails $\mathcal{O}_2$ [118]. Therefore, the semantic connection (at least in terms of entailment) between OWL DL and OWL Full seems rather weak.

OWL and RDF(S)

To sum up, OWL DL and RDF(S) are *not* compatible. OWL DL heavily restricts the syntax of RDF(S), viz. some RDF(S) annotations are not recognisable by OWL DL-compatible agents. Actually, it is not an easy task to check if an RDF graph is an OWL DL ontology. In addition, OWL DL imposes a strong condition on the RDF MT: that the interpretations of classes, properties and individuals be pairwise disjoint. This condition makes the semantics of RDF triples in OWL DL very different from their semantics given by RDF MT. This means that RDF(S) annotations that are valid OWL DL statements do not share the same meaning in OWL DL and RDF(S). As a result, the RDF(S)-compatible intelligent agents and OWL DL-compatible intelligent agents seem unlikely to be able to share their understandings.

The introduction of OWL Full does not make the situation any better. Firstly, due to the three problems discussed earlier, it is unknown whether the RDF-compatible

semantics for OWL Full could give a coherent meaning to OWL Full ontologies. Secondly, as OWL Full is undecidable, it provides no computation guarantee. Furthermore, the descriptions that OWL Full supports are so complex that it is difficult to have tool support for them. In short, it is rather unlikely that intelligent agents could use OWL Full seamlessly to support SW applications.

Therefore, OWL Full can simply be seen as an unsuccessful attempt at integrating OWL and RDF(S). In the rest of the thesis, when we mention 'OWL' we mean 'OWL DL'.

3.3 Web Datatype Formalisms

A major concern when we provide DL reasoning services for OWL DL is the support for OWL datatyping. On the one hand, OWL DL provides some new datatype features that existing DL approaches (cf. Section 2.3) do not support. On the other hand, there have already been complaints about the shortcomings of OWL datatyping. The objective of this section, therefore, is to investigate the new features as well as the limitations of OWL datatyping. The strategy of this thesis is to provide DL reasoning services to support not only OWL DL, but also a more comprehensive extension of OWL DL that overcomes the possibly fatal limitations of OWL datatyping.

OWL datatyping is closely related to two existing Web datatype formalisms. OWL uses many of the built-in XML Schema datatypes as its built-in datatypes, and it adopts the RDF(S) specification of datatypes and data values. In this section, accordingly, we will first introduce XML Schema datatypes, RDF(S) datatyping and then present the OWL datatype formalism. Finally, we will summarise the new features and limitations of OWL datatyping.

3.3.1 XML Schema Datatypes

W3C XML Schema Part 2 [17] defines facilities for defining datatypes (or simple types) to be used in XML Schema as well as other XML specifications.

An *XML Schema datatype* is a triple, consisting of a value space, a lexical space, and a set of facets that characterize properties of the value space, individual values or lexical items:

- A *value space* $V(d)$ is the set of values for a given datatype d. For example, the value space for the $decimal$ datatype is $V(decimal) = \{i \times 10^{-n} \mid i, n \in$

$V(integer) \wedge n \geq 0\}$.

- Each value in the value space of a datatype is denoted by one or more *literals*, or strings, in its lexical space. For example, "100" and "1.0E2" are two different literals from the lexical space of $float$ which both denote the same value. A *lexical space* is the set of valid literals for a datatype.

- A *facet* is a single defining aspect of a value space. For example, the $minExclusive$ facet is the exclusive lower bound of the value space for an ordered datatype. The value of $minExclusive$ must be in the value space of the datatype (called a *base datatype*).

XML Schema datatypes are divided into disjoint built-in datatypes[8] and user-derived datatypes. Derived datatypes can be defined by derivation from primitive or existing derived datatypes by the following three means:

- Derivation by *restriction*, i.e., by using facets on an existing type, so as to limit the number of possible values of the derived type.

- Derivation by *union*, i.e., to allow value from a list of datatypes.

- Derivation by *list*, i.e., to define the list type of an existing datatype.

Note that derivation by restriction is most widely used, while derivation by list is supported by neither RDF(S) nor OWL.

Example 3.2 An XML Schema User-Derived Datatype The following is the definition of a user-derived datatype (of the base datatype $integer$) which restricts values to integers greater than 20, using $minExclusive$.

```
<simpleType name = "greaterThan20">
  <restriction base = "xsd:integer>
    <minExclusive value = "20"/>
  </restriction>
</simpleType>
```
♢

There are two main limitations of the XML Schema type system:

- It does not provide a standard way to access the URI references of user-derived (customised) datatypes. This drawback makes XML Schema user-derived datatypes not accessible by RDF(S) and OWL.

[8] Readers are referred to [17] for the complete list of XML Schema built-in datatypes.

- It does not support n-ary datatype predicates, let alone providing facilities to define customised datatype predicates. This, to some extent, limits the kinds of datatype constraints that RDF(S) and OWL could provide.

3.3.2 RDF(S) Datatyping

RDF(S) provides a specification of datatypes and data values; accordingly, it allows the use of datatypes defined by any external type systems, e.g., the XML Schema type system, which conform to this specification.

Definition 3.6. (Datatype) A *datatype* d is characterised by a lexical space, $L(d)$, which is an non-empty set of Unicode strings; a value space, $V(d)$, which is an non-empty set, and a total mapping $L2V(d)$ from the lexical space to the value space. $\diamond$

For example, $boolean$ is a datatype with value space $\{true, false\}$, lexical space {"T", "F","1","0"} and lexical-to-value mapping {"T"$\mapsto$ $true$, "F"$\mapsto$ $false$, "1"$\mapsto$ $true$, "0"$\mapsto$ $false$}.

Definition 3.7. (Typed and Plain Literals) *Typed literals* are of the form "v"^^u, where v is a Unicode string, called the *lexical form* of the typed literal, and u is a URI reference of a datatype. *Plain literals* have a lexical form and optionally a *language tag* as defined by [1], normalised to lowercase. $\diamond$

The denotation of a typed literal is the value mapped from its enclosed Unicode string by the lexical-to-value mapping of the datatype associated with its enclosed datatype URIref. For example, "1"^^$\mathrm{xsd{:}\,boolean}$ is a typed literal that represents the boolean value *true*, while "1"^^$\mathrm{xsd{:}\,integer}$ represents the integer 1. Plain literals, e.g., "1", are considered to denote themselves [53].

The associations between datatype URI references (e.g., $\mathrm{xsd{:}\,boolean}$) and datatypes (e.g., $boolean$) can be provided by datatype maps defined as follows.

Definition 3.8. (Datatype Map) We consider a datatype map $\mathbf{M}_d$ that is a partial mapping from datatype URI references to datatypes. $\diamond$

Example 3.3 Datatype Map $\mathbf{M}_{d1} = \{\langle \mathrm{xsd{:}\,string}, string\rangle, \langle \mathrm{xsd{:}\,integer}, integer\rangle\}$ is a datatype map, where $\mathrm{xsd{:}\,string}$ and $\mathrm{xsd{:}\,integer}$ are datatype URI references, and $string$ and $integer$ are datatypes. $\diamondsuit$

A datatype map may include some built-in XML Schema datatypes (as seen in Example 3.3), while other built-in XML Schema datatypes are problematic and thus unsuitable for various reasons. For example, xsd:ENTITIES is a list-value datatype that does not fit the RDF datatype model.[9] Please note that derived XML Schema datatypes are not RDF(S) datatypes, because there is no standard way to access a derived XML Schema datatype through a URI reference. Therefore, there is no way to include a derived XML Schema datatype in a datatype map.

An RDFS-interpretation w.r.t. a datatype map $\mathbf{M}_d$ can be defined as follows.

Definition 3.9. (RDFS $\mathbf{M}_d$-Interpretation) Given a datatype map $\mathbf{M}_d$, an *RDFS* $\mathbf{M}_d$-*interpretation* I of a vocabulary **V** is any RDFS-interpretation of $\mathbf{V} \cup \{u \mid \exists\, d.\langle u, d\rangle \in \mathbf{M}_d\}$ which introduces

- A distinguished subset **LV** of **IR**, called the *set of literal values*, which contains all the plain literals in **V**,
- a mapping IL from typed literals in **V** into **IR**,

and satisfies the following extra conditions:

1. **LV** = $ICEXT(IS(\text{rdfs:Literal}))$,
2. for each pair $\langle u, d\rangle \in \mathbf{M}_d$,
   (a) $ICEXT(d) = V(d) \subseteq \mathbf{LV}$,
   (b) there exist $d \in \mathbf{IR}$ s.t. $IS(\text{u}) = d$,
   (c) $IS(\text{u}) \in ICEXT(IS(\text{rdfs:Datatype}))$,
   (d) for "s"^^$u' \in \mathbf{V}$,$IS(\text{u}') = d$, if $s \in L(d)$, then $IL(\text{"}s\text{"}\hat{}\hat{}u') = L2S(d)(s)$, otherwise, $IL(\text{"}s\text{"}\hat{}\hat{}u') \notin \mathbf{LV}$,
3. if $d \in ICEXT(IS(\text{rdfs:Datatype}))$, then $\langle d, IS(\text{rdfs:Literal})\rangle \in IEXT(\text{rdfs:subClassOf})$. ◇

According to Definition 3.9, **LV** is a subset of **IR**, i.e., literal values are resources. Condition (1) ensures that the class extension of rdfs:Literal is **LV**. Condition (2a) asserts that RDF(S) datatypes are classes, condition (2b) ensures that there is a resource d for datatype d in $\mathbf{M}_d$, condition (2c) ensures that the class rdfs:Datatype contains

[9] See the RDF semantics document http://www.w3.org/TR/rdf-mt/#dtype_interp for the complete list of RDF(S) built-in datatypes.

the datatypes used in any satisfying $\mathbf{M}_d$-interpretation, and condition (2d) explains why the range of IL is **IR** rather than **LV** (because, for "s"^^u, if $s \notin L(IS(\mathrm{u}))$, then $IL(\text{“}s\text{”}\hat{}\hat{}u) \notin$ **LV**). Condition (3) requires that RDF(S) datatypes are sub-classes of $\mathrm{rdfs{:}Literal}$.

If the datatypes in the datatype map $\mathbf{M}_d$ impose disjointness conditions on their value spaces, it is possible for an RDF graph to have no RDFS $\mathbf{M}_d$-interpretation which satisfies it, i.e., there exists a *datatype clash*. For example,

$\_ : \mathtt{x}\ \ \mathrm{rdf{:}type}\ \ \mathrm{xsd{:}string}\ \ .$
$\_ : \mathtt{x}\ \ \mathrm{rdf{:}type}\ \ \mathrm{xsd{:}decimal}\ \ .$

would constitute a datatype clash because the value spaces of xsd:string and xsd:decimal are disjoint. In RDF(S), an ill-typed literal does not in itself constitute a datatype clash (cf. Condition(2d) in Definition 3.9), but a graph which entails that an ill-typed literal has rdf:type rdfs:Literal would be inconsistent.

Let S, E be two sets of RDFS statements. S *RDFS-*$\mathbf{M}_d$*-entails* E if every RDFS $\mathbf{M}_d$-interpretation of S also satisfies E.

3.3.3 OWL Datatyping

Datatypes

OWL uses many of the built-in XML Schema datatypes, and it adopts the RDF(S) specification of datatypes and data values. Some built-in XML Schema datatypes are problematic for OWL, as they do not fit the RDF(S) datatype model. Readers are referred to [119] for the complete list of OWL built-in datatypes. Note that the built-in RDF datatype, rdf:XMLLiteral, is also an OWL built-in datatype.

The fundamental difference between RDF(S) datatyping and OWL datatyping is the relationship between datatypes and classes. In OWL DL, datatypes are *not* classes, and object and datatype domains are *disjoint* with each other. Such disjointness is motivated by both philosophical and pragmatic considerations (Horrocks et al. [73]):

- Datatypes (e.g., the $>_{20}$ datatype) are different from classes (e.g., Tree) in that datatypes and the predicates (such as $=, <, +$) defined over them have fixed extension (e.g., the extension of $>_{20}$ is all the integers that are greater than 20), while classes could have different interpretations in different models.

- The simplicity, compactness and the semantic integrity of the ontology language are not compromised. We do not have to provide a logical theory for each

datatype, nor do we have to worry whether each of them is still correct whenever we extend the ontology language.

- The 'implementability' of the language is not compromised. A hybrid reasoner can easily be implemented by combining a reasoner for the 'object language' with one (or possibly more) datatype reasoner(s) that can decide the satisfiability problem of datatype constraints.

OWL datatyping further distinguishes supported from unsupported datatype URI references w.r.t. a datatype map $\mathbf{M}_d$ and allows both of them to be used in an OWL DL ontology. The motivation is that different OWL DL reasoners could provide different supported datatype URIrefs, but will not simply disallow the use of datatype URIrefs that they do not support.

Definition 3.10. (Supported and Unsupported Datatype URIrefs) Given a datatype map $\mathbf{M}_d$, a datatype URI reference u is called a *supported datatype URI reference w.r.t.* $\mathbf{M}_d$ (or simply a *supported datatype URIref*) if there exists a datatype d s.t. $\mathbf{M}_d(u) = d$ (in this case, d is called a *supported datatype w.r.t.* $\mathbf{M}_d$); otherwise, u is called an *unsupported datatype URI reference w.r.t.* $\mathbf{M}_d$ (or simply an *unsupported datatype URIref*). ◇

For example, $\mathrm{xsd{:}integer}$ is a supported datatype URIref w.r.t. $\mathbf{M}_{d1}$ given in Example 3.3, while $\mathrm{xsd{:}boolean}$ is an unsupported datatype URIref w.r.t. $\mathbf{M}_{d1}$. OWL DL requires at least $\mathrm{xsd{:}integer}$ and $\mathrm{xsd{:}string}$ be supported datatype URIrefs. Other built-in OWL datatype URIrefs can be either supported or unsupported.

As built-in XML Schema datatypes are usually not enough in many SW and ontology applications, OWL provides the use of so called enumerated datatypes, which are built using literals.

Definition 3.11. (Enumerated Datatypes) Let $y_1, \ldots, y_n$ be typed literals. An *enumerated datatype* is of the form oneOf($y_1 \ \ldots \ y_n$). ◇

For example, oneOf("0"^^$\mathrm{xsd{:}integer}$ "15"^^$\mathrm{xsd{:}integer}$ "30"^^$\mathrm{xsd{:}integer}$ "40"^^$\mathrm{xsd{:}integer}$) is an enumerated datatype, which is interpreted as $\{0, 15, 30, 40\}$. The above enumerated datatype, e.g., can be used as the range of a datatype property $tennisScore$.

The semantics of OWL datatypes is defined w.r.t. a datatype map.

Definition 3.12. (Datatype Interpretation) An *OWL datatype interpretation* w.r.t. to a datatype map $\mathbf{M}_d$ is a pair $(\Delta_\mathbf{D}, \cdot^\mathbf{D})$, where the datatype domain $\Delta_\mathbf{D} = \mathbf{PL} \cup \bigcup_{\text{for each supported datatype URIref u w.r.t. } \mathbf{M}_p} V(\mathbf{M}_p(u))$ ($\mathbf{PL}$ is the value space for plain literals, i.e., the union of the set of Unicode strings and the set of pairs of Unicode strings and language tags) and $\cdot^\mathbf{D}$ is a datatype interpretation function, which has to satisfy the following conditions:

1. $\mathrm{rdfs{:}\,Literal}^\mathbf{D} = \Delta_\mathbf{D}$;
2. for each plain literal l, $l^\mathbf{D} = l \in \mathbf{PL}$;
3. for each supported datatype URIref u (let $d = \mathbf{M}_d(u)$):
   (a) $u^\mathbf{D} = V(d) \subseteq \Delta_\mathbf{D}$,
   (b) if $s \in L(d)$, then $(\text{“}s\text{”}\hat{}\hat{}u)^\mathbf{D} = L2V(d)(s)$,
   (c) if $s \notin L(d)$, then $(\text{“}s\text{”}\hat{}\hat{}u)^\mathbf{D}$ is not defined;
4. for each unsupported datatype URIref u, $u^\mathbf{D} \subseteq \Delta_\mathbf{D}$, and $(\text{“}s\text{”}\hat{}\hat{}u)^\mathbf{D} \in u^\mathbf{D}$.
5. each enumerated datatype oneOf($y_1 \ \ldots \ y_n$) is interpreted as $y_1^\mathbf{D} \cup \ldots \cup y_n^\mathbf{D}$. ◇

Condition 1 of Definition 3.12 asserts that $\mathrm{rdfs{:}\,Literal}$ is interpreted as the datatype domain of a datatype interpretation. Condition 2 ensures that plain literals are interpreted as themselves. Condition 3 ensures that supported datatype URIrefs are interpreted as the value spaces of the datatypes they represent, and it also ensures the valid interpretations of typed literals with supported datatype URIrefs. Note that, in OWL, ill-typed literals have no interpretations, which is different from RDF(S) datatyping (cf. Definition 3.9 on page 72). Condition 4 states that unsupported datatype URIrefs are interpreted as any subsets of the datatype domain, which implies that typed literals associated with unsupported datatype URIrefs are interpreted as some members of the datatype domain, although we do not know exactly which members (of the datatype domain) they are interpreted as. Condition 5 gives the interpretation of enumerated datatypes.

Object Language and Datatypes

OWL datatype properties relate objects to data values in data ranges (see Table 3.4); they are disjoint with object properties, which relate objects to objects. OWL provides various datatype property axioms: besides the usual sub-property, domain and

Abstract Syntax	DL Syntax	Semantics
rdfs:Literal	$\top_{\mathrm{D}}$	$\Delta_{\mathbf{D}}$
u a datatype URIref	u	$u^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}$
oneOf("s_1"^^u_1 ... "s_n"^^u_n)	$\{$"s_1"^^$u_1, \ldots,$ "s_n"^^$u_n\}$	$\{t_1\} \cup \ldots \cup \{t_n\} \subseteq \Delta_{\mathbf{D}}$ $(t_i = L2V(u_i)(s_i))$

Table 3.4: OWL data ranges

Abstract Syntax	DL Syntax	Semantics
(SubPropertyOf T_1, T_2)	$T_1 \sqsubseteq T_2$	$T_1^{\mathcal{I}} \subseteq T_2^{\mathcal{I}}$
(EquivalentProperties $T_1, \ldots, T_n$)	$T_1 \equiv \ldots \equiv T_n$	$T_1^{\mathcal{I}} = \ldots = T_n^{\mathcal{I}}$
DatatypeProperty(T super(T_1)...super(T_n)	$T \sqsubseteq T_i$	$T^{\mathcal{I}} \subseteq T_i^{\mathcal{I}}$
domain(C_1)...domain(C_k)	$\geqslant 1T \sqsubseteq C_i$	$T^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta_{\mathbf{D}}$
range(d_1)...range(d_h)	$\top \sqsubseteq \forall T.d_i$	$T^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times d_i^{\mathbf{D}}$
[Functional])	$\mathsf{Func}(T)$	$\forall x \in \Delta^{\mathcal{I}}.\sharp\{t \mid \langle x,t\rangle \in T^{\mathcal{I}}\} \leq 1$

Table 3.5: OWL datatype property axioms

range axioms, OWL allows users to specify a datatype property to be functional, viz. a functional datatype property can relate an object to at most one data value.[10] Table 3.5 shows the abstract syntax of OWL datatype property axioms, as well as their corresponding DL semantics , where T_i are datatype properties, d_i are data ranges, $\cdot^{\mathcal{I}}$ (as usual) is a Description Logic interpretation function for the object domain and $\sharp$ denotes cardinality.

In addition, OWL DL provides several datatype-related concept descriptions (see Table 3.6): datatype exists, datatype value restrictions, and datatype (unqualified) number restrictions, which specify the set of objects related by a datatype property to more (or less) than a certain number of data values.

[10]By default, datatype properties are not functional; e.g., a Person can have multiple $telephoneNumber$s.

Abstract Syntax	DL Syntax	Semantics
restriction(T someValuesFrom(d))	$\exists T.d$	$\{x \mid \exists t.\langle x,t\rangle \in T^{\mathcal{I}} \wedge t \in d^{\mathbf{D}}\}$
restriction(T allValuesFrom(d))	$\forall T.d$	$\{x \mid \exists t.\langle x,t\rangle \in T^{\mathcal{I}} \rightarrow t \in d^{\mathbf{D}}\}$
restriction(T minCardinality(m))	$\geqslant mT$	$\{x \mid \sharp\{t \mid \langle x,t\rangle \in T^{\mathcal{I}}\} \geq m\}$
restriction(T maxCardinality(m))	$\leqslant mT$	$\{x \mid \sharp\{t \mid \langle x,t\rangle \in T^{\mathcal{I}}\} \leq m\}$

Table 3.6: OWL datatype-related concept descriptions

New Requirements

To summarise, OWL datatyping presents new requirements to DL research:

- It adopts the RDF(S) specification of datatypes and data values, while existing DL approaches mainly consider predicates instead of datatypes and data values.
- It provides enumerated datatypes and datatype-related number restrictions, which are beyond the expressive power of existing DL languages.

Existing DL approaches should, therefore, be extended to support these new requirements.

Limitations of OWL Datatyping

There have already been complaints [29, 91] from users about two serious limitations of OWL datatyping, which discourage potential users from adopting OWL DL in their SW and ontology applications.

1. **Customised Datatypes** Many SW and ontology applications need to define customised datatypes for their own purposes. For example, to define the class AdultElephant in the animal ontology presented in Example 3.1 (page 53) requires the customised datatype $>_{20}$. OWL datatyping does not allow the use of user-derived XML Schema datatypes, since the XML Schema type system provides no standard way to access URIrefs of user-derived datatypes. Enumerated datatypes are not expressive enough to be an alternative to user-derived XML Schema datatypes, since they cannot represent infinite datatypes, such as 'GreaterThan20' presented in Example 3.2 (page 70). Furthermore, they are clumsy to represent big ranges, e.g., the integer range [1, 1000000].
2. **Customised Datatype Predicates** Many SW and ontology applications need to represent constraints over multiple datatype properties, such as the sum of $height$, $length$ and $width$ of SmallItems is no greater than 15cm, the $pricing\text{-}Pounds$ and $priceInDollars$ should conform to a certain exchange rate, etc. (see Section 1.3). OWL does not support the use of n-ary datatype predicates, not to mention customised datatype predicates.

In addition, OWL datatyping has several other limitations:

1. OWL does not support negated datatypes. For example, 'all integers but 0', which is the relativised negation of the enumerated datatype oneOf("0"^^xsd:integer), is not expressible in OWL. Moreover, negated datatypes are necessary in the negated normal form (NNF)[11] of datatype-related class descriptions in, e.g., DL tableaux algorithms.

2. An OWL DL datatype domain seriously restricts the interpretations of typed literals with unsupported datatype URIrefs. For example, given the datatype map $\mathbf{M}_{d1}$ (cf. Example 3.3), "1.278e-3"^^xsd:float will have to be interpreted as either an integer, a string or a string with a language tag, which is counter-intuitive.

3. Users cannot define names for enumerated datatypes; viz. users have to construct the enumerated datatypes whenever they want to use them. This is very inconvenient if they need to use enumerated datatypes with large numbers of typed literals multiple times in their ontologies.

3.4 Outlook for the Two Issues

Issue 1: What is the connection between RDF(S) and OWL? As explained in this chapter, the intended foundation of the Semantic Web RDF(S) and the SW standard ontology language OWL are not compatible with each other, because of RDF(S)'s non-standard semantics. In Chapter 4 we will present a modified semantics for RDF(S) that addresses this problem and restores the desired connection between RDF(S) and OWL.

Issue 2: How to extend and provide reasoning support for OWL Datatyping? In this chapter, we have shown that OWL has a very serious limitation; i.e., it does not support customised datatypes and datatype predicates. This limitation hinders the wider acceptance of OWL DL in SW and ontology applications [126].

We will provide a solution in Chapter 5 to 8. Firstly, we will propose a comprehensive formalism to unify datatypes and predicates and to identify a family of decidable DLs that provide customised datatypes and datatype predicates (Chapter 5). Secondly, we propose two extension of OWL DL, i.e., OWL-E and OWL-Eu, both of which are members of the above identified family of the decidable combined DLs. Thirdly, we

[11] A concept is in negation normal form iff negation is applied only to atomic concept names, nominals or datatype ranges/expressions; see page 133.

provide practical decision procedures for a family of DLs that are closely related to OWL DL, OWL-Eu and OWL-E (Chapter 6). Last but not least, we will present the framework architecture and discuss the flexibility of our framework (Chapter 7 and 8).

Chapter Achievements

- The construction, integration and deployment of DL-based ontologies benefit greatly from a well-defined DL semantics and powerful DL reasoning tools.
- OWL Lite and OWL DL are DL-based Web ontology languages, which can thus exploit existing DL research, e.g., to have well-defined semantics, well known formal properties, in particular the decidability and complexity of key reasoning services.
- The semantic problems of layering OWL on top of RDF(S) stem from the following characteristics of RDF(S):
  - RDF triples have built-in semantics.
  - Classes and properties, including built-in classes and properties of RDF(S) and its subsequent languages such as OWL, are treated as objects (or resources) in the domain.
  - There are no restrictions on the use of built-in vocabularies.
- Although OWL DL is very expressive and provides some new requirements about datatypes to DL research, OWL datatyping has many limitations. A very serious limitation that hinders its wider acceptance is the lack of support for customised datatypes and datatype predicates.

Chapter 4

An Important Connection

Chapter Aims:

- To investigate a novel modification of RDF(S) that keeps the main features of RDFS and is able to serve as a firm semantic foundation for the latest DL-based SW ontology languages.

Chapter Plan

As shown in Chapter 3, there are problems layering OWL on top of RDF(S). In this chapter we propose *RDFS(FA)* (RDFS with Fixed Architecture), as an alternative to RDFS, to restore the desired connection between RDF(S) and OWL DL.

4.1 RDFS(FA): A DL-ised Sub-language of RDFS

4.1.1 Introduction

Let us first consider design requirements of RDFS(FA) from the following two aspects: (i) Semantics Web language layering and (ii) SW and ontology applications.

Language Layering Requirements

To restore the connection between RDF(S) and OWL, we shall consider a sub-language of RDFS with a semantics that is compatible with the direct semantics of OWL DL. Ontologies in this sub-language are still valid RDFS ontologies.

From the lessons we learnt in the 'OWL Full and OWL DL' Section on page 65, RDFS(FA) should address the following problems of RDF(S):

- RDF triples have built-in semantics.
- Classes and properties, including built-in classes and properties of RDF(S) and its subsequent languages such as OWL, are treated as objects (or resources) in the domain.
- There are no restrictions on the use of built-in vocabularies.

Application Requirements

The main challenge to RDFS(FA) is that it should also provide the main features of RDFS that OWL does not provide, viz., it should provide meta-classes, meta-properties and enable the use of class symbols (URIrefs) as property values.

Example 4.1 RDFS: Meta-classes and Meta-properties

Applications using WordNet [98] to annotate resources, such as images [146], require the use of meta-classes and meta-properties.

```
@prefix wnc:   <http://www.cogsci.princeton.edu/~wn/concept#>
@prefix wns:   <http://www.cogsci.princeton.edu/~wn/schema#>

wns:LexicalConcept rdfs:subClassOf rdfs:Class .
wns:hyponymOf rdfs:subPropertyOf rdfs:subClassOf;
    rdfs:domain wns:LexicalConcept ;
    rdfs:range wns:LexicalConcept .
wnc:100002086 wns:hyponymOf wnc:100001740 .
```

where wnc:100002086 and wnc:100001740 are WordNet synsets (i.e., concepts like 'Elephant' and 'Animal'). The first statement specifies that the class LexicalConcept is a subclass of the built-in RDFS meta-class rdfs:Class, the instances of which are classes. This means that now all instances of LexicalConcept are also classes. In a similar vein, the second statement defines that the WordNet property $hyponymOf$ is a

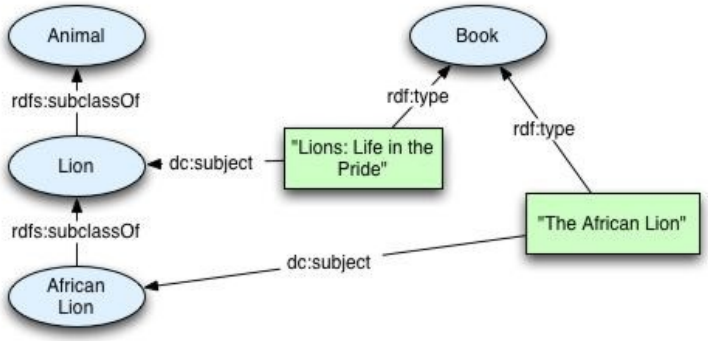

Figure 4.1: RDFS: classes as property values (from [105])

sub-property of the built-in RDFS meta-property rdfs:subClassOf. This enables us to interpret the instances of $hyponymOf$ as subclass links.

We invite the reader to note that the metamodeling architecture of RDFS may not be what one expects, as it is valid to add RDFS triples such as

rdfs:Class rdf:type wnc:100002086 .

into the above WordNet ontology, which makes the relationship between wnc:100002086 and wns:LexicalConcept rather confusing: wnc:100002086 is an instance of wns:LexicalConcept, which is an instance of an instance of wnc:100002086; nevertheless, wnc:100002086 and wns:LexicalConcept are not necessarily equivalent to each other. ♢

It has been argued [105] that it is often useful to use classes as values for properties. Users want to use whatever they believe to be intuitive as values of properties, including classes, properties, etc.

Example 4.2 RDFS: Classes as Property Values

This example is from [105]. Suppose we have a set of Books *about* Animals and want to annotate each Book with its $subject$, which is a particular species or class of Animals that it talks about (cf. Figure 4.1). Further, when retrieving all Books *about* Lions from a repository, we want Books that are annotated as books *about* AfricanLions to be included in the results.

```
@prefix bk:   <http://protege.stanford.edu/swbp/books#>

bk:AfricanLion rdf:type rdfs:Class;
    rdfs:subClassOf bk:Lion .
bk:LionsLifeInThePrideBook rdf:type bk:Book ;
```

```
    bk:bookTitle "Lions:  Life in the Pride" ;
    dc:subject bk:Lion .
bk:TheAfricanLionBook rdf:type bk:Book ;
    bk:bookTitle "The African Lion" ;
    dc:subject bk:AfricanLion .
```

As classes are objects in RDF(S), classes bk:Lion and bk:AfricanLion can be used as values of properties.

We invite the reader to note that it is not proper to use information properties (*dc:subject*) as properties in ontologies (except annotation properties, cf. Example 4.4), since properties in ontologies (except annotation properties) could have extra implications, due to various constraints in an ontology (cf. comparison between Dublin Core and ontologies on page 56). ♢

Overview of RDFS(FA)

RDFS(FA) has a First Order/Description Logic style semantics, and introduces a Fixed layered metamodeling Architecture to RDFS.

One of the interesting features of RDFS(FA) is its metamodeling architecture. Let us recall that RDFS has a non-layered metamodeling architecture; resources in RDFS can be classes, objects and properties at the same time, viz. classes and their instances (as well as relationships between the instances) are the same layer. RDFS(FA), instead, provides a layered metamodeling architecture that is very similar to that of the widely used Unified Modelling Language (UML) [31]. Like UML, RDFS(FA) divides up the universe of discourse into a series of strata (or layers). The built-in modelling primitives of RDFS are separated into different strata of RDFS(FA), and the semantics of modelling primitives depend on the stratum they belong to. Theoretically there can be a (countably) infinite number of strata in the metamodeling architecture; in practice, four strata (as shown in Figure 4.2) are usually enough. The UML-like meta-modeling architecture makes it easier for users who are familiar with UML to understand and use RDFS(FA).

Figure 4.2 shows strata 0 to strata 3, i.e., the lowest four layers of the Metamodeling Architecture of RDFS(FA); they are also called the Instance Layer, the Ontology Layer, the Language Layer and the Meta-Language Layer, respectively:

- URI references in the Instance Layer (e.g., bk:TheAfricanLionBook) are interpreted as individual objects.

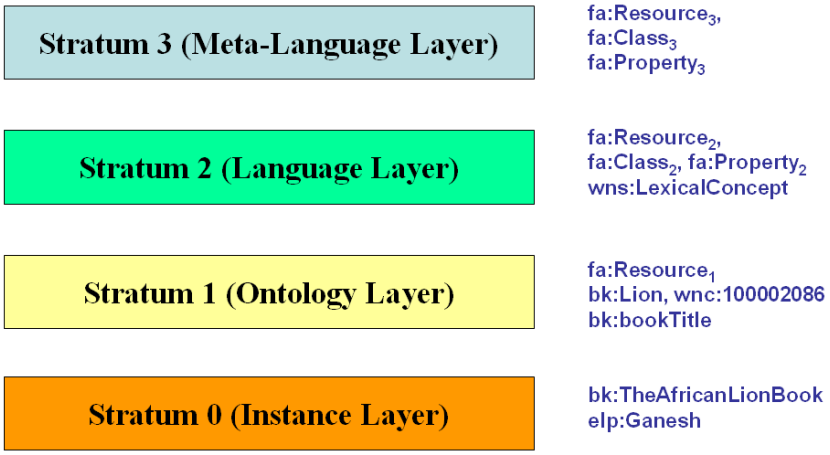

Figure 4.2: (The lowest four layers of) The metamodeling architecture of RDFS(FA)

- URI references in the Ontology Layer (e.g., wnc:100002086, bk:Lion, *bk:book-Title*) are interpreted as ontology classes and ontology properties.

- URI references in the Language Layer (e.g., fa:Class$_2$ and fa:Property$_2$) are used to define and describe elements in the Ontology Layer. Note that, besides some built-in language classes and properties, URI references of meta-classes and meta-properties (e.g., wns:LexicalConcept and *wns:hyponymOf*) can be used in the Language Layer (and the layers above).

- URI references in the Meta-Language Layer (e.g, fa:Class$_3$ and fa:Resource$_3$) are used to define and describe elements in the Language Layer.

As seen in Figure 4.2, rdfs:Resource is stratified into three layers, i.e., fa:Resource$_1$ in the Ontology Layer, fa:Resource$_2$ in the Language Layer and fa:Resource$_3$ in the Meta-Language Layer. Similarly, rdfs:Class and rdfs:Property are stratified into the Language Layer and the Meta-Language Layer.

In RDFS(FA), classes cannot be objects and vice versa; in RDFS, Web resources can be classes, properties, objects or even datatypes all at once. We argue that RDFS(FA) is more intuitive than RDFS based on the following observation: when users design their ontologies, a common concern is to decide whether to model something in the domain as a class or as an object. This concern suggests that users intuitively tend to assume that classes and objects should be different from each other. Therefore, layered meta-models are more intuitive than non-layered meta-models.

Each RDFS(FA) built-in vocabulary (except fa:AnnotationProperty, fa:Literal, and built-in annotation properties) has a number to indicate the stratum it belongs to. Here are two *rules of thumb* to get these numbers of strata right: for an RDFS(FA) triple [s p o .], (i) if p is *not* an instance-of relationship, then s and o should be in the same stratum, and p should be one stratum higher than s and o, e.g., both wns:LexicalConcept and fa:Class_2 are in stratum 2 and fa:subClassOf_3 is in stratum 3 (cf. Example 4.3); (ii) if p is an instance-of relationship, then o is one stratum higher than s, and p is in the same stratum as o (cf. Example 4.4). Developers can instrument tools that would maintain the numbers of strata automatically.

Example 4.3 RDFS(FA): Meta-classes and Meta-properties

We now revisit Example 4.1 and use RDFS(FA) to represent the meta-class wns:LexicalConcept and the meta-property *wns:hyponymOf* in WordNet.

wns:LexicalConcept fa:subClassOf_3 fa:Class_2 .
wns:hyponumOf fa:type_3 $\text{fa:AbstractProperty}_3$;
$\text{fa:subPropertyOf}_3$ fa:subClassOf_2 ;
fa:domain_3 wns:LexicalConcept ;
fa:range_3 wns:LexicalConcept .
wnc:100002086 fa:type_2 fa:Class_2 ;
wns:hyponymOf wnc:100001740 .

Now the layering of the model is much clearer: wnc:100002086 and wnc:100001740 are in stratum 1, wns:LexicalConcept and *wns:hyponymOf* are in stratum 2. The rules of thumb presented above disallow asserting that fa:Class_2 is an instance of wnc:100002086. Hence, there is no confusion here. ♢

In RDFS(FA), all Web resources can have data-value annotation properties (cf. Section 4.1.2), i.e., the values of annotation properties can be either typed literals or plain literals. For example, typed literals of the built-in XML Schema datatype xsd:anyURI can be used as values of annotation properties. Their interpretations (see Definition 4.2 on page 89) are URI references; this thus allows one to refer to URI references of any ontology elements through annotation properties.

Example 4.4 RDFS(FA): Class URIrefs as Values of Annotation Properties

We now revisit Example 4.2 and use the annotation property dc:subject to refer to class URIrefs (cf. Figure 4.3). The approach we presents here is slightly different from Approach 5 in [105] in that annotations are class URIrefs instead of classes.

```
@prefix bk:   <http://protege.stanford.edu/swbp/books#>
```

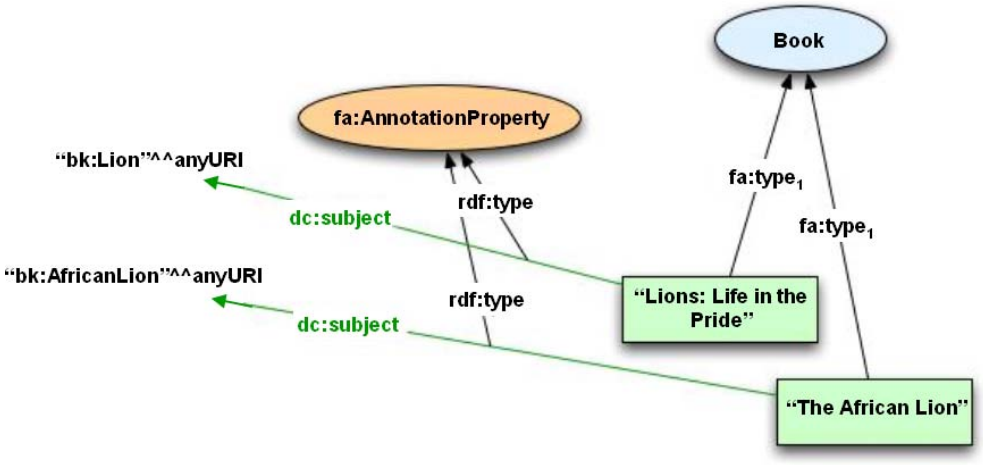

Figure 4.3: RDFS(FA): class URIrefs as annotation property values

```
bk:bookTitle rdf:type fa:AnnotationProperty .
dc:subject rdf:type fa:AnnotationProperty .
bk:AfricanLion fa:type2 fa:Class2 ;
    fa:subClassOf2 bk:Lion .
bk:LionsLifeInThePrideBook fa:type1 bk:Lion ;
    bk:bookTitle "Lions: Life in the Pride" ;
    dc:subject "bk:Lion"^^xsd:anyURI .
bk:LionsLifeInThePrideBook fa:type1 bk:Lion ;
    bk:bookTitle "The African Lion" ;
    dc:subject "bk:AfricanLion"^^xsd:anyURI .
```

We recall that in Example 4.2 dc:subject is used as an object property; its values include class URIrefs bk:Lion and bk:AfricanLion, which are interpreted as classes (arbitrary sets of resources). Here the values of dc:subject are interpretations of typed literals "bk:Lion"^^xsd:anyURI and "bk:AfricanLion"^^xsd:anyURI, viz. class URIrefs bk:Lion and bk:AfricanLion. The latter interpretation is more appropriate. As values of information properties, typed or plain literals can properly represent external agreements because they have fixed interpretations, which classes do not have.

Since the result of classification of such an RDFS(FA) ontology can be represented as partial orderings of class URIrefs (such as bk:AfricanLion $<$ bk:Lion $<$ bk:Animal), we can make use of such result when retrieving all books about bk:Lion from a repository, i.e., by retrieving books that are annotated (through dc:subject) with bk:Lion *and* books annotated with bk:AfricanLion. ♢

4.1.2 Semantics

Let us introduce the design philosophy of RDFS(FA), before moving on to the formal semantics of RDFS(FA).

Design Philosophy

The design of RDFS(FA) embodies two main principles:

1. In RDFS(FA), RDF is used (only) as standard *syntax* for annotations, i.e., the built-in semantics for RDF triples are disregarded, and new semantics is given to RDFS(FA) triples.

2. RDFS(FA) provides various Web resources with DL-style semantics.

In RDFS(FA), we distinguish seven groups of Web resources: object resources (or objects for short), class resources (or classes), datatype resources (or datatypes), abstract property resources (or abstract properties), datatype property resources (or datatype properties), annotation property resources (or annotation properties) and instance-of property resources (or instance-of properties). The URIrefs for the six groups of Web resources are pairwise *disjoint*.

Objects Informally speaking, an object URIref is interpreted as an object; e.g., `bk:TheAfricanLionBook` is interpreted as an object.

Classes Informally speaking, a class URIref is interpreted as a set of objects or sets in the adjacent lower layer. These objects (or sets) are called the instances of the class. In the Ontology Layer, a class URIref (such as bk:Book) is interpreted as a set of objects in the Instance Layer (including, e.g., the interpretation of the object URIref `bk:TheAfricanLionBook`). In the Language Layer, a class URIref (such as wns:LexicalConcept) is interpreted as a set of sets that are in the Ontology Layer (such as the interpretation of the class URIrefs bk:Book).

Datatypes The semantics of a datatype in RDFS(FA) is similar to that in RDF(S), except that a datatype is not a class.

Abstract Properties Informally speaking, an abstract property URIref is interpreted as a set of binary relationships (or pairs) between instances of two classes that are in the same stratum as the property. In the Ontology Layer, an abstract property URIref (such as *elp:liveIn*) is interpreted as a set of binary relationships between instances of the interpretations of the class URIrefs (such as elp:Elephant and elp:Habitat) in the Ontology Layer. In the Language Layer, an abstract property URIref (such as *wns:hyponymOf*) is interpreted as a set of binary relationships between instances of the interpretations of the class URIrefs (such as wns:LexicalConcept) in the Language Layer.

Datatype Properties Informally speaking, a datatype property URIref (such as *elp:age*) is interpreted as a set of binary relationships (or pairs) between instances of a class in the Ontology layer (such as elp:Animal) and data values (such as integers).

Annotation Properties Annotation properties are similar to datatype properties in that their values are data values, but they are not bound to any strata. An annotation property URIref (such as $\mathrm{dc{:}creator}$) is interpreted as a set of binary relationships between some resources and data values. In RDFS(FA), not only objects, but classes and properties can have (even share) annotation properties (such as $\mathrm{dc{:}creator}$).

Instance-Of Properties Informally speaking, an instance-of property is a built-in RDFS(FA) property that connects resources in two adjacent strata. For example, the instance-of property $\mathrm{fa{:}type_1}$, which is used between stratum 0 and 1, is interpreted as a set of binary relationships between resources in stratum 0 (e.g., the interpretation of the object URIref `bk:TheAfricanLionBook`) and resources in stratum 1 (e.g., the interpretation of the class URIref bk:Book).

Interpretations

The semantics of RDFS(FA) starts with the notation of vocabulary. Instead of having a mixed vocabulary like that of RDF(S), RDFS(FA) provides a separated vocabulary as follows. For ease of presentation, this thesis does not cover blank nodes, which can be handled similar to the way that URI references are handled.

Definition 4.1. (RDFS(FA) Vocabulary) An *RDFS(FA) vocabulary* $\mathbf{V}$ consists of a set of literals $\mathbf{V}_{\mathrm{L}}$, and seven sets of pairwise disjoint URI references, which are $\mathbf{V}_{\mathrm{C}}$ (class URIrefs), $\mathbf{V}_{\mathrm{D}}$ (datatype URIrefs), $\mathbf{V}_{\mathrm{AP}}$ (abstract property URIrefs), $\mathbf{V}_{\mathrm{DP}}$ (datatype

property URIrefs), $\mathbf{V}_{\mathrm{ANP}}$ (annotation property URIrefs), $\mathbf{V}_{\mathrm{I}}$ (individual URIrefs) and $\mathbf{V}_{\mathrm{S}}$ ={fa:Literal, fa:type_1, fa:type_2, ...}. $\mathbf{V}_{\mathrm{C}}$ ($\mathbf{V}_{\mathrm{AP}}$) is divided into disjoint *stratified subsets* $\mathbf{V}_{\mathrm{C_1}}$, $\mathbf{V}_{\mathrm{C_2}}$, ...($\mathbf{V}_{\mathrm{AP1}}$, $\mathbf{V}_{\mathrm{AP2}}$,...) of class (abstract property) URIrefs in strata 1,2, ..., where we use a subscript i to indicate URI references in the stratum i.

Let i ≥ 0; the built-in class URIrefs of RDFS(FA) are fa:$\mathrm{Resource}_{\mathrm{i+1}}$, fa:$\mathrm{Class}_{\mathrm{i+2}}$, fa:$\mathrm{Property}_{\mathrm{i+2}}$, fa:$\mathrm{AbstractProperty}_{\mathrm{i+2}}$, fa:DatatypeProperty and fa:AnnotationProperty; the built-in abstract property URIrefs of RDFS(FA) are fa:$\mathrm{subClassOf}_{\mathrm{i+2}}$, fa:$\mathrm{subPropertyOf}_{\mathrm{i+2}}$, fa:$\mathrm{domain}_{\mathrm{i+2}}$ and fa:$\mathrm{range}_{\mathrm{i+2}}$; the built-in annotation property URIrefs of RDFS(FA) are fa:label, fa:comment, fa:seeAlso and fa:isDefinedBy; other built-in URIrefs of RDFS(FA) are those in $\mathbf{V}_{\mathrm{S}}$. We use a superscript b (u) together with $\mathbf{V}_{\mathrm{C}}$, $\mathbf{V}_{\mathrm{AP}}$ and their stratified subsets, to indicate the corresponding subsets of built-in (user-defined) URI references. ◇

Formally, the semantics of RDFS(FA) individuals, classes, datatypes, abstract properties, datatype properties and typed literals is defined in terms of an interpretation as follows.

Definition 4.2. (RDFS(FA) Interpretation) Given an RDFS(FA) vocabulary $\mathbf{V}$, an *RDFS(FA) interpretation* w.r.t. a datatype map $\mathbf{M}_d$ is a tuple of the form $\mathcal{J} = (\Delta^{\mathcal{J}}, \cdot^{\mathcal{J}})$, where $\Delta^{\mathcal{J}}$ is the domain (a non-empty set) and $\cdot^{\mathcal{J}}$ is the interpretation function. Let $\Delta_A^{\mathcal{J}}$ be the abstract domain (a non-empty set), i a non-negative integer, $\Delta_{A_{\mathrm{i}}}^{\mathcal{J}}$ the abstract domain in stratum i and $\Delta_{\mathbf{D}}$ the domain (a non-empty set) for datatypes in a datatype map $\mathbf{M}_d$, $\mathcal{J}$ satisfies the following conditions:

1. $\Delta_A^{\mathcal{J}} = \bigcup_{\mathrm{i}\geq 0} \Delta_{A_{\mathrm{i}}}^{\mathcal{J}}$,
2. $\Delta_{A_{\mathrm{i}}}^{\mathcal{J}} \cap \Delta_{A_{\mathrm{j}}}^{\mathcal{J}} = \emptyset$ $(0 \leq \mathrm{i} < \mathrm{j})$,
3. $\Delta_{\mathbf{D}} \cap \Delta_A^{\mathcal{J}} = \emptyset$,
4. $\bigcup_{\forall d=\mathbf{M}_d(u)} V(d) \subseteq \Delta_{\mathbf{D}}$,
5. $\Delta^{\mathcal{J}} = \Delta_A^{\mathcal{J}} \cup \Delta_{\mathbf{D}}$,
6. $\forall \mathsf{a} \in \mathbf{V}_{\mathrm{I}} : \mathsf{a}^{\mathcal{J}} \in \Delta_{A_0}^{\mathcal{J}}$,
7. $\forall \mathsf{C} \in \mathbf{V}_{\mathrm{C_{i+1}}} : \mathsf{C}^{\mathcal{J}} \subseteq \Delta_{A_{\mathrm{i}}}^{\mathcal{J}}$,
8. $\forall p \in \mathbf{V}_{\mathrm{APi+1}} : p^{\mathcal{J}} \subseteq \Delta_{A_{\mathrm{i}}}^{\mathcal{J}} \times \Delta_{A_{\mathrm{i}}}^{\mathcal{J}}$,
9. $\forall n \in \mathbf{V}_{\mathrm{ANP}} : \langle x, y\rangle \in n^{\mathcal{J}} \rightarrow y \in \Delta_{\mathbf{D}}$,

10. $\forall r \in \mathbf{V}_{\mathrm{DP}} : \ r^{\mathcal{J}} \subseteq \Delta_{A0}^{\mathcal{J}} \times \Delta_{\mathbf{D}}$,
11. $\mathrm{fa{:}\,type_{i+1}}^{\mathcal{J}} \subseteq \Delta_{A\mathrm{i}}^{\mathcal{J}} \times \mathrm{fa{:}\,Class_{i+2}}^{\mathcal{J}}$,
12. $\mathrm{fa{:}\,Literal}^{\mathcal{J}} = \Delta_{\mathbf{D}}$,
13. $\mathrm{fa{:}\,Resource_{i+1}}^{\mathcal{J}} = \Delta_{A\mathrm{i}}^{\mathcal{J}}$,
14. $\forall \mathsf{C} \in \mathbf{V}_{\mathrm{C_{i+1}}} : \ \mathsf{C}^{\mathcal{J}} \in \mathrm{fa{:}\,Class_{i+2}}^{\mathcal{J}}$,
15. $\forall p \in \mathbf{V}_{\mathrm{APi+1}} : \ p^{\mathcal{J}} \in \mathrm{fa{:}\,AbstractProperty_{i+2}}^{\mathcal{J}}$,
16. $\forall r \in \mathbf{V}_{\mathrm{DP}} : \ r^{\mathcal{J}} \in \mathrm{fa{:}\,DatatypeProperty}^{\mathcal{J}}$,
17. $\forall n \in \mathbf{V}_{\mathrm{ANP}} : \ n^{\mathcal{J}} \in \mathrm{fa{:}\,AnnotationProperty}^{\mathcal{J}}$,
18. $\mathrm{fa{:}\,Class_{i+2}}^{\mathcal{J}} \subseteq \mathrm{fa{:}\,Resource_{i+2}}^{\mathcal{J}}$ and $\mathrm{fa{:}\,Property_{i+2}}^{\mathcal{J}} \subseteq \mathrm{fa{:}\,Resource_{i+2}}^{\mathcal{J}}$,
19. $\mathrm{fa{:}\,AbstractProperty_{i+2}}^{\mathcal{J}} \subseteq \mathrm{fa{:}\,Property_{i+2}}^{\mathcal{J}}$ and $\mathrm{fa{:}\,DatatypeProperty}^{\mathcal{J}} \subseteq \mathrm{fa{:}\,Property_2}^{\mathcal{J}}$,
20. $\forall u \in \mathbf{V}_{\mathrm{D}}$, if $\mathbf{M}_d(u) = d$, then
    (a) $u^{\mathcal{J}} = V(d)$,
    (b) if $v \in L(d)$, then $(\text{“}v\text{”}\hat{}\hat{}u)^{\mathcal{J}} = L2V(d)(v)$,
    (c) if $v \notin L(d)$, then $(\text{“}v\text{”}\hat{}\hat{}u)^{\mathcal{J}}$ is undefined;[1]

    otherwise, $u^{\mathcal{J}} \subseteq \Delta_{\mathbf{D}}$ and $\text{“}v\text{”}\hat{}\hat{}u \in \Delta_{\mathbf{D}}$. ◇

There are some remark on Definition 4.2:

1. Conditions 1-5 require that the domain (of universe) $\Delta^{\mathcal{J}}$ in RDFS(FA) is disjointly divided into the abstract domain $\Delta_A^{\mathcal{J}}$ and the datatype domain $\Delta_{\mathbf{D}}$ (cf. Figure 4.4 on page 92), where $\Delta_A^{\mathcal{J}}$ is further disjointly divided into sub-abstract domains $\Delta_{A\mathrm{i}}^{\mathcal{J}}$ in different strata (layers) and $\Delta_{\mathbf{D}}$ is a super-set of the union of the value spaces of all the datatypes in $\mathbf{M}_d$.

[1]The reader is invited to note that there is a tiny difference between OWL and RDF datatyping in handling typed literals with invalid lexical forms. Like RDFS(FA), OWL datatyping treats them as contradictions; RDF datatyping does not, but interprets them as some non-data-valued objects; cf. Section 3.3.

2. Condition 6-11 describe the interpretations of individuals, classes, abstract properties, annotation properties, datatype properties and instance-of properties, respectively. For example, Condition 6 ensures that each individual URIref $\mathtt{a} \in \mathbf{V}_{\mathrm{I}}$ are interpreted as a member of $\Delta_{A0}^{\mathcal{J}}$. We invite the reader to refer to the 'Design Philosophy' section (on page 87) for informal descriptions of these interpretations.

3. Conditions 12-19 are extra semantic constraints on the built-in URIrefs in $\mathbf{V}_{\mathrm{S}}$ and $\mathbf{V}_{\mathrm{C}}$. Condition 12 ensures that $\mathrm{fa{:}Literal}$ is interpreted as the datatype domain $\Delta_{\mathbf{D}}$, while condition 13 ensures that $\mathrm{fa{:}Resource_{i+1}}$ is interpreted as the abstract domain $\Delta_{A\mathrm{i}}^{\mathcal{J}}$. Conditions 14-17 ensures that the interpretations of $\mathrm{fa{:}Class_{i+2}}$, $\mathrm{fa{:}AbstractProperty_{i+2}}$, $\mathrm{fa{:}DatatypeProperty_{i+2}}$ and $\mathrm{fa{:}AnnotationProperty}$ should contain the interpretations of corresponding URI references. Condition 18 ensure that classes and properties are resources in corresponding strata; condition 19 ensures that abstract properties and properties in corresponding strata, and that datatype properties are in stratum 2.

4. Condition 20 ensures the valid interpretations of datatypes and typed literals. (20a) ensures that a datatype URIref in the datatype map $\mathbf{M}_d$ is interpreted as the value space of the corresponding datatype. (20b)-(20c) ensure that valid typed literals are interpreted as members of the value space of corresponding datatypes, and ill-typed literals cause contradictions. Note that the interpretations of datatype URIrefs that are not in $\mathbf{M}_d$ are interpreted as unknown subsets of $\Delta_{\mathbf{D}}$.

It is straightforward, but important, to observe that the above interpretation of RDFS(FA) is similar to that of DLs (cf. Section 2.1.1), except that it also provides interpretations for meta-classes, meta-properties and annotation properties, which are useful in SW applications (cf. Example 4.3 and Example 4.4).

Figure 4.4 illustrates the interpretation of RDFS(FA):

- Typed literals (such as $\text{“30”}\hat{}\hat{}\mathrm{xsd{:}integer}$) are interpreted as values in the value space corresponding datatypes (such as $V(integer)$). All value spaces of datatypes in $\mathbf{M}_d$ are subset of $\Delta_{\mathbf{D}}$.

- The datatype domain is disjoint with the abstract domain, which is stratified into sub-abstract domains ($\Delta_{A0}^{\mathcal{J}}$, $\Delta_{A1}^{\mathcal{J}}$, etc.).

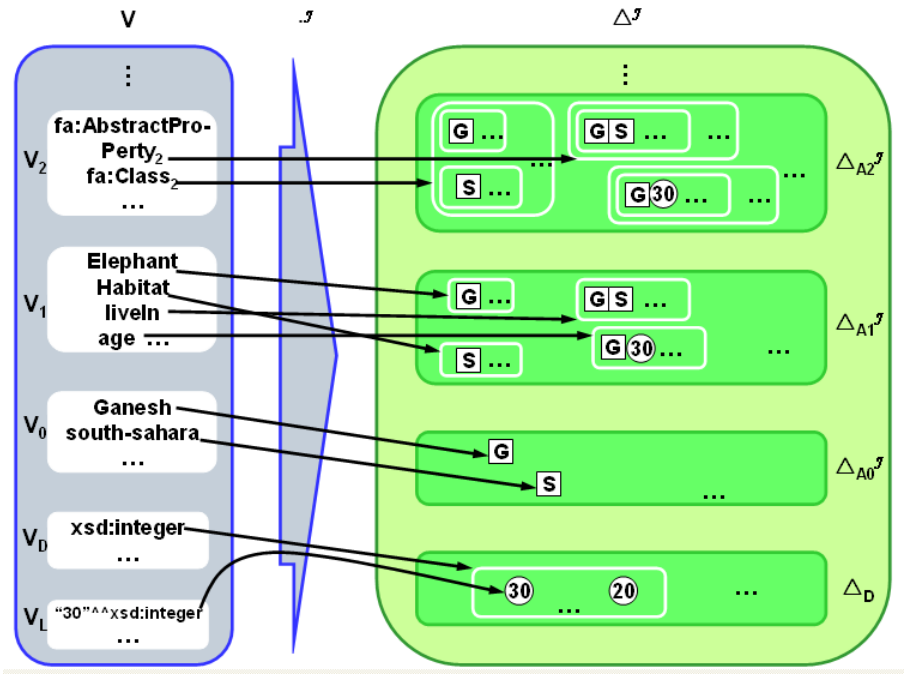

Figure 4.4: RDFS(FA) interpretation

- In stratum 0 (the Instance Layer), object URIrefs (e.g., `elp:Ganesh` and `elp:south-sahara`) are interpreted as objects (i.e., resources in $\Delta_{A0}^{\mathcal{I}}$).
- In stratum 1 (the Ontology Layer), class URIrefs (such as elp:Elephant and elp:Habitat) are interpreted as sets of objects. Abstract property URIrefs (such as *elp:liveIn*) are interpreted as sets of pairs of objects. Datatype property URIrefs (such as *elp:age*) are interpreted as a set of pairs where the first resource (e.g., `elp:Ganesh`) is an object, and the second resource is a datatyped value (e.g., the integer 30).
- In stratum 2 (the Language Layer), fa:Class$_2$ is interpreted as a set of sets of objects, and fa:AbstractProperty$_2$ is interpreted as a set of sets of pairs of objects.

4.1.3 RDFS(FA) Ontologies

Informally speaking, an RDFS(FA) ontology is a set of RDFS(FA) axioms, which are basically RDF triples (in N3 syntax)[2] with extra syntactic rules to disallow arbitrary use of its built-in vocabulary and to allow the use of meta-classes and meta-properties

[2]Here we use the N3 syntax, instead of the RDF/XML syntax, as it is more compact.

in specified layers and the use of annotation properties.

Definition 4.3. (RDFS(FA) Ontologies) Given an RDFS(FA) vocabulary $\mathbf{V}$, let i be a non-negative integer, $\mathtt{a}, \mathtt{b} \in \mathbf{V}_{\mathrm{I}}$, $\mathsf{D}_1 \in \mathbf{V}_{\mathrm{C}_1}$, $\mathsf{C} \in \mathbf{V}^u_{\mathrm{C}i+1}$, $\mathsf{D} \in \mathbf{V}_{\mathrm{C}_{i+1}}$, $\mathsf{H} \in \mathbf{V}_{\mathrm{C}_{i+2}}$, $p_1 \in \mathbf{V}^u_{\mathrm{AP}1}$, $p \in \mathbf{V}^u_{\mathrm{AP}i+1}$, $q \in \mathbf{V}_{\mathrm{APi+1}}$ $r, s \in \mathbf{V}_{\mathrm{DP}}$, $q' \in \mathbf{V}^u_{\mathrm{AP}i+2}$, $u \in \mathbf{V}_{\mathrm{D}}$, $X, Y \in \mathbf{V}^u_{\mathrm{C}i+1} \cup \mathbf{V}^u_{\mathrm{AP}i+1}$, $n \in \mathbf{V}_{\mathrm{ANP}}$ and $w \in \mathbf{V} \setminus \mathbf{V}_{\mathrm{L}}$.

An RDFS(FA) ontology is a finite, possibly empty, set of axioms of the form:

1. [C fa:$\mathrm{subClassOf}_{\mathrm{i+2}}$ D .], called *class inclusions*,
2. [p fa:$\mathrm{subPropertyOf}_{\mathrm{i+2}}$ q .], called *abstract property inclusions*;
3. [r fa:$\mathrm{subPropertyOf}_2$ s .], called *datatype property inclusions*;
4. [p fa:$\mathrm{domain}_{\mathrm{i+2}}$ D .], called *abstract property domain restrictions*;
5. [r fa:domain_2 D_1 .], called *datatype property domain restrictions*;
6. [p fa:$\mathrm{range}_{\mathrm{i+2}}$ D .], called *abstract property range restrictions*;
7. [r fa:range_2 u .], called *datatype property range restrictions*;
8. [$\mathtt{a}$ fa:type_1 D_1 .], called *class assertions*,
9. [$\mathtt{a}$ p_1 $\mathtt{b}$.], called *abstract property assertions*,
10. [$\mathtt{a}$ r "v"^^u .], called *datatype property assertions*,
11. [X fa:$\mathrm{type}_{\mathrm{i+2}}$ H .], called *meta class assertions*,
12. [X q' Y .], called *meta abstract property assertions*,
13. [w n "v"^^u .], called *annotation property assertions*,
14. [n rdf:type fa:AnnotationProperty.], called *annotation property declarations*.

An interpretation $\mathcal{J}$ *satisfies* an RDFS(FA) axiom φ, written as $\mathcal{J} \models \varphi$, if $\mathcal{J}$ meets certain semantic condition:

1. $\mathcal{J} \models$ [C fa:$\mathrm{subClassOf}_{\mathrm{i+2}}$ D .] if $\mathsf{C}^{\mathcal{J}} \subseteq \mathsf{D}^{\mathcal{J}}$;
2. $\mathcal{J} \models$ [p fa:$\mathrm{subPropertyOf}_{\mathrm{i+2}}$ q .] if $p^{\mathcal{J}} \subseteq q^{\mathcal{J}}$;
3. $\mathcal{J} \models$ [r fa:$\mathrm{subPropertyOf}_2$ s .] if $r^{\mathcal{J}} \subseteq s^{\mathcal{J}}$;

4. $\mathcal{J} \models [p \text{ fa:domain}_{\text{i+2}} \mathsf{D}]$ if $\forall x.\langle x, y\rangle \in p^{\mathcal{J}} \rightarrow x^{\mathcal{J}} \in \mathsf{D}^{\mathcal{J}}$;
5. $\mathcal{J} \models [r \text{ fa:domain}_2 \mathsf{D_1} .]$ if $\forall x.\langle x, t\rangle \in r^{\mathcal{J}} \rightarrow x^{\mathcal{J}} \in \mathsf{D}_1^{\mathcal{J}}$;
6. $\mathcal{J} \models [\mathsf{p} \text{ fa:range}_{\text{i+2}} \mathsf{D} .]$ if $\forall y.\langle x, y\rangle \in p^{\mathcal{J}} \rightarrow y^{\mathcal{J}} \in \mathsf{D}^{\mathcal{J}}$;
7. $\mathcal{J} \models [r \text{ fa:range}_2 \mathsf{u} .]$ if $\forall t.\langle x, t\rangle \in r^{\mathcal{J}} \rightarrow t^{\mathcal{J}} \in u^{\mathcal{J}}$;
8. $\mathcal{J} \models [\mathsf{a} \text{ fa:type}_1 \mathsf{C_1} .]$ if $\mathsf{a}^{\mathcal{J}} \in \mathsf{C_1}^{\mathcal{J}}$;
9. $\mathcal{J} \models [\mathsf{a}\ p_1\ \mathsf{b} .]$ if $\langle \mathsf{a}^{\mathcal{J}}, \mathsf{b}^{\mathcal{J}}\rangle \in p_1{}^{\mathcal{J}}$;
10. $\mathcal{J} \models [\mathsf{a}\ r\ \text{“}v\text{”}\hat{}\hat{}u .]$ if $\langle \mathsf{a}^{\mathcal{J}}, (\text{“}v\text{”}\hat{}\hat{}u)^{\mathcal{J}}\rangle \in r^{\mathcal{J}}$;
11. $\mathcal{J} \models [X \text{ fa:type}_{\text{i+2}} \mathsf{H} .]$ if $X^{\mathcal{J}} \in \mathsf{H}^{\mathcal{J}}$;
12. $\mathcal{J} \models [X\ q'\ Y .]$ if $\langle X^{\mathcal{J}}, Y^{\mathcal{J}}\rangle \in q'^{\mathcal{J}}$;
13. $\mathcal{J} \models [w\ n\ \text{“}v\text{”}\hat{}\hat{}u .]$ if $(\text{“}v\text{”}\hat{}\hat{}u)^{\mathcal{J}} \in \Delta_{\mathbf{D}}$,
14. $\mathcal{J} \models [\ n \text{ rdf:type fa:AnnotationProperty}.]$ if $n^{\mathcal{J}} \in \text{fa:AnnotationProperty}^{\mathcal{J}}$.

An interpretation $\mathcal{J}$ *satisfies* an ontology $\mathcal{O}$, written as $\mathcal{J} \models \mathcal{O}$, iff it satisfies all the axioms in $\mathcal{O}$; $\mathcal{O}$ is *satisfiable* (*unsatisfiable*), written as $\mathcal{O} \not\models \bot$ ($\mathcal{O} \models \bot$), iff there exists (does not exist) such an interpretation $\mathcal{J}$.

Given an RDFS(FA) axiom φ, $\mathcal{O}$ *entails* φ, written as $\mathcal{O} \models \varphi$, iff for all models $\mathcal{J}$ of $\mathcal{O}$ we have $\mathcal{J} \models \varphi$. An ontology $\mathcal{O}$ *entails* an ontology $\mathcal{O}'$, written as $\mathcal{O} \models \mathcal{O}'$, iff for all models $\mathcal{J}$ of $\mathcal{O}$ we have $\mathcal{J} \models \mathcal{O}'$. Two ontologies $\mathcal{O}$ and $\mathcal{O}'$ are *equivalent*, written as $\mathcal{O} \equiv \mathcal{O}'$, iff $\mathcal{O} \models \mathcal{O}'$ and $\mathcal{O}' \models \mathcal{O}$. ◇

We invite the reader to note that RDFS(FA) axioms of the form 1-8 and 11 are RDFS statements with extra (subscript) information specifying the strata that the related resources belong to. For example, [C $\text{fa:subClassOf}_{\text{i+2}}$ D .] requires that the concepts C and D should be on stratum i+1. Furthermore, RDFS(FA) provides the use of three kinds of properties: abstract properties, datatype properties and annotation properties (cf. RDFS(FA) axioms of the form 9,12, 10 and 13). Last but not least, let us point out that rdf:type is used in annotation property declarations because annotation property are not bound to any stratum.

The interpretation of class inclusions, property inclusions in stratum 1 as well as class assertions and property assertions are exactly the same as the corresponding DL axioms that we have seen in Chapter 2. RDFS(FA) meta-axioms are very similar to the

```
@prefix fa:   <http://dl-web.man.ac.uk/rdfsfa/ns#>
@prefix elp:   <http://example.org/Animal#>

elp:Animal  fa:type2  fa:Class2  .
elp:Habitat  fa:type2  fa:Class2  .
elp:Elephant  fa:type2  fa:Class2  ;  fa:subClassOf2  elp:Animal  .
elp:liveIn  fa:type2  fa:AbstractProperty2  ;
            fa:domain2  elp:Animal  ;  fa:range2  elp:Habitat  .

elp:south-sahara  fa:type1  elp:Habitat  .
elp:Ganesh  fa:type1  elp:Elephant  ;  elp:liveIn  elp:south-sahara  .
```

Figure 4.5: An RDFS(FA) ontology

above, except that they apply on classes and properties that are higher than stratum 1. RDFS(FA) annotation property assertions require that values of annotation properties should be data values in the datatype domain.

Figure 4.5 shows an example RDFS(FA) ontology (cf. Figure 3.3 for the RDFS version). Firstly, the layering structure is clear. elp:Animal, elp:Habitat, elp:Elephant and *elp:liveIn* are in stratum 1 (the Ontology layer), while `elp:Ganesh` and `elp:south-sahara` are in stratum 0 (the Instance Layer). Secondly, RDFS(FA) disallows arbitrary use of its built-in vocabulary. For example, in class inclusion axioms, the subjects can only be only user-defined class URIrefs (such as elp:Animal), which could disallow triples like

fa:Resource$_1$ fa:subClassOf$_2$ elp:Animal .

Furthermore, RDFS(FA) allows users to specify classes and properties in specified strata. For example, the class inclusion axiom

elp:Elephant fa:subClassOf$_2$ elp:Animal .

requires that both elp:Elephant and elp:Animal are class URIrefs in stratum 1. More about the support of meta-classes and the use of class URIrefs as values of annotation properties can be found in Example 4.3 and Example 4.4, respectively.

4.2 RDFS(FA) and OWL

In this section, we show that it is much easier to layer OWL DL, syntactically *and* semantically, on top of RDFS(FA) than on top of RDF(S).

RDFS(FA) Axioms	OWL Axioms (Abstract Syntax)	OWL Axioms (RDF Syntax)
[C_1 fa:subClassOf$_2$ D_1 .]	SubClassOf(C_1 D_1)	[C_1 rdfs:subClassOf D_1 .]
[p_1 fa:subPropertyOf$_2$ q_1 .]	SubPropertyOf(p_1 q_1)	[p_1 rdfs:subPropertyOf q_1 .]
[r_1 fa:subPropertyOf$_2$ s_1 .]	SubPropertyOf(r_1 s_1)	[r_1 rdfs:subPropertyOf s_1 .]
[p_1 fa:domain$_2$ D_1 .]	ObjectProperty(p_1 domain(D_1))	[p_1 rdfs:domain D_1 .]
[r_1 fa:domain$_2$ D_1 .]	DatatypeProperty(r_1 domain(D_1))	[r_1 rdfs:domain D_1 .]
[p_1 fa:range$_2$ D_1 .]	ObjectProperty(p_1 range(D_1))	[p_1 rdfs:range D_1 .]
[r_1 fa:range$_2$ u .]	DatatypeProperty(r_1 range(u))	[r_1 rdfs:range u .]
[a fa:type$_1$ C_1 .]	Individual(a type(C_1))	[a rdf:type C_1 .]
[a p_1 b .]	Individual(a value(p_1 b))	[a p_1 b .]
[a r_1 "v"^^u .]	Individual(a value(r_1 "v"^^u))	[a r_1 "v"^^u .]
[a fa:type$_1$ fa:Resource$_1$.]	Individual(a)	[a rdf:type rdfs:Resource.]
[C_1 fa:type$_2$ fa:Class$_2$.]	Class(C_1)	[C_1 rdf:type owl:Class .]
[p_1 fa:type$_2$ fa:AbstractProperty$_2$.]	ObjectProperty(p_1)	[p_1 rdf:type owl:ObjectProperty .]
[r_1 fa:type$_2$ fa:DatatypeProperty .]	DatatypeProperty(r_1)	[r_1 rdf:type owl:DatatypeProperty .]

Table 4.1: The mapping between the RDFS(FA) axioms in strata 0-2 and OWL DL axioms

There is a one-to-one bidirectional mapping (as shown in Table 4.1) between the RDFS(FA) axioms in stratum 0-2 and OWL DL axioms in OWL abstract syntax. For example, the RDFS(FA) class inclusion axiom [C_1 fa:subClassOf$_2$ D_1 .] can be mapped to the OWL class axiom (SubClassOf C_1 D_1) and vice versa.

It is easier, therefore, to syntactically layer OWL DL on top of RDFS(FA) than on top of RDF(S), due to the above bidirectional mapping. According to the OWL Semantics and Abstract Syntax document [119], the mapping between OWL DL axioms, or *OWL axioms* for short, and RDF(S) statements is *only* unidirectional, i.e., from OWL axioms to RDF(S) statements. For example, we can map the following OWL axiom (SubClassOf C_1 D_1) to the RDF(S) statement [C_1 rdfs:subClassOf D_1 .], with an implicit OWL constraint, viz., C_1 and D_1 can only be class URIrefs, but not URIrefs for properties or individuals, etc. An RDF(S) statement [C_1 rdfs:subClassOf D_1 .] without such constraint, however, cannot be correctly mapped to the OWL axiom (SubClassOf C_1 D_1). Interestingly, in the corresponding RDFS(FA) axioms these kinds of implicit constraints are made explicit via the syntactic constraints of the RDFS(FA) class axioms (cf. Definition 4.3). For example, the RDFS(FA) class inclusion axiom [C_1 fa:subClassOf$_2$ D_1 .] (in place of [C_1 rdfs:subClassOf D_1 .]) requires that both C_1 and D_1 are class URIrefs in stratum 1. This explains why the above bidirectional mapping (listed in Table 4.1) is possible.

Furthermore, it can be shown (by the following theorem) that the above bidirectional mapping is a semantics-preserved mapping.

Theorem 4.4. *The bidirectional mapping, shown in Table 4.1, between the RDFS(FA) axioms in strata 0-2 and the corresponding OWL axioms in the OWL abstract syntax*

is a satisfiability-preserved mapping.

Proof: Given a datatype map $\mathbf{M}_d$, we only need to show that there exists an interpretation $\mathcal{J}$ satisfying all the listed RDFS(FA) axioms iff there exists an interpretation $\mathcal{I}$ satisfying all the corresponding OWL DL axioms.

For the *only-if* direction, given an RDFS(FA) interpretation $\mathcal{J} = (\Delta^{\mathcal{J}}, \cdot^{\mathcal{J}})$ for $\mathbf{V}$ w.r.t. $\mathbf{M}_d$, we can construct an OWL DL interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ as follows: $\Delta^{\mathcal{I}} = \Delta_{A_0}^{\mathcal{J}}$ and $\Delta_{\mathbf{D}owl} = \Delta_{\mathbf{D}fa}$; for each class URIref (in stratum 1) C, $\mathsf{C}^{\mathcal{I}} = \mathsf{C}^{\mathcal{J}}$; for each datatype URIref (in stratum 1) u, $u^{\mathcal{I}} = u^{\mathcal{J}}$; for each abstract (object) property URIref p (in stratum 1), $p^{\mathcal{I}} = p^{\mathcal{J}}$; for each datatype property URIref r, $r^{\mathcal{I}} = r^{\mathcal{J}}$.

Now we only need to show that if $\mathcal{J}$ satisfies an RDFS(FA) axiom ϕ_1 in the first column of Table 4.1, we have I satisfies the corresponding OWL DL axiom ϕ_2 in the second column of Table 4.1. According to the semantics of RDFS(FA) (Definition 4.2 on page 89) and RDFS(FA) axioms (Definition 4.3 on page 93), the semantics of OWL constructs (Tables 3.1, datatypes (Definition 3.12 on page75) 3.2 and 3.6) and axioms (Tables 3.3 and 3.5), this is trivially true. Therefore, we only give the proof for the class inclusion axiom to illustrate the proofs for the rest: if $\mathcal{J} \models [\mathsf{C}_1\ \mathrm{fa{:}subClassOf}_2\ \mathsf{D}_1\ .]$, according to Definition 4.3, we have ${\mathsf{C}_1}^{\mathcal{J}} \subseteq {\mathsf{D}_1}^{\mathcal{J}}$, hence ${\mathsf{C}_1}^{\mathcal{I}} \subseteq {\mathsf{D}_1}^{\mathcal{I}}$. Thus, $\mathcal{I} \models$ SubClassOf(C_1 D_1).

Similarly, the *if* direction is trivially true, we only need to show that, in an RDFS(FA) interpretation $\mathcal{J}$, we can construct abstract domains for strata higher than stratum 0. Let i ≥ 0. According to the semantics conditions 7, 8, 13 to 19 in Definition 4.2, we have $\mathrm{fa{:}\,Class}_{\mathrm{i}+2}{}^{\mathcal{J}} = 2^{\Delta_{A_{\mathrm{i}}}{}^{\mathcal{J}}}$, $\mathrm{fa{:}\,Property}_2{}^{\mathcal{J}} = 2^{\Delta_{A_0}{}^{\mathcal{J}} \times \Delta_{A_0}{}^{\mathcal{J}}} \cup 2^{\Delta_{A_0}{}^{\mathcal{J}} \times \Delta_{\mathbf{D}}}$, $\mathrm{fa{:}\,Property}_{\mathrm{i}+3}{}^{\mathcal{J}} = 2^{\Delta_{A_{\mathrm{i}+1}}{}^{\mathcal{J}} \times \Delta_{A_{\mathrm{i}+1}}{}^{\mathcal{J}}}$ and $\Delta_{A_{\mathrm{i}+1}}{}^{\mathcal{J}} = \mathrm{fa{:}\,Resource}_{\mathrm{i}+2} = \mathrm{fa{:}\,Class}_{\mathrm{i}+2}{}^{\mathcal{J}} \cup \mathrm{fa{:}\,Property}_{\mathrm{i}+2}{}^{\mathcal{J}}$. Hence we have $\Delta_{A_1}{}^{\mathcal{J}} = 2^{\Delta_{A_0}{}^{\mathcal{J}}} \cup 2^{\Delta_{A_0}{}^{\mathcal{J}} \times \Delta_{A_0}{}^{\mathcal{J}}} \cup 2^{\Delta_{A_1}{}^{\mathcal{J}} \times \Delta_{\mathbf{D}}}$ and $\Delta_{A_{\mathrm{i}+2}}{}^{\mathcal{J}} = 2^{\Delta_{A_{\mathrm{i}+1}}{}^{\mathcal{J}}} \cup 2^{\Delta_{A_{\mathrm{i}+1}}{}^{\mathcal{J}} \times \Delta_{A_{\mathrm{i}+1}}{}^{\mathcal{J}}}$. □

We claim that OWL DL can be semantically layered on top of RDFS(FA). Firstly, [113] shows that RDFS(FA) does not have the semantic problems [113, 116, 117, 69] that RDF(S) has, when we layer OWL on top of it. Secondly, OWL DL reserves the semantics of RDFS(FA) built-in primitives; e.g., Table 4.2 shows that owl:Thing is equivalent to fa:Resource$_1$, Table 4.3 shows that OWL DL uses some RDFS modelling primitives with RDFS(FA) semantics, instead of RDFS semantics. Furthermore, OWL DL extends RDFS(FA) in strata 0-2 by introducing new class descriptions (such as class intersections), new property descriptions (such as inverse properties) and new axioms (such as functional axioms for properties). Most importantly, Theorem 4.4 shows that OWL DL preserves the meaning of the RDFS(FA) axioms in strata 0-2

OWL Modelling Primitives	RDFS(FA) Modelling Primitives
owl: Thing	fa: Resource_1
owl: Class	fa: Class_2
owl: ObjectProperty	fa: $\text{AbstractProperty}_2$
owl: DatatypeProperty	fa: DatatypeProperty

Table 4.2: OWL DL preserves the semantics of built-in RDFS(FA) primitives

RDFS Modelling Primitives	RDFS(FA) Modelling Primitives
rdfs: subClassOf	fa: subClassOf_2
rdfs: subPropertyOf	fa: subPropertyOf_2
rdfs: domain	fa: domain_2
rdfs: range	fa: range_2

Table 4.3: OWL DL uses RDFS primitives with RDFS(FA) semantics

shown in Table 4.1. Last but not least, annotation properties and ontology properties in OWL DL extends RDFS(FA) annotation properties, relating resources with not only data literal but also URI references.[3] It is possible to have a new sub-language of OWL — OWL FA, which extends OWL DL with meta classes/properties and provide better support for annotation properties.

Although Theorem 4.4 indicates that we can use DL reasoners to reason with RDFS(FA) axioms in strata 0-2, or possibly in every three adjacent strata, providing reasoning support for RDFS(FA) is beyond the scope of this thesis.

4.3 A Clarified Vision of the Semantic Web

In this chapter we have presented RDFS(FA), an alternative to RDFS with a DL-style semantics, so as to repair the broken link between RDF(S) and OWL.

RDFS(FA), consequently, provides a clarified vision of the Semantic Web: RDF is *only* a standard syntax for SW annotations and languages (i.e., the built-in semantics of RDF triples is disregarded), and the meaning of annotations comes from either external agreements (such as Dublin Core) or ontologies (which are more flexible), both of which are supported by RDFS(FA).

On the one hand, RDFS(FA) is an ontology language that provides a UML-like layered style for using RDFS; it is more intuitive, and easier to understand and use by

[3] We are not convinced, however, that it is proper to use annotation properties to relate resources with URI references.

users. Most importantly, strata 0-2 have a standard model theoretic semantics, s.t. more expressive FOL ontology languages, such as the W3C standard OWL, can be layered on top of them and are compatible with RDFS(FA)'s metamodeling architecture.

On the other hand, RDFS(FA) allows the use of Dublin Core information properties as annotation properties. In RDFS(FA), all resources can have annotation properties, such that 'anyone can say anything about anything'. Typed literals are used to precisely represent values of annotation properties, such as "1999-05-31"^^xsd:date for the dc:date property and "bk:Lion"^^xsd:anyURI for the dc:subject property. In particular, the use of URIrefs as values of annotation properties can enable SW applications to make use of URIrefs of ontology elements, such as classes, in the results of various ontology inferences (cf. Example 4.4 on page 85).

In general, introducing RDFS(FA) as a sub-language of RDF(S) makes it more flexible to layer languages on top of RDF(S), which surely solidifies RDF(S)'s proposed role as the base of the Semantic Web. The Semantic Web tower has become clearer, easier to understand and formalise. The remaining layers of the Semantic Web and the extensions of OWL, such as the Semantic Web Rule Language (SWRL), can be more easily defined in the metamodeling architecture of RDFS(FA).

Having addressed the layering problem and clarified the vision of the Semantic Web, can we now use the DL reasoning services to completely support OWL and DAML+OIL? Not yet — we still need to solve the second problem mentioned in Section 3.4, i.e., the problem of how to extend DL datatype reasoning to support OWL, DAML+OIL and their extensions that provide customised datatypes and datatype predicates. We are going to investigate this problem and provide a solution in the coming chapters.

Chapter Achievements

- RDFS(FA) is an ontology language that has well-formed syntax, DL-style model theoretic semantics and provides a layered style of using RDFS. It provides a firm semantic foundation for DL-based Web ontology languages.
- The bidirectional one-to-one mapping between RDFS(FA) axioms in strata 0-2 and OWL DL axioms enables RDFS(FA)-agents and OWL DL-agents to more easily communicate with each other.
- Annotation properties in RDFS(FA) enable SW applications to make use of URIrefs of ontology elements, such as classes, in results of various ontology inferences.
- Introducing RDFS(FA) as a sub-language of RDFS clarifies the vision of the Semantic Web and solidifies RDF(S)'s proposed role as the base of the Semantic Web.

Chapter 5

A Unified Formalism for Datatypes and Predicates

Chapter Aims:

- To propose a general formalism to unify existing ontology-related datatype and predicate formalisms and to overcome their limitations.
- To investigate, in our formalism, a family of decidable Description Logics that provide customised datatypes and datatype predicates.

Chapter Plan

This chapter introduces the datatype group approach, which provides a unified formalism for datatypes and predicates, so as to support all the new datatype features of OWL and to enable combining DLs with customised datatypes and predicates.

5.1 Datatype Groups

Existing ontology-related formalisms either focus on either datatypes (such as RDF(S) and OWL datatyping) or predicates (such as the concrete domain and the type system approach). The datatype group approach to be presented in this chapter provides a

unified formalism (Section 5.1.1) for datatypes and datatype predicates. In this formalism, datatype expressions (Section 5.1.2) can be constructed to represent customised datatypes and datatype predicates, which are very useful in SW and ontology applications (cf. Section 1.3).

5.1.1 Unifying Datatypes and Predicates

Datatypes and datatype predicates are closely related to each other, but they serve different purposes. A datatype d is characterised by its lexical space $L(d)$, value space $V(d)$ and lexical-to-value mapping $L2V(d)$ (cf. Definition 3.6 on page 71). It can be used to represent its member values through typed literals (cf. Definition 3.7 on page 71). A datatype predicate, however, is defined otherwise.

Definition 5.1. (Datatype Predicate) A *datatype predicate* (or simply *predicate*) p is characterised by an arity $a(p)$, or a minimum arity $a_{min}(p)$ if p can have multiple arities, and a predicate extension (or simply *extension*) $E(p)$. ◇

Predicates are mainly used to represent constraints over values of datatypes which they are defined over. Here are some examples:

Example 5.1 Datatype Predicates

1. $>^{int}_{[20]}$ is a (unary) predicate with $a(>^{int}_{[20]}) = 1$ and $E(>^{int}_{[20]}) = \{i \in V(integer) \mid i > L2V(integer)(\text{“20”})\}$. This example shows that predicates are defined based on datatypes (e.g., $integer$) and their values (e.g., the integer $L2V(integer)$ (“20”), i.e., 20).

2. $=^{int}$ is a (binary) predicate with arity $a(=^{int}) = 2$ and extension $E(=^{int}) = \{\langle i_1, i_2\rangle \in V(integer)^2 \mid i_1 = i_2\}$.

3. $+^{int}$ is a predicate with minimum arity $a_{min}(+^{int}) = 3$ and extension $E(+^{int}) = \{\langle i_0, \ldots, i_n\rangle \in V(integer)^n \mid i_0 = i_1 + \ldots + i_n\}$.[1] ♢

Datatypes can be regarded as *special* predicates with arity 1 and predicate extensions equal to their value spaces; e.g., the datatype $integer$ can be seen as a predicate with arity $a(integer) = 1$ and predicate extension $E(integer) = V(integer)$. They

[1] Note that some XML Schema user-derived datatypes (such as ‘GreaterThan20’) defined in Example 3.2 on page 70) can represent unary predicates (such as $>^{int}_{[20]}$), but they can not represent predicates with arities greater than 1, such as $=^{int}$ and $+^{int}$.

are special because they have lexical spaces and lexical-to-value mappings that ordinary predicates do not have.

Based on the above connection between datatypes and predicates, we can extend the definitions of datatype maps (cf. Section 3.3.2), supported/unsupported datatype URIrefs to predicate maps and supported/unsupported predicate URIrefs as follows.

Definition 5.2. (Predicate Map) We consider a predicate map $\mathbf{M}_p$ that is a partial mapping from predicate URI references to predicates. ◇

Example 5.2 Predicate Map $\mathbf{M}_{p_1} = \{\langle \text{xsd:string}, string\rangle, \langle \text{xsd:integer}, integer\rangle, \langle \text{owlx:integerEquality}, =^{int}\rangle, \langle \text{owlx:integerGreaterThanx=n}, >^{int}_{[n]}\rangle\}$, where xsd:string, xsd:integer, $\text{owlx:integerEquality}$ and $\text{owlx:integerGreaterThanx=n}$ are predicate URI references, $string$, $integer$ and $>^{int}_{[n]}$ are unary predicates, and $=^{int}$ is a binary predicate. Note that, by '$>^{int}_{[n]}$', we mean there exist a predicate $>^{int}_{[n]}$ for each integer $L2V(integer)(\text{“}n\text{”})$, which is represented by the predicate URIref $\text{owlx:integerGreaterThanx=n}$. ♢

In general, datatype predicates like $>^{int}_{[n]}$ can be seen as parameterised predicates, where n can be seen as a parameter. In this thesis, we use the "=" character to separate the lexical forms of a parameter from the rest of a URIref of a parameterised predicate. For example, in $\text{owlx:integerGreaterThanx=20}$, the lexical form of the parameter is "20".

Definition 5.3. (Supported and Unsupported Predicates) Given a predicate map $\mathbf{M}_p$, a predicate URIref u is called a *supported predicate URIref w.r.t.* $\mathbf{M}_p$ (or simply *supported predicate URIref*), if there exists a predicate p s.t. $\mathbf{M}_p(u) = p$ (in this case, p is called a *supported predicate* w.r.t. $\mathbf{M}_p$); otherwise, u is called an *unsupported predicate URIref w.r.t.* $\mathbf{M}_p$ (or simply *unsupported predicate URIref*). ◇

For example, $\text{owlx:integerEquality}$ is a supported predicate URIref w.r.t. $\mathbf{M}_{p_1}$ presented in Example 5.2, while $\text{owlx:integerInequality}$ is an unsupported predicate URIref w.r.t. $\mathbf{M}_{p_1}$. Therefore, according to $\mathbf{M}_{p_1}$, we know neither the arity nor the extension of the predicate that $\text{owlx:integerInequality}$ represents. Note that we make as few assumptions as possible about unsupported predicates; e.g., we do not even assume that they have a fixed arity.

Now we provide the definition of datatype groups which unifies datatypes and predicates. Informally speaking, a datatype group is a group of supported predicate URIrefs

'wrapped' around a set of base datatype URIrefs. It can potentially be divided into several sub-groups, so that all predicates in a sub-group are defined over the unique base datatype of the sub-group. This allows us to make use of known decidability results about the satisfiability problems of predicate conjunctions of, e.g., those exploited by the admissible (or computable) concrete domains presented in Section 2.4 of [88]. Formally, a datatype group is defined as follows, and sub-groups will be defined in Definition 5.7.

Definition 5.4. (Datatype Group) A *datatype group* $\mathcal{G}$ is a tuple ($\mathbf{M}_p$,$\mathbf{D}_\mathcal{G}$,dom), where $\mathbf{M}_p$ is the *predicate map* of $\mathcal{G}$, $\mathbf{D}_\mathcal{G}$ is the set of *base datatype* URI references of $\mathcal{G}$, and dom is the *declared domain function* of $\mathcal{G}$.

We call $\Phi_\mathcal{G}$ the set of supported predicate URI references of $\mathcal{G}$, i.e., for each $u \in \Phi_\mathcal{G}$, $\mathbf{M}_p(u)$ is defined; we require $\mathbf{D}_\mathcal{G} \subseteq \Phi_\mathcal{G}$. We assume that there exists a unary predicate URI reference $\mathrm{owlx{:}DatatypeBottom} \notin \Phi_\mathcal{G}$.

The declared domain function dom is a mapping defined as follows:

$$\mathsf{dom}(u) \quad = \quad \begin{cases} u & \text{if } u \in \mathbf{D}_\mathcal{G} \\ (d_1, \ldots, d_n), \text{ where } d_1, \ldots, d_n \in \mathbf{D}_\mathcal{G} & \text{if } u \in \Phi_\mathcal{G} \setminus \mathbf{D}_\mathcal{G} \text{ and } a(\mathbf{M}_p(u)) = n \\ \{(\underbrace{d, \ldots, d}_{i \text{ times}}) \mid i \geq n\}, \text{ where } d \in \mathbf{D}_\mathcal{G} & \text{if } u \in \Phi_\mathcal{G} \setminus \mathbf{D}_\mathcal{G} \text{ and } a_{min}(\mathbf{M}_p(u)) = n \end{cases}$$

◇

Definition 5.4 ensures that base datatype URIrefs are among the supported predicate URIrefs ($\mathbf{D}_\mathcal{G} \subseteq \Phi_\mathcal{G}$), and supported predicate URIrefs relate to base datatypes URIrefs via the declared domain function dom. In this formalism, we assume that if a supported predicate does not have a fixed arity (e.g., $+^{int}$), then it relates to only one base datatype (e.g., $integer$).

Example 5.3 Datatype Group $\mathcal{G}_1 = (\mathbf{M}_{p_1}, \mathbf{D}_{\mathcal{G}_1}, \mathsf{dom}_1)$, where $\mathbf{M}_{p_1}$ is defined in Example 5.2, $\mathbf{D}_{\mathcal{G}_1} = \{\mathrm{xsd{:}string}, \mathrm{xsd{:}integer}\}$, and $\mathsf{dom}_1 = \{\langle \mathrm{xsd{:}string}, \mathrm{xsd{:}string}\rangle, \langle \mathrm{xsd{:}integer}, \mathrm{xsd{:}integer}\rangle, \langle \mathrm{owlx{:}integerEquality}, (\mathrm{xsd{:}integer}, \mathrm{xsd{:}integer})\rangle, \langle \mathrm{owlx{:}integerGreaterThan}, \mathrm{xsd{:}integer}\rangle\}$.

According to $\mathbf{M}_{p_1}$, we have $\Phi_{\mathcal{G}_1} = \{\mathrm{xsd{:}string}, \mathrm{xsd{:}integer}, \mathrm{owlx{:}integerEquality}, \mathrm{owlx{:}integerGreaterThan}\}$, hence $\mathbf{D}_{\mathcal{G}_1} \subseteq \Phi_{\mathcal{G}_1}$. ♢

Definition 5.5. (Interpretation of Datatype Groups) A *datatype interpretation* $\mathcal{I}_\mathbf{D}$ of a datatype group $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_\mathcal{G}, \mathsf{dom})$ is a pair $(\Delta_\mathbf{D}, \cdot^\mathbf{D})$, where $\Delta_\mathbf{D}$ (the datatype

domain) is a non-empty set and $\cdot^{\mathbf{D}}$ is a datatype interpretation function, which has to satisfy the following conditions:

1. $\mathrm{rdfs{:}Literal}^{\mathbf{D}} = \Delta_{\mathbf{D}}$ and $\mathrm{owlx{:}DatatypeBottom}^{\mathbf{D}} = \emptyset$;
2. for each plain literal l, $l^{\mathbf{D}} = l \in \mathbf{PL}$ and $\mathbf{PL} \subseteq \Delta_{\mathbf{D}}$;[2]
3. for each supported datatype URIref $u \in \mathbf{D}_{\mathcal{G}}$ (let $d = \mathbf{M}_p(u)$):
   (a) $u^{\mathbf{D}} = E(d) = V(d) \subseteq \Delta_{\mathbf{D}}$,
   (b) if $s \in L(d)$, then $(\text{“}s\text{”}\hat{}\hat{}u)^{\mathbf{D}} = L2V(d)(s)$,
   (c) if $s \notin L(d)$, then $(\text{“}s\text{”}\hat{}\hat{}u)^{\mathbf{D}}$ is not defined;
4. for any two $u_1, u_2 \in \mathbf{D}_{\mathcal{G}}$: $u_1^{\mathbf{D}} \cap u_2^{\mathbf{D}} = \emptyset$;
5. $\forall u \in \Phi_{\mathcal{G}}$, $u^{\mathbf{D}} = E(\mathbf{M}_p(u))$;
6. $\forall u \in \Phi_{\mathcal{G}}$, $u^{\mathbf{D}} \subseteq (\mathsf{dom}(u))^{\mathbf{D}}$:
   (a) if $\mathsf{dom}(u) = (d_1, \ldots, d_n)$ and $a(\mathbf{M}_p(u)) = n$, then $(\mathsf{dom}(u))^{\mathbf{D}} = d_1^{\mathbf{D}} \times \ldots \times d_n^{\mathbf{D}}$,
   (b) if $\mathsf{dom}(u) = \{(\underbrace{d, \ldots, d}_{i \text{ times}}) \mid i \geq n\}$ and $a_{min}(\mathbf{M}_p(u)) = n$, then $(\mathsf{dom}(u))^{\mathbf{D}} = \bigcup_{i \geq n} (d^{\mathbf{D}})^i$;
7. $\forall u \notin \Phi_{\mathcal{G}}$, $u^{\mathbf{D}} \subseteq \bigcup_{n \geq 1} (\Delta_{\mathbf{D}})^n$, and $\text{“}v\text{”}\hat{}\hat{}u \in \Delta_{\mathbf{D}}$.

Moreover, we extend $\cdot^{\mathbf{D}}$ to (relativised) negated predicate URI references $\overline{u}$ as follows:

$$(\overline{u})^{\mathbf{D}} = \begin{cases} \Delta_{\mathbf{D}} \setminus u^{\mathbf{D}} & \text{if } u \in \mathbf{D}_{\mathcal{G}} \\ (\mathsf{dom}(u))^{\mathbf{D}} \setminus u^{\mathbf{D}} & \text{if } u \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}} \\ \bigcup_{n \geq 1} (\Delta_{\mathbf{D}})^n \setminus u^{\mathbf{D}} & \text{if } u \notin \Phi_{\mathcal{G}}. \end{cases}$$ ◇

There are some remarks about Definition 5.5:

1. According to Condition 2 and (3a), the datatype domain $\Delta_{\mathbf{D}}$ is a superset of value spaces of any base datatypes and $\mathbf{PL}$. As a result, unsupported predicate URIrefs can be interpreted intuitively; e.g., given $\mathbf{M}_{p_1}$, “1.278e-3”$\hat{}\hat{}\mathrm{xsd{:}float}$ does *not* have to be interpreted as either an integer, a string or a string with

[2] $\mathbf{PL}$ is the value space for plain literals; cf. Definition 3.12 on page 75.

a language tag. In OWL datatyping, given $\mathbf{M}_{d1}$, "1.278e-3"^^$\mathrm{xsd{:}float}$ has to be interpreted as either an integer, a string or a string with a language tag (cf. Section 3.3.3 on page 77).

2. Base datatype URIrefs and typed literals with base datatype URIrefs are interpreted in the same way as OWL datatyping (cf. Condition 3 and its counterpart in Definition 3.12 on page 75). Value spaces of the base datatypes are disjoint (Condition 4), which is essential to dividing $\Phi_{\mathcal{G}}$ into sub-groups. Note that, in case the value space of a base datatype d_1 is a subset of another base datatype d_2, d_1 can be seen as a unary predicate of d_2; cf. Example 2.4 on page 41 for the 'integer' predicate of the rational datatype.

3. Supported predicate URIrefs are interpreted as the extensions of the predicates they represent (Condition 5), which are subsets of the corresponding declared domains (Condition 6). Extensions of base datatypes are equal to the value spaces of the base datatypes (Condition (3a)).

4. Unsupported predicate URIrefs are not restricted to any fixed arity, and typed literals with unsupported predicates are interpreted as some member of the datatype domain (Condition 7).

5. Supported predicate URIrefs $u \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}$ have relativised negations (w.r.t. their declared domains). For example, $\overline{\mathrm{owlx{:}integerGreaterThanx{=}15}}$, the negated predicate URIref for $\mathrm{owlx{:}integerGreaterThanx{=}15}$, is interpreted as $V(integer) \backslash (\mathrm{owlx{:}integerGreaterThanx{=}15})^{\mathbf{D}}$; therefore, its interpretation includes the integer 5, but not the string "Fred", no matter if $string$ is a base datatype in $\mathbf{D}_{\mathcal{G}}$ or not.

Now we introduce the kind of basic reasoning mechanisms required for a datatype group.

Definition 5.6. (Predicate Conjunction) Let $\mathbf{V}$ be a set of variables, $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_{\mathcal{G}}, \mathsf{dom})$ a datatype group, we consider predicate conjunctions of $\mathcal{G}$ of the form

$$\mathscr{C} = \bigwedge_{j=1}^{k} w_j(v_1^{(j)}, \ldots, v_{n_j}^{(j)}), \tag{5.1}$$

where the $v_i^{(j)}$ are variables from $\mathbf{V}$, w_j are (possibly negated) predicate URI references of the form u_j or $\overline{u_j}$, and if $u_j \in \Phi_{\mathcal{G}}, a(\mathbf{M}_p(u_j)) = n_j$ or $a_{min}(\mathbf{M}_p(u_j)) \leq$

n_j. A predicate conjunction $\mathscr{C}$ is called *satisfiable* iff there exist an interpretation $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ of $\mathcal{G}$ and a function δ mapping the variables in $\mathscr{C}$ to data values in $\Delta_{\mathbf{D}}$ s.t. $\langle \delta(v_1^{(j)}), \ldots, \delta(v_{n_j}^{(j)}) \rangle \in w_j^{\mathbf{D}}$ for all $1 \leq j \leq k$. Such a function δ is called a *solution* for $\mathscr{C}$ (w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$). ◇

Note that we may have negated predicate URIrefs in a predicate conjunction of datatype groups. For example,

$$\begin{aligned} \mathscr{C}_1 \;=\; & \overline{\text{owlx:integerGreaterThanx=38}}(v_1) \wedge \\ & \text{owlx:integerGreaterThanx=12}(v_2) \wedge \\ & \text{owlx:integerEquality}(v_1, v_2) \end{aligned}$$

is a predicate conjunction of $\mathcal{G}_1$ (which is presented in Example 5.3 on page 104). For any interpretation $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ of $\mathcal{G}_1$, we have $26 \in (\text{owlx:integerGreaterThanx=12})^{\mathbf{D}} = E(>^{int}_{[12]}) \subseteq \Delta_{\mathbf{D}}$ and $26 \in (\overline{\text{owlx:integerGreaterThanx=38}})^{\mathbf{D}} = V(integer) \setminus E(>^{int}_{[38]}) \subseteq \Delta_{\mathbf{D}}$. Therefore, the function $\delta = \{v_1 \mapsto 26, v_2 \mapsto 26\}$ is a solution for $\mathscr{C}_1$; in other words, $\mathscr{C}_1$ is satisfiable.

The predicate conjunction over a datatype group $\mathcal{G}$ can possibly be divided into independent sub-conjunctions of sub-groups of $\mathcal{G}$. Informally speaking, a sub-group includes a base datatype URIref and the set of supported predicate URIrefs defined over it.

Definition 5.7. (Sub-Group) Given a datatype group $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_{\mathcal{G}}, \mathsf{dom})$ and a base datatype URI reference $w \in \mathbf{D}_{\mathcal{G}}$, the *sub-group of* w *in* $\mathcal{G}$, abbreviated as $\mathsf{sub\text{-}group}(w, \mathcal{G})$, is defined as:

$$\begin{aligned} \mathsf{sub\text{-}group}(w, \mathcal{G}) = \{ u \in \Phi_{\mathcal{G}} \mid \mathsf{dom}(u) = & \underbrace{(w, \ldots, w)}_{a(\mathbf{M}_p(u)) \text{ times}} \quad \text{or} \\ & \mathsf{dom}(u) = \{ \underbrace{(w, \ldots, w)}_{i \text{ times}} \mid i \geq a_{min}(\mathbf{M}_p(u)) \} \} \end{aligned}$$

◇

Example 5.4 The Sub-Group of xsd:integer in $\mathcal{G}_1$ (presented in Example 5.3) is $\mathsf{sub\text{-}group}(\text{xsd:integer}, \mathcal{G}_1) = \{\text{xsd:integer}, \text{owlx:integerEquality}, \text{owlx:integerGreaterThanx=n}\}$. According to Definition 5.7 and Condition 4 of Definition 5.5, predicate conjunctions over $\mathsf{sub\text{-}group}(\text{xsd:integer}, \mathcal{G}_1)$ and $\mathsf{sub\text{-}group}(\text{xsd:string}, \mathcal{G}_1)$ can be divided into two sub-conjunctions: one over $\mathsf{sub\text{-}group}(\text{xsd:integer}, \mathcal{G}_1)$ and the other over $\mathsf{sub\text{-}group}(\text{xsd:string}, \mathcal{G}_1)$. These sub-conjunctions can be handled

separately if there are no common variables; if there are common variables, there exist contradictions, due to the disjointness of the value spaces $V(integer)$ and $V(string)$. ♢

There is a strong connection between a sub-group and a concrete domain.

Definition 5.8. (Corresponding Concrete Domain) Let $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_\mathcal{G}, \mathsf{dom})$ be a datatype group, $w \in \mathbf{D}_\mathcal{G}$ a base datatype URI reference and $\mathbf{M}_p(w) = \mathcal{D}$. If all the supported predicates in $\mathsf{sub\text{-}group}(w, \mathcal{G})$ have fixed arities, the *corresponding concrete domain* of $\mathsf{sub\text{-}group}(w, \mathcal{G})$ is $(\Delta_\mathcal{D}, \Phi_\mathcal{D})$, where $\Delta_\mathcal{D} := V(\mathcal{D})$ and $\Phi_\mathcal{D} := \{\bot_\mathcal{D}\} \cup \{\mathbf{M}_p(u) | u \in \mathsf{sub\text{-}group}(w, \mathcal{G})\}$, where $\bot_\mathcal{D}$ corresponds to $\overline{w}$. ⋄

Note that concrete domains do not support predicates that can have multiple arities, and sub-groups including such predicates do not have corresponding concrete domains.

Example 5.5 The Corresponding Concrete Domain of $\mathsf{sub\text{-}group}(\mathrm{xsd{:}\,integer}, \mathcal{G}_1)$ is $(\Delta_{integer}, \Phi_{integer})$, where $\Delta_{integer} := V(integer)$ and $\Phi_{integer} := \{\bot_{integer}, integer, =^{int}, >^{int}_{[n]}\}$. Note that the predicate $\bot_{integer}$ corresponds to $\overline{\mathrm{xsd{:}\,integer}}$, the negated form of $\mathrm{xsd{:}\,integer}$. ♢

One of the benefits of introducing the corresponding concrete domain for a sub-group is that if the corresponding concrete domain is admissible, the sub-group is computable.

Lemma 5.9. *Let $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_\mathcal{G}, \mathsf{dom})$ be a datatype group, $w \in \mathbf{D}_\mathcal{G}$ a base datatype URI reference and $\mathcal{D} = \mathbf{M}_p(w)$ a base datatype. If the corresponding concrete domain of w, $(\Delta_\mathcal{D}, \Phi_\mathcal{D})$, is admissible, then the satisfiability problem for finite predicate conjunctions $\mathscr{C}_w$ of the $\mathsf{sub\text{-}group}(w, \mathcal{G})$ is decidable.*

Proof. Direct consequence of Definition 5.8 and Definition 2.8 on page 28 of [88]: If $(\Delta_\mathcal{D}, \Phi_\mathcal{D})$ is admissible, $\Phi_\mathcal{D}$ is closed under negation and predicate conjunctions over $(\Delta_\mathcal{D}, \Phi_\mathcal{D})$ are decidable. Hence $\forall u \in \mathsf{sub\text{-}group}(w, \mathcal{G}) \setminus \{w\}$, there exists $u' \in \mathsf{sub\text{-}group}(w, \mathcal{G})$, such that $\overline{u}^\mathbf{D} = u'^\mathbf{D}$. Accordingly, negations in $\mathscr{C}_w$ can be eliminated, and $\mathscr{C}_w$ can be equivalently transformed into predicate conjunctions of $(\Delta_\mathcal{D}, \Phi_\mathcal{D})$. $\mathscr{C}_w$ is therefore decidable. □

Note that the use of predicates with multiple arities in a sub-group does not mean that the predicate conjunction of the sub-group is not decidable. For example, if we consider the concrete domain $\mathbb{Q}$ (cf. Example 2.4 on page 41) and check carefully with

its decidability proof presented in [90, p.29-30], it is easy to show that if we replace the ternary predicate

$$E(+) = \{\langle t_1, t_2, t_3\rangle \in V(rational)^3 \mid t_1 = t_2 + t_3\}$$

with the predicate

$$E(+') = \{\langle t_1, \ldots, t_n\rangle \in V(rational)^n \mid t_1 = t_2 + \ldots + t_n\},$$

the predicate conjunction is still decidable. This is because the original proof is based on a reduction to mixed integer programming (MIP), where a *mixed integer programming problem* has the form $Ax = b$, where A is an $m \times n$-matrix of rational numbers, x is an n-vector of variables (there is no restriction that n must be 3), each of them being an integer variable or a rational variable, and b is an m-vector of rational numbers (see, e.g. [132]). In the original proof, the construct $+(x_1, x_2, x_3)$ is transformed into an equation $-x_1 + x_2 + x_3 = 0$. This can be easily extended to the $+'$ predicate, where the construct $+'(x_1, \ldots, x_n)$ can be transformed into an equation $-x_1 + x_2 + \ldots + x_n = 0$. In general, the computability of a sub-group depends on the decidability of the satisfiability of its predicate conjunctions, instead of the existence of a corresponding concrete domain. Lemma 5.9 simply shows a way to reuse existing results.

We end this section by elaborating the conditions that computable datatype groups require.

Definition 5.10. (Conforming Datatype Group) A datatype group $\mathcal{G}$ is *conforming* iff

1. for any $u \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}$ with $a(\mathbf{M}_p(u)) = n \geq 2$: $\mathsf{dom}(u) = (\underbrace{w, \ldots, w}_{n \text{ times}})$ for some $w \in \mathbf{D}_{\mathcal{G}}$, and
2. for any $u \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}$: there exist $u' \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}$ such that $u'^{\mathbf{D}} = \overline{u}^{\mathbf{D}}$, and
3. the satisfiability problems for finite predicate conjunctions of each sub-group of $\mathcal{G}$ is decidable, and
4. for each datatype $u_i \in \mathbf{D}_{\mathcal{G}}$, there exists $w_i \in \Phi_{\mathcal{G}}$, s.t. $\mathbf{M}_p(w_i) = \neq_{u_i}$ where $\neq_{u_i}$ is the binary inequality predicate for $\mathbf{M}_p(u_i)$. $\diamond$

In Definition 5.10, Condition 1 ensures that $\Phi_{\mathcal{G}}$ can be completely divided into sub-groups. Condition 2 and 3 ensure that all the sub-groups are computable, and Condition 4 ensures that number restrictions can be handled (cf. Section 5.1.3).

Example 5.6 A Conforming Datatype Group

$\mathcal{G}_1$ (presented in Example 5.3) is not conforming because it does not satisfy Condition 2 and 4 of Definition 5.10. To make it conforming, we should extend $\mathbf{M}_{p_1}$ as follows:

$$\begin{aligned}\mathbf{M}_{p_1} \;=\; \{&\langle \text{xsd:string}, string\rangle, \langle \text{owlx:stringEquality}, =^{str}\rangle,\\ &\langle \text{owlx:stringInequality}, \neq^{str}\rangle, \langle \text{xsd:integer}, integer\rangle,\\ &\langle \text{owlx:integerEquality}, =^{int}\rangle, \langle \text{owlx:integerInequality}, \neq^{int}\rangle,\\ &\langle \text{owlx:integerGreaterThanx=n}, >^{int}_{[n]}\rangle,\\ &\langle \text{owlx:integerLessThanOrEqualTox=n}, \leq^{int}_{[n]}\rangle\}.\end{aligned}$$

It is possible to devise a polynomial time algorithm to decide satisfiability predicate conjunction over sub-group$(\text{xsd:integer}, \mathcal{G}_1)$ ([88, Sec. 2.4.1]) and sub-group(xsd: string, $\mathcal{G}_1$). ♢

Theorem 5.11. *If $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_{\mathcal{G}}, \mathsf{dom})$ is a conforming datatype group, then the satisfiability problem for finite predicate conjunctions of $\mathcal{G}$ is decidable.*

Proof. Let the predicate conjunction be $\mathscr{C} = \mathscr{C}_{w_1} \wedge \ldots \wedge \mathscr{C}_{w_k} \wedge \mathscr{C}_U$,where $\mathbf{D}_{\mathcal{G}} = \{w_1, \ldots, w_k\}$ and $\mathscr{C}_{w_i}$ is the predicate conjunction for sub-group$(w_i, \mathcal{G})$ and $\mathscr{C}_U$ the sub-conjunction of $\mathscr{C}$ where only unsupported predicate appear.

According to Definition 5.10, the set of $\mathscr{C}_S = \mathscr{C}_{w_1} \wedge \ldots \wedge \mathscr{C}_{w_k}$ is decidable. According to Definition 5.4, $\mathscr{C}_U$ is *unsatisfiable* iff there exist $u(v_1, \ldots, v_n)$ and $\overline{u}(v_1, \ldots, v_n)$ for some $u \notin \Phi_{\mathcal{G}}$ appearing in $\mathscr{C}_U$, which is clear decidable. $\mathscr{C}$ is *satisfiable* iff both $\mathscr{C}_S$ and $\mathscr{C}_U$ are *satisfiable*.

5.1.2 Datatype Expressions

In the last section, we have shown that datatype groups provide an OWL-compatible unified formalism for datatypes and predicates. In this section, we further describe how to construct datatype expressions in the unified formalism, so as to represent customised datatypes and predicates.

Definition 5.12. ($\mathcal{G}$-Datatype Expressions) Let $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_{\mathcal{G}}, \mathsf{dom})$ be a datatype group, the set of *$\mathcal{G}$-datatype expressions* (or simply datatype expressions), abbreviated $\mathbf{Dexp}(\mathcal{G})$, is inductively defined as follows:

Table 5.1: Semantics of datatype expressions

Abstract Syntax	DL Syntax	Semantics
$\texttt{oneOf}(l_1, \ldots, l_n)$	$\{l_1, \ldots, l_n\}$	$\{l_1^{\mathbf{D}}\} \cup \ldots \cup \{l_n^{\mathbf{D}}\}$
$\texttt{domain}(u_1, \ldots, u_n)$	$[u_1, \ldots, u_n]$	$u_1^{\mathbf{D}} \times \ldots \times u_n^{\mathbf{D}}$
$\texttt{and}(P, Q)$	$P \wedge Q$	$P^{\mathbf{D}} \cap Q^{\mathbf{D}}$
$\texttt{or}(P, Q)$	$P \vee Q$	$P^{\mathbf{D}} \cup Q^{\mathbf{D}}$

1. for any predicate URIrefs $u : u, \overline{u} \in \mathbf{Dexp}(\mathcal{G})$;

2. let $l_1, \ldots, l_n$ be typed literals, $\{l_1, \ldots, l_n\} \in \mathbf{Dexp}(\mathcal{G})$ with arity 1, where $\{\}$ is called the `oneOf` constructor;

3. let u_i' be either rdfs:Literal, w, u_i or $\overline{u_i}$ (for all $1 \leq i \leq n$), where $w \in \mathbf{D}_{\mathcal{G}}$, $u_i \in$ sub-group$(w, \mathcal{G}) \setminus \{w\}$ and $a(\mathbf{M}_p(u_i)) = 1$, $[u_1', \ldots, u_n'] \in \mathbf{Dexp}(\mathcal{G})$ with arity n;

4. $\forall P, Q \in \mathbf{Dexp}(\mathcal{G})$, $P \wedge Q, P \vee Q \in \mathbf{Dexp}(\mathcal{G})$ if:

   (a) both P and Q have arity n (in this case, $P \wedge Q, P \vee Q$ have arity n); or

   (b) one of them has arity n, the other has minimal arity n_{min} and $n \geq n_{min}$ (in this case $P \wedge Q, P \vee Q$ have arity n); or

   (c) one of them has minimum arity n_{min}, the other has minimum arity n'_{min} and $n_{min} \geq n'_{min}$ (in this case $P \wedge Q, P \vee Q$ have minimum arity n_{min}).

   $\wedge, \vee$ are called the `and`,`or` constructors, respectively.

The semantics of datatype expressions of the above form 1 is already defined in Definition 5.4. The abstract syntax and semantics of the rest forms of $\mathcal{G}$-datatype expressions are given in Table 5.1.

Let P be a $\mathcal{G}$-datatype expression. The negation of P is of the form $\neg P$: if the arity of P is n $(n > 0)$, $\neg P$ is interpreted as $(\Delta_{\mathbf{D}})^n \setminus P^{\mathbf{D}}$; if the minimum arity of P is n $(n > 0)$, then $\neg P$ is interpreted as $\bigcup_{m \geq n} (\Delta_{\mathbf{D}})^m \setminus P^{\mathbf{D}}$. $\diamond$

Note that, similar to predicates, a datatype expression either has a fixed arity or has a minimum arity.

The following examples show how to use datatype expressions to represent customised datatypes and predicates.

Example 5.7 How to Use Datatype Expressions to Represent Customised Datatypes and Predicates

Customised Datatypes: $\mathcal{G}$-datatype expressions can be used to represent two kinds of *user-derived XML Schema datatypes*: derivation by restriction and derivation by union.[3]

- The customised XML Schema datatype 'greaterThan20' (derivation by restriction)

```
<simpleType name = "greaterThan20">
  <restriction base = "xsd:integer>
    <minExclusive value = "20"/>
  </restriction>
</simpleType>
```

  can be represented as

$$\mathrm{owlx{:}integerGreaterThanx{=}20}$$

  which is a unary predicate URIref.

- The customised XML Schema datatype 'cameraPrice', by which one might want to express camera prices as a float number between 8.00 and 100000.00, or with one of the strings "low", "medium" or "expensive" (derivation by union and restriction)

```
<simpleType name = "cameraPrice">
  <union>
    <simpleType>
      <restriction base = "xsd:float">
        <minInclusive value = "8.00"/>
        <maxInclusive value = "100000.00"/>
      </restriction>
    </simpleType>
    <simpleType>
      <restriction base = "xsd:string">
        <enumeration value = "low"/>
        <enumeration value = "medium"/>
        <enumeration value = "expensive"/>
      </restriction>
    </simpleType>
  </union>
<simpleType>
```

[3]List-value datatypes do not fit the RDF datatype model.

can be represented by

$$(\text{owlx:floatGreaterThanx=8.00} \wedge \text{owlx:floatLessThanOrEqualTox=100000.00}) \\ \vee\{\text{“}low\text{”}\text{^^xsd:string}, \text{“}medium\text{”}\text{^^xsd:string}, \text{“}expensive\text{”}\text{^^xsd:string}\},$$

which is a disjunction expression with arity 1, where the first disjunct is a conjunction expression and the second disjunct is an `oneOf` expression.[4]

Customised Predicates: $\mathcal{G}$-datatype expressions can be used to represent customised predicates, e.g., those we have seen in Section 1.3.

- The customised predicate ‘sumNoGreaterThan15’, with extension $E(sum\text{-}NoGreaterThan15) = \{\langle i_0, i_1, i_2, i_3\rangle \in V(integer)^4 \mid i_0 = i_1 + i_2 + i_3$ and $\neg(i_0 > 15)\}$ and arity $a(sumNoGreaterThan15) = 4$, can be represented by

  $$\text{owlx:integerAddition} \wedge \\ [\ \overline{\text{owlx:integerGreaterThanx=15}}, \text{xsd:integer}, \text{xsd:integer}, \text{xsd:integer}\ ],$$

  which is a conjunction expression, where the first conjunct is a predicate URIref and the second conjunct is a domain expression. Note that the predicate owlx:integerAddition (i.e., $+^{int}$) has a minimum arity 3, while the arity of the domain expression is 4. According to Definition 5.12, the arity of the conjunction expression is 4.

- The customised predicate ‘multiplyBy1.6’, with extension $E(multiplyBy1.6) = \{\langle b_0, b_1, b_2\rangle \in V(double)^3 \mid b_0 = b_1 * b_2$ and $b_1 = 1.6\}$ and arity $a(multiplyBy 1.6) = 3$, can be represented by

  $$\text{owlx:doubleMultiplication} \wedge \\ [\ \text{xsd:double}, \text{owlx:doubleEqualTox=1.6}, \text{xsd:double}\ ],$$

  which is a conjunction expression with arity 3, where owlx:doubleMultiplication is the predicate URIref for $*^{double}$.

- The customised predicate ‘multiplyBy1.6v2’, which is a variation of ‘multiplyBy1.6’, with extension $E(multiplyBy1.6v2) = \{\langle b_0, b_1\rangle \in V(double)^2 \mid b_0 = 1.6 * b_1\}$ and arity $a(multiplyBy1.6v2) = 2$, can be represented by

  $$\text{owlx:doubleMultiplicationx=1.6} \wedge [\ \text{xsd:double}, \text{xsd:double}\ ],$$

  which is a conjunction expression with arity 2, where owlx:doubleMultiplicationx=y is the predicate URIref for $*^{double}_{[y]}$, where $E(*^{double}_{[y]}) = \{\langle d_0, \ldots, d_n\rangle \mid$

[4] Note that the `oneOf` constructor is redundant (but sometimes convenient) and can be simulated by the `or` constructor and unary equality predicates.

$d_0 = L2V(double)(\text{“}y\text{”}) * d_1 * \ldots * d_n\}$. By '$*_{[y]}^{double}$', we mean there exist a predicate $*_{[y]}^{double}$ for each double number $L2V(double)(\text{“}y\text{”})$.[5] ♢

Let us now consider a more expressive reasoning mechanism than predicate conjunctions that we introduced in Definition 5.6 (page 106).

Definition 5.13. (Datatype Expression Conjunction) Let $\mathcal{G}$ be a conforming datatype group, **V** a set of variables, we consider $\mathcal{G}$-datatype expression conjunctions of the form

$$\mathscr{C}_E = \bigwedge_{j=1}^{k} P_j(v_1^{(j)}, \ldots, v_{n_j}^{(j)}), \tag{5.2}$$

where P_j is a (possibly negated) $\mathcal{G}$-datatype expression, with its arity equal to n_j or its minimum arity less than or equal to n_j, and the $v_i^{(j)}$ are variables from **V**. A $\mathcal{G}$-datatype expression conjunction $\mathscr{C}_E$ is called *satisfiable* iff there exist an interpretation $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ of $\mathcal{G}$ and a function δ mapping the variables in $\mathscr{C}_E$ to data values in $\Delta_{\mathbf{D}}$ s.t. $\langle \delta(v_1^{(j)}), \ldots, \delta(v_{n_j}^{(j)}) \rangle \in P_j^{\mathbf{D}}$ for $1 \leq j \leq k$. Such a function δ is called a *solution* for $\mathscr{C}_E$ (w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$). ♢

Lemma 5.14. *Let $\mathcal{G}$ be a conforming datatype group; $\mathcal{G}$-datatype expression conjunctions of the form (5.2) are decidable.*

Proof. It is trivial to reduce the satisfiability problem for $\mathcal{G}$-datatype expression conjunctions to the satisfiability problem for predicate conjunctions over $\mathcal{G}$:

1. Due to Condition 2 of a conforming datatype group (cf. Definition 5.10 on page 109), we can eliminate relativised negations in a `domain` expression. Hence, the `domain` constructor simply introduces predicate conjunctions because $\texttt{domain}(u_1, \ldots, u_n)\ (v_1, \ldots, v_n) \equiv u_1(v_1) \wedge \ldots \wedge u_n(v_n)$. Similarly, its negation introduces disjunctions, $\neg\texttt{domain}(u_1, \ldots, u_n)(v_1, \ldots, v_n) \equiv \neg u_1(v_1) \vee \ldots \vee \neg u_n(v_n)$, where

$$\neg u_i(v_i) \equiv \begin{cases} \overline{u_i}(v_i) \vee \overline{\mathsf{dom}(u_i)}(v_i) & \text{if } u_i \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}, \\ \overline{u_i}(v_i) & \text{otherwise,} \end{cases}$$

   according to Definition 5.5.

[5] See the discussion about parameterised predicates on page 103.

2. The `and` and `or` constructors simply introduce disjunctions of predicate conjunctions of $\mathcal{G}$. According to Lemma 5.11, the satisfiability problem of predicate conjunctions of $\mathcal{G}$ is decidable; therefore, a $\mathcal{G}$-datatype expression conjunction is satisfiable iff one of its disjuncts is satisfiable. □

We end this section by a brief explanation of the several kinds of conjunctions we have seen so far in this Chapter. From the viewpoint of users points of view, $\mathcal{G}$-datatype expressions allows users to use conjunctions to construct customised datatypes and predicates (see Definition 5.12). From the viewpoint of datatype reasoners, they are used to solve the satisfiability problems of predicate conjunctions (see Definition 5.6) and datatype expression conjunctions (see Definition 5.13).

5.1.3 Relations with Existing Formalisms

In this section, we briefly discuss the relations between our formalism and existing ontology-related formalisms.

Relation with Concrete Domains

The concrete domain approach provides a rigorous treatment of datatype predicates within concrete domains. The basic reasoning task is to compute the satisfiability of finite predicate conjunctions, which is required to be decidable in admissible concrete domains. In our formalism, a conforming datatype group requires that the satisfiability of finite predicate conjunctions of all its sub-groups should be decidable (cf. Condition 3 of Definition 5.10 on page 109). Sub-groups of a datatype group can, but not necessarily, correspond to concrete domains (cf. Definition 5.8 on page 108).

A concrete domain is similar to a datatype in that its domain corresponds to the value space of a datatype. The concrete domain approach, however, does not consider the lexical space and lexical-to-value mapping of a datatype. Accordingly, it does not provide a formal way to represent data values of a datatype (with the help of the lexical-to-value mapping of the datatype). This could introduce confusions in the ABox when data values of different concrete domains (e.g., $boolean$ and $integer$) share the same lexical forms (e.g., "1" and "0"). In addition, this could also affect definitions of some predicate extensions. For example, the extension of $>^{int}_{[n]}$ is defined as follows in the concrete domain approach:

$$E(>^{int}_{[n]}) = \{i \in E(integer) \mid i > n\}.$$

Please note that the "n" in the predicate name $>^{int}_{[n]}$ is a string, while the n in the semantic condition '$i > n$' is an integer. So strictly speaking they are *not* the same thing. In the datatype group approach, on the contrary, the extension of $>^{int}_{[n]}$ is defined as follows:

$$E(>^{int}_{[n]}) = \{i \in E(integer) \mid i > L2V(integer)(\text{“}n\text{”})\},$$

where the lexical-to-value mapping of $integer$, viz. $L2V(integer)$, maps "n", the lexical form, to its corresponding integer value.

The concrete domain approach is rather compact, or simplified, and thus not very user-friendly. Our formalism overcomes various limitations of concrete domains discussed in Section 2.3.3:

1. It supports predicates with non-fixed arities (cf. Definition 5.1 on page 102).
2. It provides relativised negations for supported predicate URIrefs in predicate conjunctions (5.1) of a datatype group.
3. Most importantly, it provides datatype expressions (cf. Definition 5.12 on page 110) to represent customised datatypes and customised predicates, which are very useful in SW and ontology applications.

Furthermore, our formalism provides the use of unsupported predicate URI references, (cf. Definition 5.3 on page 103). This again makes the formalism more user-friendly, as datatype reasoners supporting different datatype groups will not simply reject the predicate URI references that are not in their predicate maps.

Relation with OWL Datatyping

OWL datatyping, on the other hand, is focused on datatypes instead of predicates. It uses the RDF(S) specification of datatypes and data values, but it is different from RDF(S) in that datatypes are not classes in its concept language and the datatype domain (w.r.t. a datatype map) is disjoint with the object domain (of an interpretation). It uses datatype maps to distinguish supported datatype URIrefs from unsupported ones, and it also provides enumerated datatypes. The very serious limitation of OWL datatyping is that it does not support customised datatypes and predicates.

Our formalism is compatible with OWL datatyping. Furthermore, it overcome all the limitations of OWL datatyping discussed in the 'Limitations of OWL Datatyping' section on page 77:

- It extends OWL datatyping to support predicates.
- Most importantly, it introduces datatype expressions to provide the use of customised datatypes and predicates.
- It provides (relativised) negations for predicate URIrefs in a datatype group (cf. Definition 5.4 on page 104).
- The datatype domain in an interpretation of a datatype group is a superset of (instead of equivalent to) the value spaces of base datatypes and plain literals, so typed literals with unsupported predicates are interpreted more intuitively (cf. Definition 5.5 on page 104).

5.2 Integrating DLs with Datatype Groups

We shall now present our scheme for integrating an arbitrary datatype group into so called $\mathcal{G}$-combinable Description Logics, a family of Description Logics that satisfy some weak conditions. We will show that members of the family of integrated DLs are decidable.

5.2.1 $\mathcal{G}$-combinable Description Logics

Formally, a $\mathcal{G}$-combinable Description Logics is defined as follows.

Definition 5.15. ($\mathcal{G}$-combinable Description Logics) Given a Description Logic $\mathcal{L}$, $\mathcal{L}$ is called a *$\mathcal{G}$-combinable Description Logic* if (i) $\mathcal{L}$ provides the conjunction ($\sqcap$) and bottom ($\bot$) constructors and (ii) $\mathcal{L}$-concept satisfiability w.r.t. TBoxes and RBoxes is decidable. $\mathcal{L}(\mathcal{G})$ is called a *$\mathcal{G}$-combined Description Logic*. $\diamond$

Many well-known DLs, such as $\mathcal{SHIQ}$, $\mathcal{SHOQ}$ and their sub-languages (e.g., $\mathcal{ALC}$), are $\mathcal{G}$-combinable Description Logics. We can integrate an arbitrary datatype group $\mathcal{G}$ into a $\mathcal{G}$-combinable Description Logic $\mathcal{L}$. The result of this integration is called $\mathcal{L}(\mathcal{G})$. For $\mathcal{L}(\mathcal{G})$, an interpretation is of the form $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}, \Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$, where $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is an interpretation of the object domain, $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ is an interpretation of the datatype group $\mathcal{G}$ and we require $\Delta^{\mathcal{I}}$ and $\Delta_{\mathbf{D}}$ are disjoint with each other. In addition to the language constructors of $\mathcal{L}$, $\mathcal{L}(\mathcal{G})$ allows datatype group-based concept descriptions shown in Table 5.2.

Constructor	DL Syntax	Semantics
expressive predicate exists restriction	$\exists T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \exists t_1, \ldots, t_n.\langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\}$
expressive predicate value restriction)	$\forall T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \forall t_1, \ldots, t_n.\langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \rightarrow \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\}$
expressive predicate atleast restriction	$\geqslant m T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle t_1, \ldots, t_n\rangle \mid \langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\} \geq m\}$
expressive predicate atmost restriction	$\leqslant m T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle t_1, \ldots, t_n\rangle \mid \langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\} \leq m\}$

Table 5.2: Datatype group-based concept descriptions

Definition 5.16. (**$\mathcal{L}(\mathcal{G})$-concepts**) Let $\mathcal{G}$ be a datatype group, $\mathcal{L}$ a $\mathcal{G}$-combinable Description Logic, $\mathsf{CN} \in \mathbf{C}$ be an atomic concept name, $T_1, \ldots, T_n \in \mathbf{Rdsc_D}(\mathcal{L})$ concrete roles (where $T_i \not\sqsubseteq^* T_j, T_j \not\sqsubseteq^* T_i$ for all $1 \leq i < j \leq n$,[6] if $\mathcal{L}$ provides role inclusion axioms) and E a $\mathcal{G}$-datatype expression. Valid $\mathcal{L}(\mathcal{G})$-concepts include valid $\mathcal{L}$-concepts as well as the following concept descriptions defined by the abstract syntax:

$$
\begin{array}{rcl}
C & ::= & \exists T_1, \ldots, T_n.E \text{ (expressive predicate exists restriction)} \mid \\
 & & \forall T_1, \ldots, T_n.E \text{ (expressive predicate value restriction)} \mid \\
 & & \geqslant T_1, \ldots, T_n E. \text{ (expressive predicate atleast restriction)} \mid \\
 & & \leqslant T_1, \ldots, T_n E. \text{ (expressive predicate atmost restriction)}
\end{array}
$$

The semantics of the above $\mathcal{L}(\mathcal{G})$-concepts is given in Table 5.2 ◇

There are some remarks about $\mathcal{L}(\mathcal{G})$-concepts: (i) $T_1, \ldots, T_n$ are concrete roles, and feature chains (cf. Section 2.3.1) are not provided by $\mathcal{L}(\mathcal{G})$. (ii) $T_1, \ldots, T_n$ should not be sub-roles (cf. Definition 2.14 on page 47) of each other. Otherwise, the interpretations datatype group-based concepts would be less intuitive, since the value of the same concrete super-role serves as more than one argument of a datatype expression. (iii) The datatype group-based concept constructors are different from the object counterparts (i.e., the exists, value and qualified number restrictions) shown in Table 2.2 on page 46.

5.2.2 Datatype Queries

To decide $\mathcal{L}(\mathcal{G})$-concept satisfiability and subsumption problem w.r.t. TBoxes and RBoxes, a DL reasoner can use a datatype reasoner to answer so called $\mathcal{G}$-datatype

[6] $\sqsubseteq^*$ is the transitive reflexive closure of $\sqsubseteq$.

queries.

Definition 5.17. ($\mathcal{G}$-Datatype Query) For a datatype group $\mathcal{G}$, a *$\mathcal{G}$-datatype query* is of the form

$$\mathcal{Q} := \bigvee_{j=1}^{k} \mathscr{C}_{E_j} \wedge \bigwedge_{j_1=1}^{k_1} \neq (\vec{v}_{(j_1,1)}; \vec{v}_{(j_1,2)}) \wedge \bigvee_{j_2=1}^{k_2} = (\vec{v}_{(j_2,1)}; \ldots; \vec{v}_{(j_2,m_{j_2})}), \tag{5.3}$$

where $\mathscr{C}_{E_j}$ is a (possibly negated) $\mathcal{G}$-datatype expression conjunction, $\vec{v}_{(s)}$ are tuples of variables appearing in $\mathscr{C}_{E_1}, \ldots, \mathscr{C}_{E_k}$, and $\neq$ and $=$ are called the *value inequality predicate* and *value equality predicate*, respectively. A $\mathcal{G}$-datatype query is *satisfiable* iff there exists an interpretation $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ of $\mathcal{G}$ and a function δ mapping the variables in $\mathscr{C}_{E_1}, \ldots, \mathscr{C}_{E_k}$ to data values in $\Delta_{\mathbf{D}}$ s.t.

- δ is a solution for one of $\mathscr{C}_{E_1}, \ldots, \mathscr{C}_{E_k}$ w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ and,
- $\delta(\vec{v}_{(j_1,1)}) \neq \delta(\vec{v}_{(j_1,2)})$ for all $1 \leq j_1 \leq k_1$,[7]
- there exist some j_2 ($1 \leq j_2 \leq k_2$) s.t. $\delta(\vec{v}_{(j_2,1)}) = \cdots = \delta(\vec{v}_{(j_2,m_{j_2})})$.

Such a function δ is called a *solution* for $\mathcal{Q}$ w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$. ◇

Lemma 5.18. *For $\mathcal{G}$ a conforming datatype group, $\mathcal{G}$-datatype queries of the form (5.3) are decidable.*

Proof. Since the satisfiability problem of (possibly negated) $\mathcal{G}$-datatype expression conjunctions is decidable (cf. Lemma 5.14 on page 114), we only need to show how to handle the extra constraints introduced by the *value inequality predicate* and *value equality predicate*. We can transform the tuple-level equality and inequality constraints into $\mathscr{V}$, a disjunction of conjunctions of variable-level constraints, which are of the forms $= (v_i, v_j)$ or $\neq (v_{i'}, v_{j'})$. For each satisfiable $\mathcal{G}$-datatype expression conjunction $\mathscr{C}_{E_j}$, we can further extend $\mathscr{C}_{E_j}$ to $\mathscr{C}'_{E_j}$ by adding new conjuncts $=_u (v_i, v_j)$ and/or $\neq_u (v_{i'}, v_{j'})$ into $\mathscr{C}_E$. $\mathcal{Q}$ is *unsatisfiable* if all $\mathscr{C}'_{E_j}$ are *unsatisfiable*; otherwise, $\mathcal{Q}$ is *satisfiable*. □

5.2.3 Decidability

Now we show that $\mathcal{L}(\mathcal{G})$ is decidable. The proof is inspired by the proof (Lutz [88, p.32-33]) of the decidability of $\mathcal{ALCF}(\mathcal{D})$-concept satisfiability w.r.t. to general TBoxes,

[7] Note that, if $\vec{v} = \langle v_1, \ldots, v_n \rangle$, $\delta(\vec{v})$ is an abbreviation for $\langle \delta(v_1), \ldots, \delta(v_n) \rangle$.

where $\mathcal{ALCF}(\mathcal{D})$ is obtained from $\mathcal{ALC}(\mathcal{D})$ by restricting the concrete domain constructor to concrete features in place of feature chains. The $\mathcal{ALCF}(\mathcal{D})$ DL can be seen a restricted form of $\mathcal{ALC}(\mathcal{G})$, where, roughly speaking, all concrete roles are functional, $\mathcal{D}$ is the only base datatype in $\mathcal{G}$, unsupported predicates are disallowed and the only kind of datatype expression are predicates of $\mathcal{D}$.

We will show the decidability of $\mathcal{L}(\mathcal{G})$-concept satisfiability w.r.t. TBoxes and RBoxes by reducing it to the $\mathcal{L}$-concept satisfiability w.r.t. TBoxes and RBoxes. The basic idea behind the reduction is that we can replace each datatype group-based concept C in $\mathcal{T}$ with a new atomic primitive concept A_C in $\mathcal{T}'$. We then compute the satisfiability problem for all possible conjunctions of datatype group-based concepts in $\mathcal{T}$ (of which there are only a finite number), and in case a conjunction $C_1 \sqcap \ldots \sqcap C_n$ is unsatisfiable, we add an axiom $A_{C_1} \sqcap \ldots \sqcap A_{C_n} \sqsubseteq \bot$ to $\mathcal{T}'$.

For example, datatype group-based concepts $\exists T.\ >_1$ and $\forall T.\ \leq_0$ occurring in $\mathcal{T}$ would be replaced with $A_{\exists T.>_1}$ and $A_{\forall T.\leq_0}$ in $\mathcal{T}'$, and $A_{\exists T.>_1} \sqcap A_{\forall T.\leq_0} \sqsubseteq \bot$ would be added to $\mathcal{T}'$ because $\exists T.\ >_1 \sqcap \forall T.\ \leq_0$ is *unsatisfiable* (i.e., there is no solution for the predicate conjunction $>_1 (v) \wedge \leq_0 (v)$).

Theorem 5.19. *Given a conforming datatype group $\mathcal{G}$, the $\mathcal{L}(\mathcal{G})$-concept satisfiability problem w.r.t. TBoxes and RBoxes is decidable.*

Proof: We prove the theorem by reducing $\mathcal{L}(\mathcal{G})$-concept satisfiability w.r.t. TBoxes and RBoxes to the $\mathcal{L}$-concept satisfiability w.r.t. TBoxes and RBoxes. Let D be an $\mathcal{L}(\mathcal{G})$-concept for satisfiability checking, $\mathsf{cl}_{\mathcal{T}}(D)$ the set of all the sub-concepts of concepts in $\{D\} \cup \{D_1, D_2 \mid D_1 \sqsubseteq D_2 \in \mathcal{T}$ or $D_1 = D_2 \in \mathcal{T}\}$, and $\{C_1, \ldots, C_k\} \subseteq \mathsf{cl}_{\mathcal{T}}(D)$ the set of all the datatype group-based concepts in $\mathsf{cl}_{\mathcal{T}}(D)$, i.e., each C_i ($1 \leq i \leq k$) is of one of the four forms shown in the 'DL Syntax' column of Table 5.2 on page 118. Let T be a concrete role name. We assume that all the functional concrete role axioms in $\mathcal{R}$ of the form $\mathsf{Func}(T)$ are encoded into concept inclusion axioms of the form $\top \sqsubseteq \leqslant 1 T.\top_{\mathrm{D}}$ in $\mathcal{T}$.

We define a mapping π that maps datatype group-based concept conjunctions of the form $S = B_1 \sqcap \ldots \sqcap B_h$, where $\{B_1, \ldots, B_h\} \subseteq \{C_1, \ldots, C_k\}$, to a corresponding $\mathcal{G}$-datatype query (cf. equation (5.3) on page 119) $\pi(S)$, according to the semantics of the datatype group-based concepts given in Table 5.2:

(Step 1) For each B_j of the form $\exists T_1, \ldots, T_{n_j}.E$, $\pi(S)$ contains a conjunct $E(v_{j1}^{T_1}, \ldots, v_{jn_j}^{T_{n_j}})$, where each $v_{ji}^{T_i}$ ($1 \leq i \leq n_j$) is a variable, with the corresponding concrete role name T_i as its superscript.

(Step 2) For each B_j of the form $\geqslant m T_1, \ldots, T_{n_j}.E$, $\pi(S)$ contains a conjunct

$$\bigwedge_{a=1}^{m} E(v_{j_a 1}^{T_1}, \ldots, v_{j_a n_j}^{T_{n_j}}) \wedge \bigwedge_{1 \leq a < b \leq m} \neq (v_{j_a 1}^{T_1}, \ldots, v_{j_a n_j}^{T_{n_j}}; v_{j_b 1}^{T_1}, \ldots, v_{j_b n_j}^{T_{n_j}})$$

where the inequality constraints are used to make sure all the tuples $\langle v_{j_1 1}^{T_1}, \ldots, v_{j_1 n_j}^{T_{n_j}} \rangle, \ldots, \langle v_{j_m 1}^{T_1}, \ldots, v_{j_m n_j}^{T_{n_j}} \rangle$ are different from each other. We will not introduce any more new variables (with superscriptions) in the following steps.

(Step 3) For each B_j of the form $\forall T_1, \ldots, T_{n_j}.E$, let A_j be the set of all tuples $\vec{v}$ of variables that were introduced in (Step 1) and (Step 2), of the form $\langle v_1^{T'_1}, \ldots, v_{n_j}^{T'_{n_j}} \rangle$, where the superscript of each $v_1^{T'_1}$ $(1 \leq i \leq n_j)$ matches the corresponding concrete role name T_i in $\forall T_1, \ldots, T_{n_j}.E$. A variable $v^{T'}$ *matches* a concrete role T if $T' \sqsubseteq^{*} T$. Then $\pi(S)$ contains a conjunct

$$\bigwedge_{\forall \vec{v} \in A_j} E(\vec{v}).$$

(Step 4) For each B_j of the form $\leqslant m T_1, \ldots, T_{n_j}.E$, similarly to (Step 3), we can define a set A_j for B_j. Let $\mid A_j \mid = m'$. If $m' \leq m$, then B_j will not introduce any new datatype constraints. Otherwise, let $P(A, x)$ be the function that maps a set A to the set of all the partitions of A with size x; i.e., for each partition $Q = \{q_1, \ldots, q_x\} \in P(A, x)$, $q_1, \ldots, q_n$ are non-empty sets, $q_1 \cap \ldots \cap q_x = \emptyset$ and $A = q_1 \cup \ldots \cup q_x$. Then $\pi(S)$ contains a conjunct

$$\bigvee_{Q \in P(A_j, m)} \bigwedge_{q \in Q} \bigwedge_{\vec{v_1}, \vec{v_2} \in q} E(\vec{v_1}) \wedge E(\vec{v_2}) \rightarrow = (\vec{v_1}; \vec{v_2}),$$

we can apply the "$x \Rightarrow y \equiv \neg x \vee y$" equivalence and DeMorgan's law to this conjunct to give

$$\bigvee_{Q \in P(A_j, m)} \bigwedge_{q \in Q} \bigwedge_{\vec{v_1}, \vec{v_2} \in q} \neg E(\vec{v_1}) \vee \neg E(\vec{v_2}) \vee \ = (\vec{v_1}; \vec{v_2}).$$

Since the satisfiability problem for a datatype query is decidable, for each possible $S = B_1 \sqcap \ldots \sqcap B_h$, where $\{B_1, \ldots, B_h\} \subseteq \{C_1, \ldots, C_k\}$, we can decide if $\pi(S)$ is *satisfiable* or not.

Now we can reduce the $\mathcal{L}(\mathcal{G})$-concept satisfiability problem to the $\mathcal{L}$-concept satisfiability problem, by introducing some new atomic primitive concepts (to represent C_i, for each $1 \leq i \leq k$) and some concept inclusion axioms about these atomic primitive concepts (to capture all the possible contradictions caused by S) as follows:

1) We create an atomic primitive concept A_{C_i} for each $C_i \in \{C_1, \ldots, C_k\}$, and transform $\mathcal{T}$ and D into $\mathcal{T}'$ and D' by replacing all C_i with A_{C_i} in $\mathcal{T}$ and D. We transform $\mathcal{R}$ into $\mathcal{R}'$ by removing all the concrete role inclusion axioms.

2) For each $S = B_1 \sqcap \ldots \sqcap B_h$, where $\{B_1, \ldots, B_h\} \subseteq \{C_1, \ldots, C_k\}$:

   - If $\pi(S)$ is *satisfiable*, we do nothing.
   - If $\pi(S)$ is *unsatisfiable*, we add the following concept inclusion axiom into $\mathcal{T}'$:

$$A_{B_1} \sqcap \ldots \sqcap A_{B_h} \sqsubseteq \bot,$$

Claim: (i) For any $S = B_1 \sqcap \ldots \sqcap B_h$, where $\{B_1, \ldots, B_h\} \subseteq \{C_1, \ldots, C_k\}$, S is *satisfiable* iff $\pi(S)$ is *satisfiable*. (ii) All the possible contradictions caused by possible datatype group-based sub-concept conjunctions in $\mathsf{cl}_{\mathcal{T}}(D)$ have been encoded in the TBox $\mathcal{T}'$. (iii) D is *satisfiable* w.r.t. $\mathcal{T}$ and $\mathcal{R}$ iff D' is *satisfiable* w.r.t. $\mathcal{T}'$ and $\mathcal{R}'$.

Claim (i) is true because the mappings in (Step1) - (Step4) exactly generate the needed $\mathcal{G}$-datatype queries $\pi(S)$ according to the semantics of datatype group-based concepts.

- (Step1): For each B_j of the form $\exists T_1, \ldots, T_{n_j}.E$, $\pi(S)$ contains a conjunct $E(v_{j1}^{T_1}, \ldots, v_{jn_j}^{T_{n_j}})$. If $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ is an interpretation of $\mathcal{G}$ and δ is a solution of $E(v_{j1}^{T_1}, \ldots, v_{jn_j}^{T_{n_j}})$ w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$, we have $(\delta(v_{j1}^{T_1}), \ldots, \delta(v_{jn}^{T_{n_j}})) \in E^{\mathbf{D}}$. Furthermore, the concrete role names T_i are used in superscripts of the corresponding variables, so as to assure that further constraints from datatype expression value and atmost restrictions can be properly added to these variables.

- (Step2): For each B_j of the form $\geqslant m T_1, \ldots, T_{n_j}.E$, $\pi(S)$ contains a conjunct $\bigwedge_{a=1}^{m} E(v_{j_a1}^{T_1}, \ldots, v_{j_an_j}^{T_{n_j}}) \wedge \bigwedge_{1 \leq a < b \leq m} \neq (v_{j_a1}^{T_1}, \ldots, v_{j_an_j}^{T_{n_j}}; v_{j_b1}^{T_1}, \ldots, v_{j_bn_j}^{T_{n_j}})$. If $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ is an interpretation of $\mathcal{G}$ and δ is a solution w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ of this conjunct, we have $(\delta(v_{j_a1}^{T_1}), \ldots, \delta(v_{j_an}^{T_{n_j}})) \in E^{\mathbf{D}}$ for all $1 \leq a \leq m$ and $(\delta(v_{j_a1}^{T_1}), \ldots, \delta(v_{j_an}^{T_{n_j}})) \neq (\delta(v_{j_b1}^{T_1}), \ldots, \delta(v_{j_bn}^{T_{n_j}}))$ for all $1 \leq a < b \leq m$; viz. there are at least m tuples of data values that satisfy the datatype expression E. The purpose of using superscripts in variables is the same as (Step1).

- (Step3): For each B_j of the form $\forall T_1, \ldots, T_{n_j}.E$, $\pi(S)$ contains a conjunct $\bigwedge_{\forall \vec{v} \in A_j} E(\vec{v})$. Since in (Step1) and (Step2) we have generated all the needed variables, the set A_j includes all the tuples of variables, the superscripts of which match $T_1, \ldots, T_{n_j}$. If $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ is an interpretation of $\mathcal{G}$ and δ is a solution w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ of the above conjunct, we have $\delta(\vec{v}) \in E^{\mathbf{D}}$, for all $\vec{v} \in A_j$.

- (Step4): For each B_j of the form $\leqslant m T_1, \ldots, T_{n_j}.E$, $\pi(S)$ contains a conjunct

$$\bigvee_{Q \in P(A_j, m)} \bigwedge_{q \in Q} \bigwedge_{\vec{v_1}, \vec{v_2} \in q} E(\vec{v_1}) \wedge E(\vec{v_2}) \rightarrow = (\vec{v_1}; \vec{v_2}),$$

  if $m <\mid A_j \mid$. The set A_j is constructed as that in (Step3), and $P(A_j, m)$ is the set of all the partitions of A_j with size m. If $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ is an interpretation of $\mathcal{G}$ and δ is a solution w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ of this conjunct, there exists a partition Q, s.t. for all $q_i \in Q$ ($1 \leq i \leq m$), any pairs of variable tuples $\vec{v_1}, \vec{v_2}$ must satisfy that if both $\delta(\vec{v_1}) \in E^{\mathbf{D}}$ and $\delta(\vec{v_2}) \in E^{\mathbf{D}}$ are true, then $\delta(\vec{v_1}) = \delta(\vec{v_2})$. In other words, there are at most m different tuples of data values that are linked through the concrete roles $T_1, \ldots, T_{n_j}$ and satisfy E.

For claim (ii). Firstly, due to the 1), it is obvious that D' is an $\mathcal{L}$-concept and $\mathcal{T}'$ contains no datatype group-based concepts, and there are no concrete roles in $\mathcal{R}'$. Secondly, due to 2), claim (i) and that $\mathcal{G}$-datatype queries are decidable, for any possible datatype group-based concept conjunction $S = B_1 \sqcap \ldots \sqcap B_h$ and if $\pi(S)$ is *unsatisfiable*, there is an axiom $A_{B_1} \sqcap \ldots \sqcap A_{B_h} \sqsubseteq \bot$ in $\mathcal{T}'$. Therefore, all the possible contradictions caused by possible datatype group-based sub-concept conjunctions in $\mathsf{cl}_{\mathcal{T}}(D)$ have been encoded in the TBox $\mathcal{T}'$.

For claim (iii). If D is *satisfiable* w.r.t. $\mathcal{T}$ and $\mathcal{R}$, then there is a model $\mathcal{I}$, s.t. $\mathcal{I} \models D, \mathcal{I} \models \mathcal{T}$ and $\mathcal{I} \models \mathcal{R}$. We show how to construct a model $\mathcal{I}'$ of D' w.r.t. $\mathcal{T}'$ and $\mathcal{R}'$ from $\mathcal{I}$. $\mathcal{I}'$ will be identical to $\mathcal{I}$ in every respect except for concrete roles (there are no concrete roles in $\mathcal{I}'$) and the atomic primitive concepts A_{C_i} for each $C_i \in \{C_1, \ldots, C_k\}$ (there are no A_{C_i} in $\mathcal{I}$). So we only need to construct $A_{C_i}^{\mathcal{I}'}$: $A_{C_i}^{\mathcal{I}'} = C_i^{\mathcal{I}}$. Due to the constructions of $D', \mathcal{T}', \mathcal{R}'$, we have $D'^{\mathcal{I}'} \neq \emptyset$, $\mathcal{I}' \models \mathcal{T}'$ and $\mathcal{I}' \models \mathcal{R}'$.

For the converse direction, let $\mathcal{I}'$ be a model of D' w.r.t. $\mathcal{T}'$ and $\mathcal{R}'$. $\mathcal{I}$ will be identical to $\mathcal{I}'$ in every respect except for concrete roles and datatype group-based concepts $C_1, \ldots, C_k$. We can construct $C_i^{\mathcal{I}}$ ($1 \leq i \leq k$) as $C_i^{\mathcal{I}} = A_{C_i}^{\mathcal{I}'}$ and the interpretations of concrete roles as follows: Let $C = C_1^{\mathcal{I}} \cup \ldots \cup C_k^{\mathcal{I}}$. For each $x_j \in C$,

there exists a set $\{C_{j_1}, \ldots, C_{j_{n_x}}\}$ s.t. for each $C_{j_h} \in \{C_{j_1}, \ldots, C_{j_{n_x}}\}$, $x_j \in C_{j_h}^{\mathcal{I}}$. Let $S_j = C_{j_1} \sqcap \ldots \sqcap C_{j_{n_x}}$. Obviously, $\mathcal{I} \models S_j$. Due to claim (i), the datatype query $\pi(S_j)$ is decidable; therefore, there exists a datatype interpretation $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ and a solution δ of $\pi(S_j)$ w.r.t. $(\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$. Let T be a concrete role, $V_T^{(j)}$ the set of variables in $\pi(S_j)$ that match T, $\delta(V_T^{(j)})$ the set of data values to which δ maps the set of variables in $V_T^{(j)}$. Initially, we set all $T^{\mathcal{I}}$ as $\emptyset$, then for each T used in each S_j, we have $T^{\mathcal{I}} = T^{\mathcal{I}} \cup \{S_j^{\mathcal{I}} \times \delta(V_T^{(j)})\}$. Obviously, we have $\mathcal{I} \models D$. Due to claim (ii) and the construction of $\mathcal{T}'$, we have $\mathcal{I} \models \mathcal{T}$. Due to the definition of match, the constructions of $\mathcal{R}'$ and the interpretations of concrete roles, we have $\mathcal{I} \models \mathcal{R}$. □

5.3 Related Work

The story started when Lutz [85] proved that the $\mathcal{ALC}(\mathcal{D})$-concept satisfiability problem w.r.t. TBoxes is usually undecidable even when we use simple concrete domains, such as some arithmetic concrete domains. In order to retain decidability, an obvious way out is to restrict the powerful concrete domain constructor, viz. feature chains (cf. Definition 2.10), to concrete features.

In this direction, Haarslev et al. [50] proved that $\mathcal{SHN}(\mathcal{D})^-$-concept satisfiability w.r.t. TBoxes and RBoxes is decidable, where $(\mathcal{D})^-$ means disallowing feature chains. Horrocks and Sattler [75] showed that the $\mathcal{SHOQ}(\mathbf{D})$-concept satisfiability w.r.t. TBoxes and RBoxes is decidable, where **D** is a universal concrete domain (cf. Definition 2.12). In $\mathcal{SHOQ}(\mathbf{D})$ (cf. Definition 2.3.2), feature chains are disallowed and only unary datatype predicates are considered. This result is strengthened by Pan and Horrocks [109], who show that $\mathcal{SHOQ}(\mathbf{D})$ can be extended to allow arbitrary arity datatype predicates and predicate qualified number restriction constructors to give $\mathcal{SHOQ}(\mathbf{D_n})$, and $\mathcal{SHOQ}(\mathbf{D_n})$-concept satisfiability w.r.t. TBoxes and RBoxes is still decidable.

Interestingly, Baader et al. [5] shows that we can combine an arbitrary DL $\mathcal{L}'$ with $(\mathcal{D})^-$ to give $\mathcal{L}'(\mathcal{D})^-$, and $\mathcal{L}'(\mathcal{D})^-$-concept satisfiability w.r.t. TBoxes and RBoxes is decidable if:

1. $\mathcal{L}'$ provides the conjunction ($\sqcap$) and full concept negation ($\neg$) constructors;
2. $\mathcal{L}'$ does *not* provide any of the following constructors: nominals ($\mathcal{O}$), role complement ($\bar{\cdot}$), universal role (U);
3. $\mathcal{L}'$-concept satisfiability w.r.t. TBoxes and RBoxes is decidable.

Obviously, Baader et al. [5]'s result covers $\mathcal{SHN}(\mathcal{D})^-$ and many other DLs integrated with $(\mathcal{D})^-$, but not $\mathcal{SHOQ}(\mathbf{D})$ or $\mathcal{SHOQ}(\mathbf{D_n})$.

As far as the Description Logics of practical interest (e.g., the $\mathcal{AL}$-family DLs) are concerned, we have further generalised Baader et al. [5]'s result with Theorem 5.19: $\mathcal{L}(\mathcal{G})$-concept satisfiability w.r.t. TBoxes and RBoxes is decidable if

1. $\mathcal{L}$ provides the conjunction ($\sqcap$) and bottom ($\bot$) constructors;
2. $\mathcal{L}$-concept satisfiability w.r.t. TBoxes and RBoxes is decidable.

$\mathcal{SHN}(\mathcal{D})^-$, $\mathcal{SHOQ}(\mathbf{D})$ and $\mathcal{SHOQ}(\mathbf{D_n})$ are all $\mathcal{G}$-combinable Description Logics.

In Chapter 6, we shall provide two extensions of OWL DL, which use the $\mathcal{SHOIQ}(\mathcal{G})$ and $\mathcal{SHOIN}(\mathcal{G}_1)$ $\mathcal{G}$-combined Description Logics as their underpinnings.

Chapter Achievements

- The datatype group approach provides a general formalism to unify existing ontology-related datatype and predicate formalisms and to overcome their limitations.
- In the unified formalism, $\mathcal{G}$-datatype expressions can be used to represent customised datatypes and predicates.
- The family of $\mathcal{G}$-combinable Description Logics are identified, in the formalism, to integrate with conforming datatype groups, so as to give decidable Description Logics that can provide customised datatypes and datatype predicates.
- Theorem 5.19 generalises (almost) all the known decidability results, about concept satisfiability w.r.t. TBoxes and RBoxes, of feature-chain-free Description Logics combined with admissible concrete domains.

Chapter 6

OWL-E: OWL Extended with Datatype Expressions

Chapter Aims

- To provide decidable extensions of OWL DL that support customised datatypes and datatype predicates.
- To design practical decision procedures for a family of Description Logics that are closely related to OWL DL and its above extensions.

Chapter Plan

In Chapter 5, the datatype group-based framework extends OWL datatyping to support customised datatypes and predicates. In this chapter, accordingly, we will propose OWL-E and OWL-Eu, which are decidable extensions of OWL DL with datatype groups, and present *practical* decision procedures for a family of Description Logics that are related to OWL DL, OWL-Eu and OWL-E.

6.1 The Design of OWL-E

A serious limitation of OWL DL is that its datatype support is too limited, i.e., it does not provide customised datatypes and predicates. Therefore, we have designed

an extension of OWL DL that overcomes this limitation.

6.1.1 Meeting the Requirements

Four main requirements have been considered when OWL-E is designed:

1. OWL-E should provide customised datatypes and predicates; therefore, it should be based on a datatype formalism which is compatible with OWL datatyping and provides facilities to construct customised datatypes and predicates.

2. OWL-E should be a decidable extension of OWL DL.

3. It is desirable that OWL-E is also an extension of DAML+OIL.

4. It is desirable for OWL-E to overcome all the other limitations mentioned in the 'Limitations of OWL Datatyping' section on page 77 (i.e., OWL-E should provide negations of datatypes, the datatype domain should properly accommodate interpretations of typed literals with unsupported predicate URIrefs, and OWL-E should provide names for enumerated datatypes).

According to Chapter 5, the datatype group approach matches the datatype formalism needed in Requirement 1, and we can integrate an arbitrary datatype group into a $\mathcal{G}$-combinable Description Logic without losing decidability of concept satisfiability w.r.t. TBoxes and RBoxes. As OWL-E is designed to be an extension of OWL DL, and it is desirable also to be an extension of DAML+OIL (Requirement 3), we should check whether the concept languages of OWL DL (i.e., $\mathcal{SHOIN}$) and DAML+OIL (i.e., $\mathcal{SHOIQ}$) are $\mathcal{G}$-combinable Description Logics. Since both $\mathcal{SHOIN}$-concept and $\mathcal{SHOIQ}$-concept satisfiability problems w.r.t. TBoxes and RBoxes are decidable (Tobies [139, Corollary 6.31]), according to Definition 5.15, both of them are $\mathcal{G}$-combinable Description Logics. Given that both $\mathcal{SHOIN}(\mathcal{G})$ and $\mathcal{SHOIQ}(\mathcal{G})$ are closed under negation, Theorem 5.19 now yields the following decidability result.

Corollary 6.1. *The $\mathcal{SHOIN}(\mathcal{G})$- and $\mathcal{SHOIQ}(\mathcal{G})$-concept satisfiability and subsumption problems w.r.t. TBoxes and RBoxes are decidable.*

Combining this result with Theorem 2.7, we obtain the following theorem.

Theorem 6.2. *The knowledge base satisfiability problems of $\mathcal{SHOIN}(\mathcal{G})$ and $\mathcal{SHOIQ}(\mathcal{G})$ are decidable.*

According to Schaerf [128], if a Description Logic is closed under negation, then all the basic reasoning services (described in Section 2.1.3) are reducible to knowledge base satisfiability. Therefore, Theorem 6.2 indicates the following theorem.

Theorem 6.3. *All the basic reasoning services of $\mathcal{SHOIN}(\mathcal{G})$ and $\mathcal{SHOIQ}(\mathcal{G})$ are decidable.*

Since $\mathcal{SHOIN}(\mathcal{G})$ is a special case of $\mathcal{SHOIQ}(\mathcal{G})$, if we choose $\mathcal{SHOIQ}$ as the underpinning of OWL-E, then we can satisfy both Requirement 2 and 3.

Finally, we consider the minor limitations of OWL DL mentioned in the 'Limitations of OWL Datatyping' (Requirement 4). The first two limitations, i.e., those about negations and the datatype domain, have been considered in the datatype group approach (cf. Section 5.1.3), and we only need to take care of the last limitation, i.e., names for enumerated datatypes. In response to this requirement, OWL-E allows users to define names (in fact, URIrefs) for $\mathcal{G}$-datatype expressions.

6.1.2 From OWL DL to OWL-E

As discussed in the last section, OWL-E extends OWL DL as follows.

1. OWL-E contains an arbitrary conforming datatype group. Informally speaking, a datatype group contains a set of datatypes and the predicates defined over them (cf. Definition 5.4 on page 104).

2. OWL-E provides $\mathcal{G}$-datatype expressions. Given a datatype group $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_\mathcal{G}, \mathsf{dom})$, Table 6.1 shows the abstract syntax and DL syntax of datatype expressions, where u is a datatype predicate URIref, "s_i"^^d_i are typed literals, $v_1, \ldots, v_n$ are (possibly negated) unary supported predicate URIrefs, P, Q are datatype expressions and $\Phi_\mathcal{G}$ is the set of supported predicate URIrefs in $\mathcal{G}$. Note that there are some extra syntactic conditions on the conjunctive and disjunctive expressions: if both P and Q have fixed arity, viz. $a(P)$ and $a(Q)$, then we require $a(P) = a(Q)$; or if one of them has minimum arity (suppose it is P), then we require $a_{min}(P) \leq a(Q)$ (cf. Definition 5.12 on page 110). Unary datatype expressions can be used as data ranges in datatype range axioms (cf. Table 3.5), while arbitrary datatype expressions can be used in datatype group-based class descriptions (cf. Table 6.3).

Abstract Syntax	DL Syntax	Semantics
rdfs:Literal	$\top_{\mathrm{D}}$	$\Delta_{\mathbf{D}}$
owlx:DatatypeBottom	$\bot_{\mathrm{D}}$	$\emptyset$
u a predicate URIref	u	$u^{\mathbf{D}}$
`not`(u)	$\overline{u}$	if $u \in \mathbf{D}_{\mathcal{G}}$, $\Delta_{\mathbf{D}} \setminus u^{\mathbf{D}}$ if $u \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}$, $(\mathsf{dom}(u))^{\mathbf{D}} \setminus u^{\mathbf{D}}$ if $u \notin \Phi_{\mathcal{G}}$, $\bigcup_{n>1}(\Delta_{\mathbf{D}})^n \setminus u^{\mathbf{D}}$
`oneOf`("s_1"^^d_1 ... "s_n"^^d_n)	$\{$"s_1"^^$d_1, \ldots,$ "s_n"^^$d_n\}$	$\{(\text{“}s_1\text{”}\text{\^{}\^{}}d_1)^{\mathbf{D}}\} \cup \ldots \cup \{(\text{“}s_n\text{”}\text{\^{}\^{}}d_n)^{\mathbf{D}}\}$
`domain`($v_1, \ldots, v_n$)	$[v_1, \ldots, v_n]$	$v_1^{\mathbf{D}} \times \ldots \times v_n^{\mathbf{D}}$
`and`(P, Q)	$P \wedge Q$	$P^{\mathbf{D}} \cap Q^{\mathbf{D}}$
`or`(P, Q)	$P \vee Q$	$P^{\mathbf{D}} \cup Q^{\mathbf{D}}$

Table 6.1: OWL-E datatype expressions

Abstract Syntax	Semantics
$DatatypeExpression(u\ E)$	$u^{\mathbf{D}} = E^{\mathbf{D}}$

Table 6.2: OWL-E datatype expression axiom

Abstract Syntax	DL Syntax	Semantics
restriction($\{T\}$ someTuplesSatisfy(E))	$\exists T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \exists t_1, \ldots, t_n.\langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\}$
restriction($\{T\}$ allTuplesSatisfy(E))	$\forall T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \forall t_1, \ldots, t_n.\langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \rightarrow \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\}$
restriction($\{T\}$ minCardinality(m) someTuplesSatisfy(E))	$\geqslant m T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle t_1, \ldots, t_n\rangle \mid \langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\} \geq m\}$
restriction($\{T\}$ maxCardinality(m) someTuplesSatisfy(E))	$\leqslant m T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle t_1, \ldots, t_n\rangle \mid \langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\} \leq m\}$
restriction(R minCardinality(m) someValuesFrom(C))	$\geqslant m R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \mid \langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq m\}$
restriction(R maxCardinality(m) someValuesFrom(C))	$\leqslant m R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \mid \langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq m\}$

Table 6.3: OWL-E introduced class descriptions

3. OWL-E provides *datatype expression axioms*, which introduce URIrefs for datatype expressions. Note that the set of URIrefs of datatype expressions should be disjoint from the URIrefs of classes, properties, individuals datatypes and datatype predicates, etc. Table 6.2 shows the abstract syntax of OWL-E datatype expression axioms, where u is a datatype expression URIref and E is a datatype expression as defined in Table 6.1.

4. OWL-E provides some new classes descriptions, which are listed in Table 6.3, where $T, T_1, \ldots, T_n$ are datatype properties (where $T_i \not\sqsubseteq^* T_j, T_j \not\sqsubseteq^* T_i$ for all $1 \leq i < j \leq n$),[1] R is an object property, C is a class, E is a datatype expression or a

[1] $\sqsubseteq^*$ is the transitive reflexive closure of $\sqsubseteq$.

datatype expression URIref, and ♯ denotes cardinality. Note that the first four are datatype group-based class descriptions, and the last two are qualified number restrictions.

6.1.3 How OWL-E Helps

Now we revisit the examples presented in Section 1.3 to show that how OWL-E supports the use of customised datatypes and predicates in various SW and ontology applications.

Example 6.1 OWL-E: Matchmaking

A service requirement may ask for a PC with memory size greater than 512Mb, unit price less than 700 pounds and delivery date earlier than 15/03/2004. The set of PCs that satisfy this required capability can be expressed by the following OWL-E class description.

```
Class(RequiredPC complete PC
   restriction(memorySizeInMb
      maxCardinality(1) minCardinality(1)
      someTuplesSatisfy(owlx:integerGreatThanx=512))
   restriction(priceInPound
      maxCardinality(1) minCardinality(1)
      someTuplesSatisfy(owlx:integerLessThanx=700))
   restriction(deliveryDate
      maxCardinality(1) minCardinality(1)
      someTuplesSatisfy(owlx:dateEarlierThanx=2004-03-15)))
```

This class description says, informally speaking, a RequiredPC is a PC which relates to (at most) three values, through the datatype properties $memorySizeInMb$, $pricingPound$ and $deliveryDate$, that satisfy the unary predicate URIrefs owlx:integerGreatThanx=512, owlx:integerLessThanx=700 and owlx:dateEarlierThanx=2004-03-15, respectively. ♢

Example 6.2 OWL-E: Unit Mapping

Now we revisit Example 1.4 on page 21. Assume we need to ensure that the distance in miles (in ontology A) should be equal to 1.6 times the data value of distance in kilometres (in the current ontology B) between the same pairs of locations. We can use the following OWL-E class inclusion axioms to represent this constraint:

```
SubClassOf(owl:Thing
    restriction(a:distanceInMile distanceInKilometre
        allTuplesSatisfy(and(owlx:doubleMultiplicationx=1.6
            domain(xsd:double xsd:double)))))
```

where owlx:doubleMultiplicationx=1.6 is the predicate URIref for $*_{[y]}^{double}$ (cf. Example 5.7 on page 112). The axiom ensures that if an object has all the two datatype properties, then the value it relates to through the $distanceInKilometre$ property should be 1.6 times of that through the $distanceInMile$ property. ♢

Example 6.3 OWL-E: Small Items

Assume that electronic-shops want to define small items as items of which the sum of height, length and width is no greater than 15cm. SmallItem can be represented by the following class description:

```
Class(SmallItem complete Item
    restriction(hlwSumInCm heightInCm lengthInCm widthInCm
        maxCardinality(1) minCardinality(1)
        someTuplesSatisfy(sumNoGreaterThan15)))
DatatypeExpression(sumNoGreaterThan15 and(
    owlx:integerAddition
    domain(not(owlx:integerGreaterThanx=15) xsd:integer
        xsd:integer xsd:integer)))
```

The class description says a SmallItem is an Item which relates through the group of datatype properties $hlwSumInCm$, $heightInCm$, $lengthInCm$ and $widthInCm$ to exactly one tuple of values that satisfy the datatype expression $sumNoGreaterThan15$. The datatype expression axiom defines $sumNoGreater$- $Than$ as a conjunction, where the first conjunct is a predicate URIref owlx:integer- Addition and the second conjunct is a domain expression, which requires the first argument of owlx:integerAddition should be no greater than 15 (cf. Example 5.7). Note that datatype expression axioms are very useful when datatype expressions (e.g., $sumNoGreaterThan15$) need to be used multiple times in the ontology. ♢

6.2 Reasoning with OWL-E

In this section, we will investigate how to provide practical decision procedures for OWL-E. To the best of our knowledge, there exist no published tableaux algorithms to handle the $\mathcal{SHOIQ}$ DL. Therefore, in what follows, we use the 'divide and conquer' strategy. That is, instead of providing a decision procedure for the $\mathcal{SHOIQ}(\mathcal{G})$ DL, we provide decision procedures for the $\mathcal{SHOQ}(\mathcal{G})$, $\mathcal{SHIQ}(\mathcal{G})$ and $\mathcal{SHIO}(\mathcal{G})$ DLs, where the $\mathcal{SHOQ}(\mathbf{D})$ DL has been argued to be useful in the context of the Semantic Web [75],[2] and the $\mathcal{SHIQ}$ DL is the underpinning of the OIL Web ontology language and is implemented in popular DL systems like FaCT [59] and RACER [51].

Most importantly, the decision procedures we will present for the above DLs call for a datatype reasoner as a sub-procedure for checking the satisfiability of some datatype queries via a well-defined 'interface'. Therefore, once we have a decision procedure for $\mathcal{SHOIQ}$, we can easily upgrade it to one for $\mathcal{SHOIQ}(\mathcal{G})$ (cf. Theorem 6.14 on page 156).

Before introducing the above DLs, we define some notation. Let $\mathbf{R}_A$ and $\mathbf{R_D}$ be a countable set of abstract role names and datatype role names, such that $\mathbf{R} = \mathbf{R}_A \uplus \mathbf{R_D}$; accordingly, for a Description Logic $\mathcal{L}$, let $\mathbf{Rdsc}_A(\mathcal{L})$ and $\mathbf{Rdsc_D}(\mathcal{L})$ be a set of abstract role expressions and a set of concrete role expressions (or simply called *abstract roles* and *concrete roles*) of $\mathcal{L}$, respectively. For a datatype group $\mathcal{G}$, let $\mathbf{DPexp}(\mathcal{G})^{(1)} \subseteq \mathbf{Dexp}(\mathcal{G})$ be a set of unary $\mathcal{G}$-predicate expressions. We use $\top_{\mathrm{D}}$ and $\bot_{\mathrm{D}}$ to represent $\mathrm{rdfs{:}Literal}$ and $\mathrm{owlx{:}DatatypeBottom}$, respectively, in the DL syntax.

6.2.1 The $\mathcal{SHOQ}(\mathcal{G})$ DL

Syntax and Semantics

The $\mathcal{SHOQ}(\mathcal{G})$ DL extends the $\mathcal{SHOQ}(\mathbf{D})$ DL (cf. Section 2.3.2) by replacing the two datatype related constructors (cf. Table 2.2 on page 46) with the four datatype group-based concept constructors given in Table 5.2 on page 118. I.e., $\mathcal{SHOQ}(\mathcal{G})$-roles and $\mathcal{SHOQ}(\mathcal{G})$-RBoxes are the same as $\mathcal{SHOQ}(\mathbf{D})$-roles and $\mathcal{SHOQ}(\mathbf{D})$-RBoxes (cf. Definition 2.14 on page 47).

Formally, $\mathcal{SHOQ}(\mathcal{G})$-concepts are defined as followed.

[2] $\mathcal{SHOQ}(\mathbf{D})$ is a special case of $\mathcal{SHOQ}(\mathcal{G})$, where datatype expressions are restricted to simply datatype URIrefs and datatype-related number restrictions are not allowed; cf. Table 2.2 on page 46 and Table 5.2 on page 118.

Definition 6.4. **($\mathcal{SHOQ}(\mathcal{G})$-concepts)** Let $\mathsf{CN} \in \mathbf{C}$, $R \in \mathbf{Rdsc}_A(\mathcal{SHOQ}(\mathcal{G}))$, T_1, $\ldots, T_n \in \mathbf{Rdsc}_{\mathbf{D}}(\mathcal{SHOQ}(\mathcal{G}))$ and $T_i \not\sqsubseteq T_j, T_j \not\sqsubseteq T_i$ for all $1 \leq i < j \leq n$, $C, D \in \mathbf{Cdsc}(\mathcal{SHOQ}(\mathcal{G}))$, $o \in \mathbf{I}$, $E \in \mathbf{Dexp}(\mathcal{G})$, $n, m \in \mathbb{N}$, $n \geq 1$. Valid $\mathcal{SHOQ}(\mathcal{G})$-concepts are defined by the abstract syntax:

$$\begin{array}{rcl} C & ::= & \top \mid \bot \mid \mathsf{CN} \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \{o\} \\ & & \exists R.C \mid \forall R.C \mid \geqslant mR.C \mid \leqslant mR.C \\ & & \exists T_1, \ldots, T_n.E \mid \forall T_1, \ldots, T_n.E \mid \geqslant mT_1, \ldots, T_n.E \mid \leqslant mT_1, \ldots, T_n.E. \end{array}$$

The semantics of datatype group-based $\mathcal{SHOQ}(\mathcal{G})$-concepts is given in Table 5.2 on page 118; the semantics of other $\mathcal{SHOQ}(\mathcal{G})$-concepts is given in Table 2.2 on page 46. ◇

A $\mathcal{SHOQ}(\mathcal{G})$-Tableau

Since the $\mathcal{SHOQ}(\mathcal{G}_1)$-concept satisfiability and subsumption problem w.r.t. to a knowledge base can be reduced to the $\mathcal{SHOQ}(\mathcal{G}_1)$-concept satisfiability problem w.r.t. an RBox (cf. Section 2.1.3), we only discuss a tableaux algorithm for the latter problem.

For ease of presentation, we assume all concepts to be in *negation normal form* (NNF): a concept is in negation normal form iff negation is applied only to atomic concept names, nominals or datatype expressions. Each $\mathcal{SHOQ}(\mathcal{G})$-concept can be transformed into an equivalent one in NNF by pushing negation inwards, making use of DeMorgan's laws and the following equivalences:

$$\begin{array}{rcl@{\qquad}rcl} \neg\exists R.C & \equiv & \forall R.(\neg C) & \neg\exists T_1, \ldots, T_n.E & \equiv & \forall T_1, \ldots, T_n.(\neg E) \\ \neg\forall R.C & \equiv & \exists R.(\neg C) & \neg\forall T_1, \ldots, T_n.E & \equiv & \exists T_1, \ldots, T_n.(\neg E) \\ \neg\leqslant nR.C & \equiv & \geqslant (n+1)R.C & \neg\leqslant mT_1, \ldots, T_n.E & \equiv & \geqslant (m+1)T_1, \ldots, T_n.E \\ \neg\geqslant (n+1)R.C & \equiv & \leqslant nR.C & \neg\geqslant (m+1)T_1, \ldots, T_n.E & \equiv & \leqslant mT_1, \ldots, T_n.E \\ \neg\geqslant 0R.C & \equiv & \bot & \neg\geqslant 0T_1, \ldots, T_n.E & \equiv & \bot. \end{array}$$

We use $\sim C$ to denote the NNF of $\neg C$.

Like other tableaux algorithms, the tableaux algorithm for $\mathcal{SHOQ}(\mathcal{G})$ tries to prove the satisfiability of a concept expression D by constructing a model of D. The model is represented by a so-called completion forest (cf. Section 2.2), nodes of which correspond to either individuals (labelled nodes) or variables (non-labelled nodes), each labelled node being labelled with a set of $\mathcal{SHOQ}(\mathcal{G})$-concepts. When testing the satisfiability of a $\mathcal{SHOQ}(\mathcal{G})$-concept D, these sets are restricted to subsets of $\mathsf{cl}(D)$,

which denotes the set of all sub-concepts of D and the NNF of their negations. We use $\mathsf{cl}_{d_n}(D)$ to denote the set of all the $\mathcal{G}$-datatype expressions and their negations occurring in these (NNFs of) sub-concepts.

The soundness and completeness of the algorithm will be proved by showing that it creates a *tableau* for D.

Definition 6.5. Let D be a $\mathcal{SHOQ}(\mathcal{G})$-concept in NNF, $\mathcal{R}$ a $\mathcal{SHOQ}(\mathcal{G})$ RBox, $\mathbf{R}_A^D$ and $\mathbf{R}_\mathbf{D}^D$ the sets of abstract and concrete roles, respectively, occurring in D or $\mathcal{R}$, $\mathcal{G}$ a datatype group, $P \in \mathsf{cl}_{d_n}(D)$ a (possibly negated) datatype predicate expression. A *tableau* $\mathcal{T} = (\mathbf{S}_A, \mathbf{S}_\mathbf{D}, \mathcal{L}, DC^\mathcal{T}, \mathcal{E}_A, \mathcal{E}_\mathbf{D})$ for D w.r.t. $\mathcal{R}$ is defined as follows:

- $\mathbf{S}_A$ is a set of individuals,
- $\mathbf{S}_\mathbf{D}$ is a set of variables,
- $\mathcal{L} : \mathbf{S}_A \rightarrow 2^{\mathsf{cl}(D)}$ maps each individual to a set of concepts which is a sub-set of $\mathsf{cl}(D)$,
- $DC^\mathcal{T}$ is a set of datatype constraints of the form $P(v_1, \ldots, v_n)$, where $P \in \mathsf{cl}_{d_n}(D)$ and $v_1, \ldots, v_n \in \mathbf{S}_\mathbf{D}$,
- $\mathcal{E}_A : \mathbf{R}_A^D \rightarrow 2^{\mathbf{S}_A \times \mathbf{S}_A}$ maps each abstract role in $\mathbf{R}_A^D$ to a set of pairs of individuals,
- $\mathcal{E}_\mathbf{D} : \mathbf{R}_\mathbf{D}^D \rightarrow 2^{\mathbf{S}_A \times \mathbf{S}_\mathbf{D}}$ maps each concrete role in $\mathbf{R}_\mathbf{D}^D$ to a set of pairs of individuals and variables.

There is some individual $s_0 \in \mathbf{S}_A$ such that $D \in \mathcal{L}(s_0)$. For all $s, x \in \mathbf{S}_A$, $v \in \mathbf{S}_\mathbf{D}$, $C, C_1, C_2 \in \mathsf{cl}(D)$, $R, S \in \mathbf{R}_A^D$, $T, T' \in \mathbf{R}_\mathbf{D}^D$, $d \in \mathsf{cl}_{d_1}(D)$,

$$S^\mathcal{T}(s, C) := \{x \mid \langle s, x\rangle \in \mathcal{E}_A(S) \wedge C \in \mathcal{L}(x)\},$$

and it holds that:

(P0) there exists a datatype interpretation $\mathcal{I}_\mathbf{D} = (\Delta_\mathbf{D}, \cdot^\mathbf{D})$ and a mapping δ from $\mathbf{S}_\mathbf{D}$ to $\Delta_\mathbf{D}$ s.t. for each $P(v_1, \ldots, v_n) \in DC^\mathcal{T}$, there exist $t_i = \delta(v_i)$ for all $1 \leq i \leq n$, and $\langle t_1, \ldots, t_n\rangle \in P^\mathbf{D}$; based on $\mathcal{I}_\mathbf{D}$ and δ, we define $T_1, \ldots, T_n^\mathcal{T}(s, P)$ as follows

$$T_1, \ldots, T_n^\mathcal{T}(s, P) := \{\langle \delta(v_1), \ldots, \delta(v_n)\rangle \mid \langle s, v_1\rangle \in \mathcal{E}_\mathbf{D}(T_1) \wedge \ldots \wedge \langle s, v_n\rangle \in \mathcal{E}_\mathbf{D}(T_n) \\ \wedge \langle \delta(v_1), \ldots, \delta(v_n)\rangle \in P^\mathbf{D}\},$$

(P1) if $C \in \mathfrak{L}(s)$, then $\sim C \notin \mathfrak{L}(s)$,

(P2) if $C_1 \sqcap C_2 \in \mathfrak{L}(s)$, then $C_1 \in \mathfrak{L}(s)$ and $C_2 \in \mathfrak{L}(s)$,

(P3) if $C_1 \sqcup C_2 \in \mathfrak{L}(s)$, then $C_1 \in \mathfrak{L}(s)$ or $C_2 \in \mathfrak{L}(s)$,

(P4) if $\langle s, x\rangle \in \mathcal{E}_A(R)$ and $R \sqsubseteq^* S$, then $\langle s, x\rangle \in \mathcal{E}_A(S)$, and
if $\langle s, t\rangle \in \mathcal{E}_{\mathbf{D}}(T)$ and $T \sqsubseteq^* T'$, then $\langle s, t\rangle \in \mathcal{E}_{\mathbf{D}}(T')$,

(P5) if $\forall R.C \in \mathfrak{L}(s)$ and $\langle s, x\rangle \in \mathcal{E}_A(R)$, then $C \in \mathfrak{L}(x)$,

(P6) if $\exists R.C \in \mathfrak{L}(s)$, then there is some $x \in \mathbf{S}$ such that $\langle s, x\rangle \in \mathcal{E}_A(R)$ and $C \in \mathfrak{L}(x)$,

(P7) if $\forall S.C \in \mathfrak{L}(s)$ and $\langle s, x\rangle \in \mathcal{E}_A(R)$ for some $R \sqsubseteq^* S$ with $\mathsf{Trans}(R)$, then $\forall R.C \in \mathfrak{L}(x)$,

(P8) if $\geqslant n S.C \in \mathfrak{L}(s)$, then $\sharp S^{\mathcal{I}}(s, C) \geqslant n$,

(P9) if $\leqslant n S.C \in \mathfrak{L}(s)$, then $\sharp S^{\mathcal{I}}(s, C) \leqslant n$,

(P10) if $\{\leqslant n S.C, \geqslant n S.C\} \cap \mathfrak{L}(s) \neq \emptyset$ and $\langle s, x\rangle \in \mathcal{E}_A(S)$, then $\{C, \sim C\} \cap \mathfrak{L}(x) \neq \emptyset$,

(P11) if $\{\mathsf{o}\} \in \mathfrak{L}(s) \cap \mathfrak{L}(x)$, then $s = x$,

(P12) if $\forall T_1, \ldots, T_n.P \in \mathfrak{L}(s)$ and $\langle s, v_i\rangle \in \mathcal{E}_{\mathbf{D}}(T_i)$ for all $1 \leq i \leq n$, then $P(v_1, \ldots, v_n) \in DC^{\mathcal{I}}$,

(P13) if $\exists T_1, \ldots, T_n.P \in \mathfrak{L}(s)$, then there is some $v_1, \ldots, v_n \in \mathbf{S_D}$ such that $\langle s, v_i\rangle \in \mathcal{E}_{\mathbf{D}}(T_i)$ for all $1 \leq i \leq n$, and $P(v_1, \ldots, v_n) \in DC^{\mathcal{I}}$,

(P14) if $\geqslant m T_1, \ldots, T_n.P \in \mathfrak{L}(s)$, then $\sharp T^{\mathcal{I}}(s, P) \geqslant m$,

(P15) if $\leqslant m T_1, \ldots, T_n.P \in \mathfrak{L}(s)$, then $\sharp T^{\mathcal{I}}(s, P) \leqslant m$,

(P16) if $\{\leqslant m T_1, \ldots, T_n.P, \geqslant m T_1, \ldots, T_n.P\} \cap \mathfrak{L}(s) \neq \emptyset$ and $\langle s, v_i\rangle \in \mathcal{E}_{\mathbf{D}}(T_i)$ for all $1 \leq i \leq n$, then we have either $P(v_1, \ldots, v_n) \in DC^{\mathcal{I}}$ or $\neg P(v_1, \ldots, v_n) \in DC^{\mathcal{I}}$. ◇

Lemma 6.6. *A $\mathcal{SHOQ}(\mathcal{G})$-concept D in NNF is satisfiable w.r.t. an RBox $\mathcal{R}$ iff D has a tableau w.r.t. $\mathcal{R}$.*

Proof: For the *if* direction, given a datatype group $\mathcal{G}$, if $\mathcal{T} = (\mathbf{S}_A, \mathbf{S}_\mathbf{D}, \mathcal{L}, DC^\mathcal{T}, \mathcal{E}_A, \mathcal{E}_\mathbf{D})$ is a tableau for D, a model $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I}, \Delta_\mathbf{D}, \cdot^\mathbf{D})$ of D w.r.t. $\mathcal{G}$ can be defined as:

$$\begin{aligned}
\Delta^\mathcal{I} &= \mathbf{S}_A \\
\mathsf{CN}^\mathcal{I} &= \{s \in \mathbf{S}_A \mid \mathsf{CN} \in \mathcal{L}(s)\} \quad \text{for all concept names } \mathsf{CN} \text{ in } \mathsf{cl}(D) \\
R^\mathcal{I} &= \begin{cases} \mathcal{E}_A(R)^+ & \text{if } \mathrm{Trans}(R) \in \mathcal{R} \\ \mathcal{E}_A(R) \cup \bigcup\limits_{P \sqsubseteq^* R, P \neq R} P^\mathcal{I} & \text{otherwise} \end{cases} \\
T^\mathcal{I} &= \{\langle s, \delta(v)\rangle \mid \langle s, v\rangle \in \mathcal{E}_\mathbf{D}(T)\}
\end{aligned}$$

where $\mathcal{E}_A(R)^+$ denotes the transitive closure of $\mathcal{E}_A(R)$, and according to (P0), there exist a datatype interpretation $(\Delta_\mathbf{D}, \cdot^\mathbf{D})$ of $\mathcal{G}$.

The interpretation of non-transitive roles is recursive in order to correctly interpret those non-transitive roles that have a transitive sub-role. From the definition of $R^\mathcal{I}$ and (P4) of a tableau, it follows that, if $\langle s, x\rangle \in S^\mathcal{I}$, then either $\langle s, x\rangle \in \mathcal{E}_A(S)$ directly, or there exists a path $\langle s, s_1\rangle$, $\langle s_1, s_2\rangle$, ..., $\langle s_n, x\rangle \in \mathcal{E}_A(R)$ for some R with $\mathrm{Trans}(R)$ and $R \sqsubseteq^* S$. From the definition of $T^\mathcal{I}$ and (P4) of a tableau, it follows that, if $\langle s, v\rangle \in \mathcal{E}_\mathbf{D}(T)$ and $T \sqsubseteq^* T'$, then $\langle s, \delta(v)\rangle \in T'^\mathcal{I}$.

To show that $\mathcal{I}$ is a model of D w.r.t. $\mathcal{R}$, we have to prove (1) $\mathcal{I} \models \mathcal{R}$ and (2) $D^\mathcal{I} \neq \emptyset$. The first one is obvious due to (P4) and the construction of the model. The second one is shown by proving $\mathcal{I} \models \mathcal{R}$ and $C \in \mathcal{L}(s) \Rightarrow s \in C^\mathcal{I}$ for any $s \in \mathbf{S}_A$. This implies $D^\mathcal{I} \neq \emptyset$ since T is a tableau for D and hence there must be some $s_0 \in \mathbf{S}_A$ with $D \in \mathcal{L}(s_0)$.

We will use an induction on the structure of concept to show that $C \in \mathcal{L}(s)$ implies $s \in C^\mathcal{I}$ for any C. The two base cases of the induction are $C = \mathsf{CN}$ or $C = \neg\mathsf{CN}$. If $\mathsf{CN} \in \mathcal{L}(s)$, then by definition of $\mathsf{CN}^\mathcal{I}$, $s \in \mathsf{CN}^\mathcal{I}$. If $\neg\mathsf{CN} \in \mathcal{L}(s)$, then by (P1), $\mathsf{CN} \notin \mathcal{L}(s)$ and hence $s \notin \mathsf{CN}^\mathcal{I}$. For the induction step we have to distinguish several cases:

- $C = C_1 \sqcap C_2$. Since T is a tableau, $C \in \mathcal{L}(s)$ implies $C_1 \in \mathcal{L}(s)$ and $C_2 \in \mathcal{L}(s)$. Hence, by induction, we have $s \in C_1^\mathcal{I}$ and $s \in C_2^\mathcal{I}$, which yields $s \in (C_1 \sqcap C_2)^\mathcal{I}$.
- $C = C_1 \sqcup C_2$. Similar to the previous case.
- $C = \forall S.E$. Let $s \in \mathbf{S}_A$ with $C \in \mathcal{L}(s)$ and $x \in \mathbf{S}_A$ an arbitrary individual such that $\langle s, x\rangle \in \mathcal{E}_A(S)$. Then there are two possibilities:
  - $\langle s, x\rangle \in \mathcal{E}_A(S)$. Then (P5) implies that $E \in \mathcal{L}(x)$ and by induction $x \in$

$E^{\mathcal{I}}$.

– $\langle s, x\rangle \notin \mathcal{E}_A(S)$. Then there exists a path $\langle s, s_1\rangle$, $\langle s_1, s_2\rangle$, ..., $\langle s_n, x\rangle \in \mathcal{E}_A(R)$ for some R with $\mathrm{Trans}(R)$ and $R \boxdot S$. Then (P7) implies $\forall R.E \in \mathcal{L}(s_i)$ for all $1 \leq i \leq n$, and $\forall R.E \in \mathcal{L}(x)$. From (P5), we have $E \in \mathcal{L}(x)$. By induction, this implies $x \in E^{\mathcal{I}}$.

- $C = \exists S.E$. Since T is a tableau, $C \in \mathcal{L}(s)$ implies the existence of an individual $x \in \mathbf{S}_A$ such that $\langle s, x\rangle \in \mathcal{E}_A(S)$ and $x \in \mathcal{L}(E)$. By induction, we have $x \in E^{\mathcal{I}}$, and from the definition of $S^{\mathcal{I}}$ it follows that $\langle s, x\rangle \in S^{\mathcal{I}}$ and hence $s \in C^{\mathcal{I}}$.

- $C = (\geqslant n S.E)$. For an $s \in \mathbf{S}_A$ with $C \in \mathcal{L}(s)$, we have $\sharp S^{\mathcal{T}}(s, E) \geqslant n$. Hence there are n individuals $x_1, \ldots, x_n$ such that $x_i \neq x_j$ for $i \neq j$, $\langle s, x_i\rangle \in \mathcal{E}_A(S)$, and $E \in \mathcal{L}(x_i)$ for all i. By induction, we have $x_i \in E^{\mathcal{I}}$ and , since $\mathcal{E}_A(S) \subseteq S^{\mathcal{I}}$, $s \in C$ also holds.

- $C = (\leqslant n S.E)$. For this case, we need that S is a simple role, which implies that $S^{\mathcal{I}} = \mathcal{E}_A(S)$. Let $s \in \mathbf{S}_A$ with $C \in \mathcal{L}(s)$. Due to (P10) we have $E \in \mathcal{L}(x)$ or $\sim E \in \mathcal{L}(x)$ for each x with $\langle s, x\rangle \in \mathcal{E}_A(S)$. Moreover, $\sharp S^{\mathcal{T}}(s, E) \leqslant n$ holds due to (P9). We define

$$S^{\mathcal{I}}(s, E) := \{x \in \Delta^{\mathcal{I}} \mid \langle s, x\rangle \in S^{\mathcal{I}} \wedge x \in E^{\mathcal{I}}\}$$

and can show that $\sharp S^{\mathcal{I}}(s, E) \leqslant \sharp S^{\mathcal{T}}(s, E)$: assume $\sharp S^{\mathcal{I}}(s, E) > \sharp S^{\mathcal{T}}(s, E)$, this implies the existence of some x with $\langle s, x\rangle \in S^{\mathcal{I}}$ with $x \in E^{\mathcal{I}}$ but $E \notin \mathcal{L}(x)$ (because $S^{\mathcal{I}} = \mathcal{E}_A(S)$). By (P10) this implies $\sim E \in \mathcal{L}(x)$, which, by induction, yields $x \in (\sim E)^{\mathcal{I}}$ in contradiction to $x \in E^{\mathcal{I}}$. Therefore $\sharp S^{\mathcal{I}}(s, E) \leqslant n$, and $s \in C^{\mathcal{I}}$ also holds.

- $C = \{o\}$. Let $s, x \in \mathbf{S}_A$, $\{o\} \in \mathcal{L}(s) \cap \mathcal{L}(x)$ implies $\{o\} \in \mathcal{L}(s)$ and $\{o\} \in \mathcal{L}(x)$. By induction, we have $s \in \{o\}$ and $x \in \{o\}$ and hence $s \in C^{\mathcal{I}}$. By (P11) we have $s = x$, which ensures that nominals are interpreted as singletons.

- $C = \forall T_1, \ldots, T_n.P$. Let $s \in \mathbf{S}_A$ with $C \in \mathcal{L}(s)$, and $v_1, \ldots, v_n \in \mathbf{S_D}$ with $\langle s, v_i\rangle \in \mathcal{E}_{\mathbf{D}}(T_i)$ for all $1 \leq i \leq n$. By the definition of $T^{\mathcal{I}}$, we have $\langle s, \delta(v_i)\rangle \in T_i^{\mathcal{I}}$ for all $1 \leq i \leq n$. (P12') implies that $P(v_1, \ldots, v_n) \in DC^{\mathcal{T}}$. By (P0'), we have $\langle \delta(v_1), \ldots, \delta(v_n)\rangle \in P^{\mathbf{D}}$, hence $s \in C^{\mathcal{I}}$.

- $C = \exists T_1, \ldots, T_n.P$. Since $\mathcal{T}$ is a tableau, $C \in \mathcal{L}(s)$ implies the existence of variables $v_1, \ldots, v_n \in \mathbf{S_D}$ such that $\langle s, v_i\rangle \in \mathcal{E}_{\mathbf{D}}(T_i)$ for all $1 \leq i \leq n$, and

$P(v_1, \ldots, v_n) \in DC^{\mathfrak{T}}$. By the definition of $T^{\mathcal{I}}$, we have $\langle s, \delta(v_i)\rangle \in T_i^{\mathcal{I}}$ for all $1 \leq i \leq n$. By (P0'), we have $\langle \delta(v_1), \ldots, \delta(v_n)\rangle \in P^{\mathbf{D}}$, hence $s \in C^{\mathcal{I}}$.

- $C = (\geqslant m T_1, \ldots, T_n.P)$. For an s with $C \in \mathfrak{L}(s)$, we have $\sharp T^{\mathfrak{T}}(s, P) \geqslant m$. Hence there are $m \cdot n$ variables $v_{11}, \ldots, v_{mn} \in \mathbf{S_D}$ such that $\langle s, v_{ij}\rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$, $\langle \delta(v_{i1}), \ldots, \delta(v_{in})\rangle \in P^{\mathbf{D}}$ and $(\langle \delta(v_{i1}), \ldots, \delta(v_{in})\rangle, \langle \delta(v_{k1}), \ldots, \delta(v_{kn})\rangle) \in \neq^{\mathbf{D}}$ for all $1 \leq i < k \leq m$, $1 \leq j \leq n$.[3] By the definition of $T^{\mathcal{I}}$, we have $\langle s, \delta(v_{ij})\rangle \in T_j^{\mathcal{I}}$ for $1 \leq i \leq m$, $1 \leq j \leq n$. Hence $s \in C^{\mathcal{I}}$.

- $C = (\leqslant m T_1, \ldots, T_n.P)$. Let $s \in \mathbf{S}_A$ with $C \in \mathfrak{L}(s)$. Due to (P16') we have $P(v_1, \ldots, v_n) \in DC^{\mathfrak{T}}$ or $\neg P(v_1, \ldots, v_n) \in DC^{\mathfrak{T}}$ for each tuple $\langle v_1, \ldots, v_n\rangle$ with $\langle s, v_i\rangle \in \mathcal{E}_{\mathbf{D}}(T_i)$ for all $1 \leq i \leq n$. Moreover, $\sharp T^{\mathfrak{T}}(s, P) \leqslant m$ holds due to (P15'). We define

$$T^{\mathcal{I}}(s, P) = \{\langle t_1, \ldots, t_n\rangle \mid \langle s, t_i\rangle \in T_i^{\mathcal{I}} \text{ (for all } 1 \leq i \leq n) \wedge \langle t_1, \ldots, t_n\rangle \in P^{\mathbf{D}}\}.$$

  We can show that $\sharp T^{\mathcal{I}}(s, P) \leqslant \sharp T^{\mathfrak{T}}(s, P)$: assume $\sharp T^{\mathcal{I}}(s, P) > \sharp T^{\mathfrak{T}}(s, P)$. This implies the existence of some tuple $\langle t_1, \ldots, t_n\rangle$ with $\langle s, t_i\rangle \in T_i^{\mathcal{I}}$ for all $1 \leq i \leq n$ and $\langle t_1, \ldots, t_n\rangle \in P^{\mathbf{D}}$, but there is no $v_1, \ldots, v_n$ such that $\delta(v_i) = t_i$ for all $1 \leq i \leq n$, which is in contradiction with the definition of $T^{\mathcal{I}}$.

For the *only-if* direction, we have to show that satisfiability of D w.r.t. $\mathcal{R}$ implies the existence of a tableau $\mathfrak{T}$ for D w.r.t. $\mathcal{R}$.

Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}, \Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ be a model of D w.r.t. $\mathcal{G}$ and $\mathcal{I} \models \mathcal{R}$. A tableau $\mathfrak{T} = (\mathbf{S}_A, \mathbf{S_D}, \mathfrak{L}, DC^{\mathfrak{T}}, \mathcal{E}_A, \mathcal{E}_{\mathbf{D}})$ for D can be defined as:

- $\mathbf{S}_A = \Delta^{\mathcal{I}}$;
- $\mathfrak{L}(s) = \{C \in \mathsf{cl}(D) \mid s \in C^{\mathcal{I}}\}$;
- $\mathcal{E}_A(R) = R^{\mathcal{I}}$, where R is an abstract role;
- we initialise $\mathbf{S_D}$ as an empty set, δ and $\mathcal{E}_{\mathbf{D}}$(T) as empty relations, and construct them as follows: for all concrete role T and for each $\langle s, t\rangle \in T^{\mathcal{I}}$,
  1. we generate a new variable v,
  2. $\mathbf{S_D} := \mathbf{S_D} \cup \{v\}$,

[3] $\neq$ is the value inequality predicate; cf. Definition 5.17 on page 119.

3. $\delta := \delta \cup \{\langle v, t\rangle\}$,
4. $\mathcal{E}_{\mathbf{D}}(T) := \mathcal{E}_{\mathbf{D}}(T) \cup \{\langle s, v\rangle\}$;

- $DC^{\mathcal{T}} := \{P(v_1, \ldots, v_n) \mid \exists v_1, \ldots, v_n \in \mathbf{S_D}.P \in \mathsf{cl}_{d_n}(D), \delta(v_i) = t_i$ (for all $1 \leq i \leq n$) and $\langle t_1, \ldots, t_n\rangle \in P^{\mathbf{D}}\}$.

It remains to demonstrate that $\mathcal{T}$ is a tableau for D:

- (P0): Since $\mathcal{I}$ is an interpretation w.r.t. to $\mathcal{G}$, $\mathcal{I}_{\mathbf{D}} = (\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ is a datatype interpretation of $\mathcal{G}$. By definition of $DC^{\mathcal{T}}$, for each $P(v_1, \ldots, v_n) \in DC^{\mathcal{T}}$, there exist $t_1, \ldots, t_n$ with $t_i = \delta(v_i)$ for all $1 \leq i \leq n$ and $\langle t_1, \ldots, t_n\rangle \in P^{\mathbf{D}}$.

- (P1) - (P3), (P5) - (P6), and (P8) - (P11) are satisfied as a direct consequence of the definition of the semantics of $\mathcal{SHOQ}(\mathcal{G}_1)$-concepts.

- (P4) is satisfied because $\mathcal{I} \models \mathcal{R}$.

- (P7): If $s \in (\forall S.C)^{\mathcal{I}}$ and $\langle s, x\rangle \in R^{\mathcal{I}}$ with $\mathrm{Trans}(R)$ and $R \sqsubseteq^{*} S$, then $x \in (\forall R.C)^{\mathcal{I}}$ unless there is some y such that $\langle x, y\rangle \in R^{\mathcal{I}}$ and $y \notin C^{\mathcal{I}}$. In this case, if $\langle s, x\rangle \in R^{\mathcal{I}}$, $\langle x, y\rangle \in R^{\mathcal{I}}$ and $\mathrm{Trans}(R)$, then $\langle s, y\rangle \in R^{\mathcal{I}}$. Hence $\langle s, y\rangle \in S^{\mathcal{I}}$ and $s \notin (\forall S.C)^{\mathcal{I}}$–in contradiction to the assumption. $\mathcal{T}$ therefore satisfies (P7).

- (P12): Assume $\forall T_1, \ldots, T_n.P \in \mathcal{L}(s)$ and $\langle s, v_j\rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$ for all $1 \leq j \leq n$. According to the definition of $\mathcal{E}_{\mathbf{D}}$(T) and δ, there must be n data values $t_1, \ldots, t_n \in \Delta_{\mathbf{D}}$ such that $\langle s, t_j\rangle \in \mathcal{T}_j^{\mathcal{I}}$ and $\delta(v_j) = t_j$ for all $1 \leq j \leq n$. By the definition of $\mathcal{L}$(s), we have $s \in \forall T_1, \ldots, T_n.P$, hence $\langle t_1, \ldots, t_n\rangle \in P^{\mathbf{D}}$. By the definition of $DC^{\mathcal{T}}$, we have $P(v_1, \ldots, v_n) \in DC^{\mathcal{T}}$.

- (P13): Assume $\exists T_1, \ldots, T_n.P \in \mathcal{L}(s)$. By the definition of $\mathcal{L}$(s), we have $s \in \exists T_1, \ldots, T_n.P$, then there must be n data values $t_1, \ldots, t_n \in \Delta_{\mathbf{D}}$ such that $\langle s, t_j\rangle) \in T_j^{\mathcal{I}}$ for all $1 \leq j \leq n$ and $\langle t_1, \ldots, t_n\rangle \in P^{\mathbf{D}}$. By the definition of $\mathbf{S_D}$ and δ, there are n variables $v_1, \ldots, v_n \in \mathbf{S_D}$ s.t. $t_j = \delta(v_j)$ for all $1 \leq j \leq n$. By the definition of $DC^{\mathcal{T}}$, we have $P(v_1, \ldots, v_n) \in DC^{\mathcal{T}}$.

- (P14) and (P15): Assume $\geqslant m T_1, \ldots, T_n.P \in \mathcal{L}(s)$. By the definition of $\mathcal{L}$(s), we have $s \in \geqslant m T_1, \ldots, T_n.P$. This implies that there are $m \cdot n$ data values $t_{11}, \ldots, t_{mn}$ such that $\langle s, t_{ij}\rangle \in T_j^{\mathcal{I}}$ and $(\langle \delta(v_{i1}), \ldots, \delta(v_{in})\rangle, \langle \delta(v_{k1}, \ldots, \delta(v_{kn}\rangle \in \neq^{\mathbf{D}}$ for $1 \leq i < k \leq m, 1 \leq j \leq n$. By the definition of $\mathcal{E}_{\mathbf{D}}$(T), there are $m \cdot n$ variables $v_{11}, \ldots, v_{mn} \in \mathbf{S_D}$, s.t. $t_{ij} = \delta(v_{ij})$ and $\langle s, v_{ij}\rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$ for

$1 \leq i \leq m, 1 \leq j \leq n$. By the definition of $T^{\mathcal{T}}(s, P)$, we have $(\sharp T^{\mathcal{T}}(s, P) \geqslant m)$. Similarly, (P15) is satisfied.

- (P16): Assume $\{\geqslant mT_1, \ldots, T_n.P, \leqslant mT_1, \ldots, T_n.P\} \cap \mathfrak{L}(s) \neq \emptyset$ and $\langle s, v_j \rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$ for $1 \leq j \leq n$. By the definition of $\mathfrak{L}$(s), we have $s \in \geqslant mT_1, \ldots, T_n.P$, or $s \in \leqslant mT_1, \ldots, T_n.P$. According to the definition of $\mathcal{E}_{\mathbf{D}}$(T), there must be n data values $t_1, \ldots, t_n \in \Delta_{\mathbf{D}}$ such that $\langle s, t_j \rangle \in \mathcal{T}_j^{\mathcal{I}}$ and $t_j = \delta(v_j)$ for all $1 \leq j \leq n$. Since either $\langle t_1, \ldots, t_n \rangle \in P^{\mathbf{D}}$ or $\langle t_1, \ldots, t_n \rangle \in (\neg P)^{\mathbf{D}}$, by the definition of $DC^{\mathcal{T}}$, we have either $P(v_1, \ldots, v_n) \in DC^{\mathcal{T}}$ or $\neg P(v_1, \ldots, v_n) \in DC^{\mathcal{T}}$. □

Constructing a $\mathcal{SHOQ}(\mathcal{G})$-Tableau

Now we consider a tableaux algorithm for $\mathcal{SHOQ}(\mathcal{G})$.[4] Generally speaking, given a $\mathcal{G}$-combinable Description Logic $\mathcal{L}$, if we have a tableaux algorithm for $\mathcal{L}$ (we call its completion rules *$\mathcal{L}$-rules*), then the set of the datatype group-related completion rules, or *$\mathcal{G}$-rules* (cf. Figure 6.2 on page 143) and the set of clash conditions about datatype constraints, or *$\mathcal{G}$-clash conditions* (cf. Figure 6.4 on page 144) can be seen as *plug-in*s to the tableaux algorithm of $\mathcal{L}(\mathcal{G})$. I.e., the set of completion rules for $\mathcal{L}(\mathcal{G})$ can be constructed as the union of the sets of $\mathcal{L}$-rules and $\mathcal{G}$-rules, and the set of clash conditions for $\mathcal{L}(\mathcal{G})$ can be constructed as the union of the sets of $\mathcal{L}$-clash conditions (those for $\mathcal{L}$) and $\mathcal{G}$-clash conditions.

In what follows, we will demonstrate that we can extend the known decidable tableaux algorithm for $\mathcal{SHOQ}$, which is provided by Horrocks and Sattler [75], in the above way and give a tableaux algorithm for $\mathcal{SHOQ}(\mathcal{G})$. From Lemma 6.6, an algorithm which constructs a tableau for a $\mathcal{SHOQ}(\mathcal{G})$-concept D can be used as a decision procedure for the satisfiability of D w.r.t. a role box $\mathcal{R}$.

Definition 6.7. (**A Tableaux Algorithm for** $\mathcal{SHOQ}(\mathcal{G})$) Let $\mathcal{R}$ be an RBox, D a $\mathcal{SHOQ}(\mathcal{G})$-concept in NNF, $\mathbf{R}_A^D$ and $\mathbf{R}_{\mathbf{D}}^D$ the sets of abstract roles and concrete roles occurring in D or $\mathcal{R}$, $\mathbf{I}^D$ the set of nominals occurring in D, and $\mathcal{G}$ a datatype group.

A tableaux algorithm works on a *completion forest* F for D w.r.t. $\mathcal{R}$. Labels on nodes and edges are defined as usual, except that there are two kinds of nodes in the completion forest: *abstract nodes* (the normal labelled nodes, by default we simply call them *nodes*) and *concrete nodes* (non-labelled leaves of F). Each (abstract) node

[4] For a brief introduction of tableaux algorithms, see Section 2.2.

x is labelled with a set

$$\mathcal{L}(x) \subseteq \mathsf{cl}(D) \cup \{\uparrow (R, \{o\}) \mid R \in \mathbf{R}_A^D \text{ and } \{o\} \in \mathbf{I}^D\},^{5}$$

and each edge $\langle x, y\rangle$ is labelled with a set of role names $\mathcal{L}(\langle x, y\rangle)$ containing either abstract roles occurring in $\mathbf{R}_A^D$, or concrete roles in $\mathbf{R}_\mathbf{D}^D$: in the first case, y is a node and called an *abstract successor* (or simply *successor*) of x; in the second case, y is a concrete node, and called a *concrete successor* of x. Predecessors, ancestors, and roots are defined as usual. Additionally, we keep track of inequalities between (abstract) nodes of the forest with a symmetric binary relation $\not\approx$. For each $\{\mathsf{o}\} \in \mathbf{I}^D$, there is a *distinguished* node $z_{\{\mathsf{o}\}}$ in $\mathtt{F}$ such that $\{\mathsf{o}\} \in \mathcal{L}(z)$. We use $\uparrow (R, \{o\}) \in \mathcal{L}(x)$ to represent an R labelled edge from x to $z_{\{\mathsf{o}\}}$.

Given a completion forest, a node y is called an *R-successor* of a node x if, for some R' with $R' \boxeq R$, either y is a successor of x and $R' \in \mathcal{L}(\langle x, y\rangle)$, or $\uparrow (R', \{o\}) \in \mathcal{L}(x)$ and $y = z_{\{o\}}$. For an abstract role S and a node x in $\mathtt{F}$, we define $S^{\mathtt{F}}(x, C)$ as

$$S^{\mathtt{F}}(x, C) := \{y \mid y \text{ is an abstract } S\text{-successor of x and } C \in \mathcal{L}(y)\}.$$

Given a completion forest, a concrete node v is called a *concrete T-successor* of a node x if, for some concrete role T' with $T' \boxeq^{*} T$, v is a concrete successor of x and $T' \in \mathcal{L}(\langle x, v\rangle)$. A tuple of concrete nodes $\langle v_1, \ldots, v_n\rangle$ is called a $\langle T_1 \ldots T_n\rangle$-successor of a node x if, for all $1 \leq j \leq n$, v_j is a concrete T_j-successor of x.

We will use a set $DC(x)$ to store the datatype expressions that must hold w.r.t. concrete successors of a node x in F. Each element of $DC(x)$ is either of the form $(\langle v_{k1}, \ldots, v_{kn}\rangle, P)$, or of the form $(\langle v_{i1}, \ldots, v_{in}\rangle, \langle v_{j1}, \ldots, v_{jn}\rangle, \not\approx)$, where $\langle v_{k1}, \ldots, v_{kn}\rangle$, $\langle v_{i1}, \ldots, v_{in}\rangle$, and $\langle v_{j1}, \ldots, v_{jn}\rangle$ are tuples of concrete successors of x, $P \in \mathsf{cl}_{d_n}(D)$ is a (possibly negated) datatype predicate expression, and $\not\approx$ is the value inequality predicate. The tableaux algorithm calls a datatype reasoner as a sub-procedure for the satisfiability of $DC(x)$. We say that $DC(x)$ is satisfiable if the datatype query

$$\begin{array}{c} \bigwedge_{(\langle v_{k1},\ldots,v_{kn}\rangle, P) \in DC(x)} P(v_{k1}, \ldots, v_{kn}) \wedge \\ \bigwedge_{(\langle v_{i1},\ldots,v_{in}\rangle, \langle v_{j1},\ldots,v_{jn}\rangle, \not\approx) \in DC(x)} \not\approx (v_{i1}, \ldots, v_{in}; v_{j1}, \ldots, v_{jn}) \end{array} \tag{6.1}$$

is satisfiable. For n concrete role $T_1, \ldots, T_n$ and a node x in $\mathtt{F}$, we define $T_1, \ldots, T_n^{\mathtt{F}}(x,$

[5] cf. page 134 for the definition of $\mathsf{cl}(D)$.

$\sqcap$-rule: if 1.$C_1 \sqcap C_2 \in \mathcal{L}(x)$, x is not blocked, and
2.$\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$,
then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$

$\sqcup$-rule: if 1.$C_1 \sqcup C_2 \in \mathcal{L}(x)$, x is not blocked, and
2.$\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$,
then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$

$\exists$-rule: if 1.$\exists R.C \in \mathcal{L}(x)$, x is not blocked, and
2.x has no R-successor y with $C \in \mathcal{L}(y)$
then create a new node y with $\mathcal{L}(\langle x, y\rangle) = \{R\}$ and $\mathcal{L}(y) = \{C\}$

$\forall$-rule: if 1.$\forall R.C \in \mathcal{L}(x)$, x is not blocked, and
2.there is an R-successor y of x with $C \notin \mathcal{L}(y)$,
then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$

$\forall_+$-rule: if 1.$\forall S.C \in \mathcal{L}(x)$, x is not blocked, and
2.there is some R with $\mathsf{Trans}(R)$ and $R \sqsubseteq S$,
3.and there is an R-successor y of x with $\forall R.C \notin \mathcal{L}(y)$,
then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\forall R.C\}$

$\geqslant$-rule: if 1.$\geqslant nS.C \in \mathcal{L}(x)$, x is not blocked, and
2.there are no n S-successors $y_1, \ldots, y_n$ of x with $C \in \mathcal{L}(y_i)$ and
3.$y_i \neq y_j$ for $1 \leq i < j \leq n$,
then create n new nodes $y_1, \ldots, y_n$ with $\mathcal{L}(\langle x, y_i\rangle) = \{S\}$,
$\mathcal{L}(y_i) = \{C\}$, and $y_i \neq y_j$ for $1 \leq i < j \leq n$.

$\leqslant$-rule: if 1.$\leqslant nS.C \in \mathcal{L}(x)$, x is not blocked, and
2.x has $n+1$ S-successors $y_1, \ldots, y_{n+1}$ with
$C \in \mathcal{L}(y_i)$ for each $1 \leq i \leq n+1$, and
3.there exist $i \neq j$ s. t. not $y_i \neq y_j$ and
4.if only one of y_i, y_j is distinguished, then it is y_i.
then1. $\mathcal{L}(y_i) \longrightarrow \mathcal{L}(y_i) \cup \mathcal{L}(y_j)$ and add $y \neq y_i$ for each y with
$y \neq y_j$, and
2. if both y_i, y_j are not distinguished,
then $\mathcal{L}(\langle x, y_i\rangle) \longrightarrow \mathcal{L}(\langle x, y_i\rangle) \cup \mathcal{L}(\langle x, y_j\rangle)$
else $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{\uparrow(S, \{\mathsf{o}\}) \mid S \in \mathcal{L}(\langle x, y_j\rangle)\}$
for some $\{\mathsf{o}\} \in \mathcal{L}(y_i)$ and
3. remove y_j and
4. remove all edges leading to y_j from the completion forest

choose-rule: if 1.$\{\geqslant nS.C, \leqslant nS.C\} \cap \mathcal{L}(x) \neq \emptyset$, x is not blocked, and
2.y is an S-successor of x with $\{C, \sim C\} \cap \mathcal{L}(y) = \emptyset$,
then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{E\}$ for some $E \in \{C, \sim C\}$

O-rule: if 1.$\{\mathsf{o}\} \in \mathcal{L}(x)$, x is neither blocked nor distinguished,
2.and not $x \neq z_{\{\mathsf{o}\}}$
then for z distinguished with $\{\mathsf{o}\} \in \mathcal{L}(z)$, do
1. $\mathcal{L}(z) \longrightarrow \mathcal{L}(z) \cup \mathcal{L}(x)$, and
2. if x has a predecessor x',
then $\mathcal{L}(x') \longrightarrow \mathcal{L}(x') \cup \{\uparrow(R, \{\mathsf{o}\}) \mid R \in \mathcal{L}(\langle x', x\rangle)\}$,
3. add $y \neq z$ for each y with $y \neq x$, and remove x and
all edges leading to x from the completion forest.

Figure 6.1: The $\mathcal{SHOQ}$-rules

P) as

$$T_1 \ldots T_n^{\mathtt{F}}(x, P) := \{\langle v_1, \ldots, v_n\rangle \mid \langle v_1, \ldots, v_n\rangle \text{ is a } \langle T_1 \ldots T_n\rangle\text{-successor of } x \text{ and } (\langle v_1, \ldots, v_n\rangle, P) \in DC(x)\}.$$

$\forall_P$-rule: if 1.$\forall T_1, \ldots, T_n.P \in \mathcal{L}(x)$, x is not blocked, and
2.there is a $\langle T_1 \ldots T_n\rangle$-successor $\langle v_1, \ldots, v_n\rangle$ of x
with $(\langle v_1, \ldots, v_n\rangle, P) \notin DC(x)$,
then $DC(x) \longrightarrow DC(x) \cup \{(\langle v_1, \ldots, v_n\rangle, P)\}$.

$\exists_P$-rule: if 1.$\exists T_1, \ldots, T_n.P \in \mathcal{L}(x)$, x is not blocked, and
2.x has no $\langle T_1 \ldots T_n\rangle$-successor $\langle v_1, \ldots, v_n\rangle$ with
$(\langle v_1, \ldots, v_n\rangle, P) \in DC(x)$,
then 1. create a $\langle T_1 \ldots T_n\rangle$-successor $\langle v_1, \ldots, v_n\rangle$ of x with
$\mathcal{L}(\langle x, v_j\rangle) = \{T_j\}$ for all $1 \leq j \leq n$, and
2. $DC(x) \longrightarrow DC(x) \cup \{(\langle v_1, \ldots, v_n\rangle, P)\}$.

$\geqslant_P$-rule: if 1.$\geqslant m T_1, \ldots, T_n.P \in \mathcal{L}(x)$, x is not blocked, and
2.there are no m $\langle T_1 \ldots T_n\rangle$-successors $\langle v_{11}, \ldots, v_{1n}\rangle, \ldots,$
$\langle v_{m1}, \ldots, v_{mn}\rangle$ of x with $(\langle v_{i1}, \ldots, v_{in}\rangle, P) \in DC(x)$ and
$(\langle v_{i1}, \ldots, v_{in}\rangle, \langle v_{j1}, \ldots, v_{jn}\rangle, \neq) \in DC(x)$ for $1 \leq i < j \leq m$,
then 1. create m new $\langle T_1 \ldots T_n\rangle$-successors $\langle v_{11}, \ldots, v_{1n}\rangle, \ldots,$
$\langle v_{m1}, \ldots, v_{mn}\rangle$ of x with $\mathcal{L}(\langle x, v_{ij}\rangle) = \{T_j\}$,
for all $1 \leq i \leq m, 1 \leq j \leq n$, and
2. $DC(x) \longrightarrow DC(x) \cup \{(\langle v_{i1}, \ldots, v_{in}\rangle, P)\}$ for $1 \leq i \leq m$, and
3. $DC(x) \longrightarrow DC(x) \cup \{(\langle v_{i1}, \ldots, v_{in}\rangle, \langle v_{j1}, \ldots, v_{jn}\rangle, \neq)\}$
for all $1 \leq i < j \leq m$.

$\leqslant_P$-rule: if 1.$\leqslant m T_1, \ldots, T_n.P \in \mathcal{L}(x)$, x is not blocked, and
2.x has $m+1$ $\langle T_1 \ldots T_n\rangle$-successors $\langle v_{11}, \ldots, v_{1n}\rangle, \ldots,$
$\langle v_{m+1,1}, \ldots, v_{m+1,n}\rangle$ of x, with $(\langle v_{i1}, \ldots, v_{in}\rangle, P) \in DC(x)$
for $1 \leq i \leq m+1$, and
3.among them, there exist two different $\langle T_1 \ldots T_n\rangle$-successors
$\langle v_{i1}, \ldots, v_{in}\rangle$ and $\langle v_{j1}, \ldots, v_{jn}\rangle$ $(i \neq j)$, s.t.
$(\langle v_{i1}, \ldots, v_{in}\rangle, \langle v_{j1}, \ldots, v_{jn}\rangle, \neq) \notin DC(x)$,
then for each pair v_{ik}, v_{jk}, where v_{ik} and v_{jk} are not the same
concrete T_k-successor of x,
do 1. $\mathcal{L}(\langle x, v_{ik}\rangle) \longrightarrow \mathcal{L}(\langle x, v_{ik}\rangle) \cup \mathcal{L}(\langle x, v_{jk}\rangle)$ and
2. replace v_{jk} with v_{ik} in $DC(x)$, remove v_{jk}, and
3. remove all edges leading to v_{jk} from x.

$choose_P$-rule: if 1.$\{\leqslant m T_1, \ldots, T_n.P, \geqslant m T_1, \ldots, T_n.P\} \cap \mathcal{L}(x) \neq \emptyset$, x is not blocked,
2.$\langle v_1, \ldots, v_n\rangle$ is a $\langle T_1 \ldots T_n\rangle$-successor of x,
with $(\langle v_1, \ldots, v_n\rangle, P) \notin DC(x)$ and $(\langle v_1, \ldots, v_n\rangle, \neg P) \notin DC(x)$
then either $DC(x) \longrightarrow DC(x) \cup \{(\langle v_1, \ldots, v_n\rangle, P)\}$,
or $DC(x) \longrightarrow DC(x) \cup \{(\langle v_1, \ldots, v_n\rangle, \neg P)\}$.

Figure 6.2: The $\mathcal{G}$-rules

A node x is *directly blocked* if none of its ancestors are blocked, and it has an ancestor x' that is not distinguished such that $\mathcal{L}(x) \subseteq \mathcal{L}(x')$. We say x' blocks x. A node is *blocked* if it is directly blocked or if its predecessor is blocked.

If $\{\{o_1\}, \ldots, \{o_\ell\}\} = \mathbf{I}^D$, the algorithm initialises the completion forest F to contain $l+1$ root nodes $x_0, z_{\{o_1\}}, \ldots, z_{\{o_l\}}$ with $\mathcal{L}(x_0) = \{D\}$ and $\mathcal{L}(z_{\{o_i\}}) = \{\{o_i\}\}$. The inequality relation $\dot{\neq}$ is initialised with the empty relation and the $DC(x)$ of any

A node x of a completion forest F contains a *clash* if (at least) one of the following conditions holds:

(1) $\bot \in \mathcal{L}(x)$;

(2) for some $\mathsf{A} \in \mathbf{C}$, $\{\mathsf{A}, \neg\mathsf{A}\} \subseteq \mathcal{L}(x)$;

(3) for some abstract role S, $\leqslant nS.C \in \mathcal{L}(x)$ and there are $n+1$ S-successors $y_1, \ldots, y_{n+1}$ of x with $C \in \mathcal{L}(y_i)$ for each $1 \leq i \leq n+1$ and $y_i \neq y_j$ for each $1 \leq i < j \leq n+1$;

(4) for some $\{o\} \in \mathcal{L}(x), x \doteq z_{\{o\}}$.

Figure 6.3: The $\mathcal{SHOQ}$-clash conditions

(abstract) node x is initialised to the empty set. F is then expanded by repeatedly applying the *completion rules*, listed in Figure 6.1 and Figure 6.2, until F is complete, or some $\mathcal{L}(x)$ contains a clash (see Figure 6.3 and Figure 6.4 for clash conditions for $\mathcal{SHOQ}(\mathcal{G})$).

A node x of a completion forest F contains a *clash* if (at least) one of the following conditions holds:

($\mathcal{G}$1) for some concrete roles $T_1, \ldots, T_n$, $\leqslant mT_1, \ldots, T_n.P \in \mathcal{L}(x)$ and there are $m+1$ $\langle T_1 \ldots T_n \rangle$-successors $\langle v_{11}, \ldots, v_{1n} \rangle, \ldots, \langle v_{m+1,1}, \ldots, v_{m+1,n} \rangle$ of x with $(\langle v_{i1}, \ldots, v_{in} \rangle, P) \in DC(x)$ for each $1 \leq i \leq m+1$, and $(\langle v_{i1}, \ldots, v_{in} \rangle, \langle v_{j1}, \ldots, v_{jn} \rangle, \neq) \in DC(x)$ for each $1 \leq i < j \leq m+1$;

($\mathcal{G}$2) for some abstract node x, $DC(x)$ is not satisfiable;

Figure 6.4: The $\mathcal{G}$-clash conditions

If the completion rules can be applied in such a way that they yield a complete, clash-free completion forest, then the algorithm returns "D is *satisfiable* w.r.t. $\mathcal{R}$", and "D is *unsatisfiable* w.r.t. $\mathcal{R}$" otherwise. ◇

There are some remarks about $DC(x)$ in the algorithm:

1. $DC(x)$ represents a local datatype query for the node x; it is the 'interface' between the tableaux algorithm and datatype reasoners.

2. For any $(\langle v_1, \ldots, v_n \rangle, P) \in DC(x)$, the algorithm is not affected by the structure of the datatype expression P. I.e., the kind of datatype expressions and the kind of supported predicates provided by datatype reasoners are independent of the tableaux algorithm.

3. The merging of concrete nodes (variables) is handled by the $\leqslant_P$-rule, instead of by datatype reasoners (through the value equality predicate $=$). Therefore, the form of datatype queries (5.3) is simplified to

$$\mathscr{Q}' := \mathscr{C}_E \wedge \bigwedge_{j=1}^{k} \neq (\vec{v}_{(j,1)}; \vec{v}_{(j,2)}), \tag{6.2}$$

where $\mathscr{C}_E$ is a datatype expression conjunction of the form (5.14), and $\neq$ is the value inequality predicate. According to Lemma 5.18, if $\mathcal{G}$ is conforming, a datatype query of the form (6.2) is decidable.

Sound and Complete

The soundness and completeness of the algorithm will be demonstrated by that, for a $\mathcal{SHOQ}(\mathcal{G})$-concept D, it always terminates and that it returns *satisfiable* if and only if D is satisfiable.

Lemma 6.8. (Termination) *For each $\mathcal{SHOQ}(\mathcal{G})$-concept D and RBox $\mathcal{R}$, the completion algorithm terminates.*

Proof: Let $h = |\mathtt{cl}(D)|$, $k = |\mathbf{R}_A^D|$, m_{max} the maximal number in atleast number restrictions or datatype atleast restrictions in D and $\ell = |\mathbf{I}^D|$. Note that datatype checkers will be used to test the satisfiability of $DC(x)$, and that this is known to be decidable (cf. Lemma 5.18 on page 119). Termination is a consequence of the following properties of the expansion rules:

1. All the rules except the $\leqslant$-, $\leqslant_P$- and **O**-rules strictly extend the completion forest by extending node labels or adding nodes, while removing neither nodes nor elements from node labels.

2. New nodes are only generated by the $\exists$-, $\exists_P$-, $\geqslant$-rule or the $\geqslant_P$-rule as successors of an abstract node x for concepts of the form $\exists R.C$, $\exists T_1, \ldots, T_n.P$, $\geqslant nS.C$ and $\geqslant mT_1, \ldots, T_n.P$ in $\mathcal{L}(x)$. For x, each of these concepts can trigger the generation of successors at most once—even if the node(s) generated are later removed by either the $\leqslant$-, $\leqslant_P$- or the **O**-rule.

   - For the $\exists$-rule, if a successor y of x was generated for a concept $\exists S.C \in \mathcal{L}(x)$ and later y is removed by an application of the $\leqslant$-rule or the **O**-rule, then there will always be some S-successor z of x such that $C \in \mathcal{L}(z)$, and hence the $\exists$-rule can not be applied again to x and $\exists S.C$.

- For the $\geqslant$-rule, if $y_1, \ldots, y_n$ were generated by an application of the $\geqslant$-rule for a concept $(\geqslant nS.C)$, then $y_i \neq y_j$ holds for all $1 \leq i < j \leq n$. This implies that there will always be n S-successors $y'_1, \ldots, y'_n$ of x with $C \in \mathcal{L}(y'_i)$ and $y'_i \neq y'_j$ holds for all $1 \leq i < j \leq n$, since neither the $\leqslant$-rule nor the **O**-rule can ever merge two nodes y'_i, y'_j (because $y'_i \neq y'_j$), and, whenever an application of the $\leqslant$-rule or the **O**-rule removes y'_i, there will be an S-successor z of x that "inherits" all the inequalities from y'_i.
- For the $\exists_P$-rule, if a concrete $\langle T_1, \ldots, T_n \rangle$-successor $\langle v_1, \ldots, v_n \rangle$ of x was generated by an application of the $\exists_P$-rule for a concept $\exists T_1, \ldots, T_n.P$, then the $\leqslant_P$-rule can only remove some v_i $(1 \leq i \leq n)$ by merging $\mathcal{L}(\langle x, v_i \rangle)$ into $\mathcal{L}(\langle x, v'_i \rangle)$, where v' is some other concrete T_i-successor of x, and replacing v_i with v'_i in $DC(x)$ such that $\sharp T^{\mathtt{F}}(x, P) \neq \emptyset$.
- For the $\geqslant_P$-rule, if $\langle T_1 \ldots T_n \rangle$-successors $\langle v_{11}, \ldots, v_{1n} \rangle$, …, $\langle v_{m1}, \ldots, v_{mn} \rangle$ were generated by an application of the $\geqslant_P$-rule for a concept $(\leqslant mT_1, \ldots, T_n.P)$, then $(\langle v_{i1}, \ldots, v_{in} \rangle, \langle v_{j1}, \ldots, v_{jn} \rangle, \neq) \in DC(x)$ holds for all $1 \leq i < j \leq m$. This implies that any $\langle v_{i1}, \ldots, v_{in} \rangle$, $\langle v_{j1}, \ldots, v_{jn} \rangle$ of these $\langle T_1 \ldots T_n \rangle$-successors of x can never be merged by an application of the $\leqslant_P$-rule. Whenever an application of the $\leqslant_P$-rule merges $\langle v_{i1}, \ldots, v_{in} \rangle$ and $\langle v_{k1}, \ldots, v_{kn} \rangle$, and removes some v_{ig}, where $1 \leq g \leq n$, there will be a concrete T_g-successor v_{kg} of x within $\langle v_{k1}, \ldots, v_{kn} \rangle$ which 'inherits' $\mathcal{L}(\langle x, v_{ig} \rangle)$, and the $\langle T_1 \ldots T_n \rangle$-successor $\langle v_{k1}, \ldots, v_{kn} \rangle$ will 'inherit' all the datatype constraints (including all the inequalities) in $DC(x)$ about $\langle v_{i1}, \ldots, v_{in} \rangle$.

  Since $\mathsf{cl}(D)$ contains a total of at most h $\exists S.C$, $(\geqslant nS.C)$, $\exists T_1, \ldots, T_n.P$ and $(\geqslant mT_1, \ldots, T_n.P)$ concepts, the out-degree of the forest is bounded by $h \cdot m_{max}$.

3. Nodes are labeled with subsets of $\mathtt{cl}(D) \cup \{\uparrow(R, \{\mathtt{o}\}) \mid R \in \mathbf{R}_A^D \text{ and } \{\mathtt{o}\} \in \mathbf{I}^D\}$. Since concrete nodes have no labels and are always leaves, there are at most $2^{h+k \cdot \ell}$ different abstract node labellings. Therefore, if a path p is of length at least $2^{h+k \cdot \ell}$, then, from the blocking condition above, there are two nodes x, y on p such that x is directly blocked by y. Hence paths are of length at most $2^{h+k \cdot \ell}$. □

In the following proof of soundness, we will show that we can construct a tableau $T = (\mathbf{S}_A, \mathbf{S}_\mathbf{D}, \mathcal{L}, DC^{\mathcal{T}}, \mathcal{E}_A, \mathcal{E}_\mathbf{D})$ from a complete and clash-free completion forest $\mathtt{F}$.

Intuitively, it works as follows: An individual in $\mathbf{S}_A$ corresponds to a *path* in F from the root node to some (abstract) node that is not blocked. As these paths may be *cyclic*, tableaux can be infinite. Due to qualifying number restrictions, we must distinguish different nodes that are blocked by the same node. A variable in $\mathbf{S_D}$ corresponds to a concrete node in F, which is a concrete successor of the *tail* of a path, and datatype constraints in $DC^{\mathcal{T}}$ correspond to the elements of the DC set in the completion forest F.

Lemma 6.9. (Soundness) *If the expansion rules can be applied to a $\mathcal{SHOQ}(\mathcal{G})$-concept D in NNF and an RBox $\mathcal{R}$ such that they yield a complete and clash-free completion forest, then D has a tableau w.r.t. $\mathcal{R}$.*

Proof: Let F be the complete and clash-free completion forest constructed by the completion algorithm for D. A path is sequence of pairs of (abstract) nodes of F of the form $p = [(x_0, x'_0), \ldots, (x_n, x'_n)]$ (we disregard concrete nodes). For such a path, we define $\mathtt{Tail}(p) = x_n$ and $\mathtt{Tail}'(p) = x'_n$. With $[p|(x_{n+1}, x'_{n+1})]$ we denote the path $[(x_0, x'_0), \ldots, (x_n, x'_n), (x_{n+1}, x'_{n+1})]$. The set $\mathtt{Paths}(\mathtt{F})$ is defined inductively as follows:

1. For a root node x_0 in F, $[(x_0, x_0)] \in \mathtt{Paths}(\mathtt{F})$, and
2. For a path $p \in \mathtt{Paths}(\mathtt{F})$ and a node z in F:
   (a) if z is a successor[6] of $\mathtt{Tail}(p)$ and z is not blocked, then $[p|(z, z)] \in \mathtt{Paths}(\mathtt{F})$, or
   (b) if, for some node y in F, y is a successor of $\mathtt{Tail}(p)$ and z blocks y, then $[p|(z, y)] \in \mathtt{Paths}(\mathtt{F})$.

Please note that, due to the construction of Paths, for $p \in \mathtt{Paths}(\mathtt{F})$ with $p = [p'|(x, x')]$, x' is blocked iff $x \neq x'$.

Now we can define a tableau $\mathcal{T} = (\mathbf{S}_A, \mathbf{S_D}, \mathcal{L}, DC^{\mathcal{T}}, \mathcal{E}_A, \mathcal{E}_{\mathbf{D}})$ with:

$$
\begin{aligned}
\mathbf{S}_A &= \mathtt{Paths}(\mathtt{F}) \\
\mathbf{S_D} &= \{v \mid v \text{ is a concrete node in F}\} \\
\mathcal{L}(p) &= \mathcal{L}(\mathtt{Tail}(p)) \\
DC^{\mathcal{T}} &= \{P(v_1, \ldots, v_n) \mid \exists p \in \mathbf{S}_A.(\langle v_1, \ldots, v_n\rangle, P) \in DC(\mathtt{Tail}(p))\} \\
\mathcal{E}_A(R) &= \{\langle p, q\rangle \in \mathbf{S}_A \times \mathbf{S}_A \mid q = [p|(x, x')] \text{ and } x' \text{ is an } R\text{-successor of } \mathtt{Tail}(p)\} \\
\mathcal{E}_{\mathbf{D}}(T) &= \{\langle p, v\rangle \in \mathbf{S}_A \times \mathbf{S_D} \mid p \in \mathbf{S}_A \text{ and } v \text{ is a concrete } T\text{-successor of } \mathtt{Tail}(p)\}
\end{aligned}
$$

[6] By default, we simply call abstract successors *successors*; see Definition 6.7 on page 140.

Claim: $\mathcal{T}$ is a tableau for D w.r.t. $\mathcal{R}$.

We have to show that $\mathcal{T}$ satisfies all the properties from Definition 6.5.

- (P0): because F is clash-free, there exists a datatype interpretation $\mathcal{I}_{\mathbf{D}} = (\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ and a mapping $\delta : \mathbf{S}_{\mathbf{D}} \rightarrow \Delta_{\mathbf{D}}$ s.t. for each $p \in \mathbf{S}_A$ and each $(\langle v_1, \ldots, v_n \rangle, P) \in DC(\mathtt{Tail}(p))$. Thus, according to the definition of $DC^{\mathcal{T}}$, for each $P(v_1, \ldots, v_n) \in DC^{\mathcal{T}}$, there exists $t_j = \delta(v_j)$ for all $1 \leq j \leq n$ and $\langle t_1, \ldots, t_n \rangle \in P^{\mathbf{D}}$.

- (P1): If $C \in \mathcal{L}(p)$, then $\neg C \notin \mathcal{L}(p)$; otherwise F is not clash-free.

- (P2) and (P3) hold because $\mathtt{Tail}(p)$ is not blocked and F is complete.

- (P4) is satisfied due to the definition of R-successor that takes into account the role hierarchy $\sqsubseteq^*$.

- (P5) : Assume $\forall S.C \in \mathcal{L}(p)$ and $\langle p, q \rangle \in \mathcal{E}_A(S)$. Let $q = [p|(x, x')]$, x' is an S-successor of $\mathtt{Tail}(p)$ and thus $C \in \mathcal{L}(x')$ must hold (otherwise the $\forall$-rule is still applicable). Since $\mathcal{L}(x') \subseteq \mathcal{L}(x) = \mathcal{L}(q)$,[7] we have $C \in \mathcal{L}(q)$.

- (P6): Assume $\exists S.C \in \mathcal{L}(p)$. Define $x := \mathtt{Tail}(p)$. In F there must be an S-successor y' of x with $C \in \mathcal{L}(y')$, otherwise the $\exists$-rule is still applicable. Let $q := [p|(y, y')] \in \mathbf{S}_A$, since $\mathcal{L}(y') \subseteq \mathcal{L}(y) = \mathcal{L}(q)$, we have $C \in \mathcal{L}(q)$.

- (P7): Assume $\forall S.C \in \mathcal{L}(p)$, $\langle p, q \rangle \in \mathcal{E}_A(R)$ for some $R \sqsubseteq^* S$ with $\mathrm{Trans}(R)$. Let $q := [p|(x, x')]$, then x' is an R-successor of $\mathtt{Tail}(p)$ and thus $\forall R.C \in \mathcal{L}(x')$ must hold, otherwise the $\forall_+$-rule would be applicable. Since $\mathcal{L}(x') \subseteq \mathcal{L}(x) = \mathcal{L}(q)$, we have $\forall R.C \in \mathcal{L}(q)$.

- (P8): Assume $(\geqslant n S.C) \in \mathcal{L}(P)$. This implies that there exist n individuals $y_1, \ldots, y_n$ in F such that each y_i is an S-successor of $\mathtt{Tail}(p)$ and $C \in \mathcal{L}(y_i)$. We claim that for each of these individuals, there is a path q_i such that $\langle p, q_i \rangle \in \mathcal{E}_A(S), C \in \mathcal{L}(q_i)$, and $q_i \neq q_j$ for all $1 \leq i < j \leq n$. Obviously, this implies $\sharp S^{\mathcal{T}}(p, C) \geqslant n$. For each y_i there are two possibilities:

  - y_i is an S-successor of x and y_i is not blocked in F. Then $q_i = [p|(y_i, y_i)]$ is a path with the desired properties.

[7] If x' is not blocked, $\mathcal{L}(x') = \mathcal{L}(x)$; otherwise $\mathcal{L}(x') \subseteq \mathcal{L}(x)$.

– y_i is an S-successor of x and y_i is blocked in $\mathtt{F}$ by some node z. Then $q_i = [p|(z, y_i)]$ is a path with the desired properties. Since the same z may block several of the y_is, it is indeed necessary to include y_i explicitly in the path to make them distinct.

- (P9): Assume that there is some $p \in \mathbf{S}_A$ with $(\leqslant nS.C) \in \mathcal{L}(p)$ and $S^{\mathcal{I}}(p, C) > n$. We will show that this implies $S^{\mathtt{F}}(\mathtt{Tail}(p), C) > n$ which is a contradiction to either the clash-freeness or completeness of $\mathtt{F}$. Define $x := \mathtt{Tail}(p)$ and $P := S^{\mathcal{I}}(p, C)$. Due to the assumption, we have $\sharp P > n$ and P contains paths of the form $q = [p|(y, y')]$.

  We claim that the function $\mathtt{Tail}'$ is injective on P. If we assume that there are two paths $q_1, q_2 \in P$ with $q_1 \neq q_2$ and $\mathtt{Tail}'(q_1) = \mathtt{Tail}'(q_2) = y'$, then this implies that q_1 is of the form $q_1 = [p|(y_1, y')]$ and q_2 is of the form $q_2 = [p|(y_2, y')]$ with $y_1 \neq y_2$. If y' is not blocked in $\mathtt{F}$, then $y_1 = y' = y_2$ holds, contradicting $y_1 \neq y_2$. If y' is blocked in $\mathtt{F}$, then both y_1 and y_2 block y', which implies $y_1 = y_2$, again a contradiction.

  Since $\mathtt{Tail}'$ is injective on P, it holds that $\sharp P = \sharp \mathtt{Tail}'(P)$. Also for each $y' \in \mathtt{Tail}'(P)$, y' is an S-successor of x and $C \in \mathcal{L}(y')$. This implies $S^{\mathtt{F}}(x, C) > n$, i.e., $S^{\mathtt{F}}(\mathtt{Tail}(p), C) > n$.

- (P10): Assume $\{\geqslant nS.C, \leqslant nS.C\} \cap \mathcal{L}(p) \neq \emptyset$, $\langle p, q\rangle \in \mathcal{E}_A(S)$. Let $q = [p|(x, x')]$, x' is an S-successor of $\mathtt{Tail}(p)$ and thus $\{C, \sim C\} \cap \mathcal{L}(x') \neq \emptyset$ must hold (otherwise the $choose$-rule is still applicable). Since $\mathcal{L}(x') \subseteq \mathcal{L}(x) = \mathcal{L}(q)$, we have $\{C, \sim C\} \cap \mathcal{L}(q) \neq \emptyset$.

- (P11): Assume $\{\mathsf{o}\} \in \mathcal{L}(p) \cap \mathcal{L}(q)$. $\mathtt{Tail}(p) = \mathtt{Tail}(q) = z_{\{\mathsf{O}\}}$ must hold, otherwise the **O**-rule is still applicable. Since distinguished nodes can never be blocked, we have $p = q$.

- (P12): Assume $\forall T_1, \ldots, T_n.P \in \mathcal{L}(p)$ and $\langle p, v_j\rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$. This implies that v_j is a concrete T_j-successor of $\mathtt{Tail}(p)$, for all $1 \leq j \leq n$, and that $\langle v_1, \ldots, v_n\rangle$ is a $\langle T_1 \ldots T_n\rangle$-successor of $\mathtt{Tail}(p)$. Hence $(\langle v_1, \ldots, v_n\rangle, P) \in DC(\mathtt{Tail}(p))$ must hold, otherwise the $\forall_P$-rule is still applicable. Due to the construction of $DC^{\mathcal{I}}$, we have $P(v_1, \ldots, v_n) \in DC^{\mathcal{I}}$.

- (P13): Assume $\exists T_1, \ldots, T_n.P \in \mathcal{L}(p)$. In $\mathtt{F}$ there must be a $\langle T_1 \ldots T_n\rangle$-successor $\langle v_1, \ldots, v_n\rangle$ of $\mathtt{Tail}(p)$ s.t. $(\langle v_1, \ldots, v_n\rangle, P) \in DC(\mathtt{Tail}(p))$, otherwise the

$\exists_P$-rule is still applicable. Due to the construction of $DC^{\mathcal{I}}$, we have $P(v_1, \ldots, v_n) \in DC^{\mathcal{I}}$.

- (P14): Assume $\geqslant m T_1, \ldots, T_n.P \in \mathcal{L}(p)$. This implies $\geqslant m T_1, \ldots, T_n.P \in \mathcal{L}(\mathtt{Tail}(p))$, and hence there are m $\langle T_1 \ldots T_n \rangle$-successors $\langle v_{11}, \ldots, v_{1n} \rangle$, ..., $\langle v_{m1}, \ldots, v_{mn} \rangle$ of $\mathtt{Tail}(p)$ in F such that $(\langle v_{i1}, \ldots, v_{in} \rangle, P) \in DC(\mathtt{Tail}(p))$ and $(\langle v_{i1}, \ldots, v_{in} \rangle, \langle v_{j1}, \ldots, v_{jn} \rangle, \neq) \in DC(\mathtt{Tail}(p))$ for all $1 \leq i < j \leq m$ must hold (otherwise the $\geqslant_P$-rule is still applicable). Due to the completeness of F, we have $\langle \delta(v_{i1}), \ldots, \delta(v_{in}) \rangle \in P^{\mathbf{D}}$ and $(\langle \delta(v_{i1}), \ldots, \delta(v_{in}) \rangle, \langle \delta(v_{j1}), \ldots, \delta(v_{jn}) \rangle) \in \neq^{\mathbf{D}}$ for all $1 \leq i < j \leq m$. Hence $T^{\mathcal{I}}(p, P) \geqslant m$.

- (P15): Assume that there is some $p \in \mathbf{S}_A$ such that $\leqslant m T_1, \ldots, T_n.P \in \mathcal{L}(p)$ and $T^{\mathcal{I}}(p, P) > m$. Define $x := \mathtt{Tail}(p)$. Due to the assumption, we have at least $m+1$ $\langle T_1 \ldots T_n \rangle$-successors $\langle v_{11}, \ldots, v_{1n} \rangle$, ..., $\langle v_{m,1}, \ldots, v_{m,n} \rangle$ of x, such that $\langle v_{i1}, \ldots, v_{in} \rangle \in P^{\mathbf{D}}$ and $\langle \delta(v_i), \delta(v_j) \rangle \in \neq^{\mathbf{D}}$ for all $1 \leq i < j \leq m$. This implies that we have $(\langle v_i, v_j \rangle, \neq) \in DC(x)$ for all $1 \leq i < j \leq m+1$, which means there is a clash of form ($\mathcal{G}1$), contradicting the clash-freeness of F.

- (P16): Assume $\{\leqslant m T_1, \ldots, T_n.P, \geqslant m T_1, \ldots, T_n.P\} \cap \mathcal{L}(p) \neq \emptyset$, and there exists a $\langle T_1 \ldots T_n \rangle$-successor $\langle v_1, \ldots, v_n \rangle$ of $\mathtt{Tail}(p)$, such that $\langle p, v_j \rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$ for all $1 \leq j \leq n$. Thus $(\langle v_1, \ldots, v_n \rangle, P) \in DC(x)$ or $(\langle v_1, \ldots, v_n \rangle, \neg P) \in DC(x)$ must hold (otherwise the $choose_P$-rule is still applicable). Due to the definition of $DC^{\mathcal{I}}$, we have either $P(v_1, \ldots, v_n) \in DC^{\mathcal{I}}$ or $\neg P(v_1, \ldots, v_n) \in DC^{\mathcal{I}}$. □

Lemma 6.10. (Completeness) *If a $\mathcal{SHOQ}(\mathcal{G})$-concept D in NNF has a tableau w.r.t. an RBox $\mathcal{R}$, then the expansion rules can be applied to D such that they yield a complete, clash-free completion forest w.r.t. $\mathcal{R}$.*

Proof: Let $\mathcal{I} = (\mathbf{S}_A, \mathbf{S}_{\mathbf{D}}, \mathcal{L}, DC^{\mathcal{I}}, \mathcal{E}_A, \mathcal{E}_{\mathbf{D}})$ be a tableau for D w.r.t. an RBox $\mathcal{R}$. Using $\mathcal{I}$, we trigger the application of the completion rules such that they yield a completion forest F that is both complete and clash-free.

We start with F consisting of $l+1$ root nodes $x_0, z_{\{o_1\}}, \ldots, z_{\{o_l\}}$ with $\mathcal{L}(x_0) = \{D\}$, $\neq = \emptyset$, and $DC(x) = \emptyset$. Since $\mathcal{I}$ is a tableau, there is some $s_0 \in \mathbf{S}_A$ with $D \in \mathcal{L}(s_0)$, for all abstract nodes x. Due to (P0), there exists a datatype interpretation $\mathcal{I}_{\mathbf{D}} = (\Delta_{\mathbf{D}}, \cdot^{\mathbf{D}})$ and a mapping δ that maps the variables in $\mathbf{S}_{\mathbf{D}}$ to data values in $\Delta_{\mathbf{D}}$.

When applying the completion rules to F, the application of the non-deterministic rules are guided by the labelling in the tableau $\mathcal{I}$. To this purpose, we define a mapping

π which maps the abstract nodes of F to elements of $\mathbf{S}_A$ and maps concrete nodes of F to elements of $\mathbf{S_D}$. We steer the non-deterministic rules such that $\mathcal{L}(x) \subseteq \mathcal{L}(\pi(x))$ holds for all abstract nodes x of the completion forest. More precisely, we define π inductively as follows, for each $x, y, v, v_1, \ldots, v_n, w_1, \ldots, w_n$ in F:

$$\left.\begin{array}{l}
\mathcal{L}(x) \subseteq \mathcal{L}(\pi(x)) \\
\langle \pi(x), \pi(y) \rangle \in \mathcal{E}_A(S) \\
\qquad \text{if } y \text{ is an } S\text{-successor of } x, \\
\langle \pi(x), \pi(v_j) \rangle \in \mathcal{E}_{\mathbf{D}}(T_j) \text{ (for all } 1 \leq j \leq n) \\
\qquad \text{if } \langle v_1, \ldots, v_n \rangle \text{ is a } \langle T_1 \ldots T_n \rangle\text{-successor of } x \\
\pi(x) \neq \pi(y) \\
\qquad \text{if } x \text{ and } y \text{ are abstract nodes and } x \neq y \\
(\langle v_1, \ldots, v_n \rangle, P) \in DC(x) \\
\qquad \text{if } v_1, \ldots, v_n \text{ are a concrete successors of } x \text{ and } P(\pi(v_1), \ldots, \pi(v_n)) \in DC^{\mathcal{T}} \\
(\langle \delta(\pi(v_1)), \ldots, \delta(\pi(v_n)) \rangle, \langle \delta(\pi(v_1)), \ldots, \delta(\pi(v_n)) \rangle) \in \neq_P^{\mathbf{D}} \\
\qquad \text{if } v_1, \ldots, v_n, w_1, \ldots, w_n \text{ are concrete successors of } x \text{ and} \\
\qquad (\langle v_1, \ldots, v_n \rangle, \langle w_1, \ldots, w_n \rangle, \neq_P) \in DC(x)
\end{array}\right\} (*)$$

Claim: Let F be a completion forest and π a function that satisfies $(*)$. If a rule is applicable to F then the rule is applicable to F in a way that yields a completion forest F' and a function π' that satisfy $(*)$.

Let F to be a completion forest and π a function that satisfies $(*)$. We have to consider the various rules.

- The $\sqcap$-rule: If $C_1 \sqcap C_2 \in \mathcal{L}(x)$, then $C_1 \sqcap C_2 \in \mathcal{L}(\pi(x))$. This implies $C_1 \in \mathcal{L}(\pi(x))$ and $C_2 \in \mathcal{L}(\pi(x))$ due to (P2) from Definition 6.5, and hence the rule can be applied without violating $(*)$.

- The $\sqcup$-rule: If $C_1 \sqcup C_2 \in \mathcal{L}(x)$, then $C_1 \sqcup C_2 \in \mathcal{L}(\pi(x))$. Since $\mathcal{T}$ is a tableau, (P3) from Definition 6.5 implies $\{C_1, C_2\} \cap \mathcal{L}(\pi(x)) \neq \emptyset$. Hence the $\sqcup$-rule can add a concept $E \in \{C_1, C_2\}$ to $\mathcal{L}(x)$ such that $\mathcal{L}(x) \subseteq \mathcal{L}(\pi(x))$ holds.

- The $\exists$-rule: If $\exists S.C \in \mathcal{L}(x)$, then $\exists S.C \in \mathcal{L}(\pi(x))$ and, since $\mathcal{T}$ is a tableau, (P6) from Definition 6.5 implies that there is an element $t \in \mathbf{S}_A$ such that $\langle \pi(x), t \rangle \in \mathcal{E}_A(S)$ and $C \in \mathcal{L}(t)$. The application of the $\exists$-rule generates a new abstract node y with $\mathcal{L}(\langle x, y \rangle) = \{S\}$ and $\mathcal{L}(y) = \{C\}$. Hence we set $\pi' := \pi[y \mapsto t]$ which yields a function that satisfies $(*)$ for the modified forest.

- The $\forall$-rule: If $\forall S.C \in \mathcal{L}(x)$, then $\forall S.C \in \mathcal{L}(\pi(x))$, and if y is an S-successor

of x, then also $\langle \pi(x), \pi(y) \rangle \in \mathcal{E}_A(S)$ due to $(*)$. Since $\mathcal{T}$ is a tableau, (P5) from Definition 6.5 implies $C \in \mathcal{L}(\pi(y))$ and hence the $\forall$-rule can be applied without violating $(*)$.

- The $\forall_+$-rule: If $\forall S.C \in \mathcal{L}(x)$, then $\forall S.C \in \mathcal{L}(\pi(x))$, and, if there is some $R \sqsubseteq^* S$ with $\mathrm{Trans}(R)$ and y is an R-successor of x, then also $\langle \pi(x), \pi(y) \rangle \in \mathcal{E}_A(S)$ due to $(*)$. Since $\mathcal{T}$ is a tableau, (P7) from Definition 6.5 implies $\forall R.C \in \mathcal{L}(\pi(y))$ and hence the $\forall_+$-rule can be applied without violating $(*)$.

- The $\geqslant$-rule: If $(\geqslant n S.C) \in \mathcal{L}(x)$, then $(\geqslant n S.C) \in \mathcal{L}(\pi(x))$. Since $\mathcal{T}$ is a tableau, (P8) from Definition 6.5 implies $\sharp S^{\mathcal{T}}(\pi(x), C) \geqslant n$. Hence there are individuals $t_1, \ldots, t_n \in \mathbf{S}_A$ such that $\langle \pi(x), t_i \rangle \in \mathcal{E}_A(S)$, $C \in \mathcal{L}(t_i)$, and $t_i \neq t_j$ for $1 \leq i < j \leq n$. The $\geqslant$-rule generates n new abstract nodes $y_1, \ldots, y_n$. By setting $\pi' := \pi[y_1 \mapsto t_1, \ldots, y_n \mapsto t_n]$, we have a function π' that satisfies $(*)$ for the modified forest.

- The $\leqslant$-rule: If $(\leqslant n S.C) \in \mathcal{L}(x)$, then $(\leqslant n S.C) \in \mathcal{L}(\pi(x))$. Since $\mathcal{T}$ is a tableau, (P9) from Definition 6.5 implies $\sharp S^{\mathcal{T}}(\pi(x), C) \leqslant n$. If the $\leqslant$-rule is applicable, we have $\sharp S^{\mathsf{F}}(x, C) > n$, which implies that there are at least $n + 1$ S-successors $y_1, \ldots, y_{n+1}$ of x such that $C \in \mathcal{L}(y_i)$. Thus, there must be two nodes $y, z \in \{y_1, \ldots, y_{n+1}\}$ such that $\pi(y) = \pi(z)$ (because otherwise $\sharp S^{\mathcal{T}}(\pi(x), C) > n$ would hold). $\pi(y) = \pi(z)$ implies that $y \neq z$ cannot hold because of $(*)$. Hence the $\leqslant$-rule can be applied without violating $(*)$.

- The *choose*-rule: If $\{\geqslant n S.C, \leqslant n S.C\} \cap \mathcal{L}(x) \neq \emptyset$, then $\{\geqslant n S.C, \leqslant n S.C\} \cap \mathcal{L}(\pi(x)) \neq \emptyset$, and, if there is an S-successor y of x , then $\langle \pi(x), \pi(y) \rangle \in \mathcal{E}_A(S)$ due to $(*)$. Since $\mathcal{T}$ is a tableau, (P10) from Definition 6.5 implies $\{C, \sim C\} \cap \mathcal{L}(\pi(y)) \neq \emptyset$. Hence the *choose*-rule can add an appropriate concept $E \in \{C, \sim C\}$ to $\mathcal{L}$(y) such that $\mathcal{L}(y) \subseteq \mathcal{L}(\pi(y))$ holds.

- The **O**-rule: If $\{\mathsf{o}\} \in \mathcal{L}(x)$, then $\{\mathsf{o}\} \in \mathcal{L}(\pi(x))$, and if z is distinguished with $\{\mathsf{o}\} \in \mathcal{L}(z)$, then $\{\mathsf{o}\} \in \mathcal{L}(\pi(z))$ due to $(*)$. Since $\mathcal{T}$ is a tableau, (P11) from Definition 6.5 implies $\pi(x) = \pi(z)$; hence $\mathcal{L}(\pi(x)) = \mathcal{L}(\pi(z)) = \mathcal{L}(\pi(x)) \cup \mathcal{L}(\pi(z))$. Hence the **O**-rule can be applied without violating $(*)$.

- The $\forall_P$-rule: If $\forall T_1, \ldots, T_n.P \in \mathcal{L}(x)$, then $\forall T_1, \ldots, T_n.P \in \mathcal{L}(\pi(x))$, and, if $\langle v_1, \ldots, v_n \rangle$ is a $\langle T_1 \ldots T_n \rangle$-successor of x, then also $\langle \pi(x), \pi(v_j) \rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$ for all $1 \leq j \leq n$ due to $(*)$. Since $\mathcal{T}$ is a tableau, (P12') from Definition 6.5

implies $P(\pi(v_1), \ldots, \pi(v_n)) \in DC^{\mathcal{T}}$, hence $(\langle v_1, \ldots, v_n\rangle, P) \in DC(x)$ due to $(*)$. Hence the $\forall_P$-rule can be applied without violating $(*)$.

- The $\exists_P$-rule: If $\exists T_1, \ldots, T_n.P \in \mathfrak{L}(x)$, then $\exists T_1, \ldots, T_n.P \in \mathfrak{L}(\pi(x))$. Since $\mathcal{T}$ is a tableau, (P13') from Definition 6.5 implies that there are variables $w_1, \ldots, w_n \in \mathbf{S_D}$ such that $\langle \pi(x), w_j\rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$ for all $1 \leq j \leq n$, and $P(w_1, \ldots, w_n) \in DC^{\mathcal{T}}$. The application of the $\exists_P$-rule generates a new $\langle T_1 \ldots T_n\rangle$-successor $\langle v_1, \ldots, v_n\rangle$ with $\mathfrak{L}(\langle x, v_j\rangle) = \{T_j\}$ for all $1 \leq j \leq n$, and $(\langle v_1, \ldots, v_n\rangle, P) \in DC(x)$. Hence we set $\pi' := \pi[v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$ which yields a function that satisfies $(*)$ for the modified forest.

- The $\geqslant_P$-rule: If $(\geqslant m T_1, \ldots, T_n.P) \in \mathfrak{L}(x)$, then $(\geqslant m T_1, \ldots, T_n.P) \in \mathfrak{L}(\pi(x))$. Since $\mathcal{T}$ is a tableau, (P14') from Definition 6.5 implies $\sharp T^{\mathcal{T}}(\pi(x), P) \geqslant m$. Hence there are mn variables $w_{11}, \ldots, w_{mn} \in \mathbf{S_D}$ such that $\langle \pi(x), w_{ij}\rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$, $\langle \delta(w_{i1}), \ldots, \delta(w_{in})\rangle \in P^{\mathbf{D}}$ and $(\langle \delta(w_{i1}), \ldots, \delta(w_{in})\rangle, \langle \delta(w_{k1}), \ldots, \delta(w_{kn})\rangle) \in \neq^{\mathbf{D}}$ for all $1 \leq i < k \leq m, 1 \leq j \leq n$. The application of the $\geqslant_P$-rule generates m new $\langle T_1 \ldots T_n\rangle$-successors $\langle v_{11}, \ldots, v_{1n}\rangle$, …, $\langle v_{m1}, \ldots, v_{mn}\rangle$ of x, with $\mathfrak{L}(\langle x, v_{ij}\rangle) = \{T_j\}$, $(\langle v_{i1}, \ldots, v_{in}\rangle, P) \in DC(x)$ and $(\langle v_{i1}, \ldots, v_{in}\rangle, \langle v_{k1}, \ldots, v_{kn}\rangle, \neq) \in DC(x)$ for all $1 \leq i < k \leq m, 1 \leq j \leq n$. By setting $\pi' := \pi[v_{11} \mapsto w_{11}, \ldots, v_{mn} \mapsto w_{mn}]$, we have a function π' that satisfies $(*)$ for the modified forest.

- The $\leqslant_P$-rule: If $(\leqslant m T_1, \ldots, T_n.P) \in \mathfrak{L}(x)$, then $(\leqslant m T_1, \ldots, T_n.P) \in \mathfrak{L}(\pi(x))$. Since $\mathcal{T}$ is a tableau, (P15') from Definition 6.5 implies $\sharp T^{\mathcal{T}}(\pi(x), P) \leqslant m$. If the $\leqslant_P$-rule is applicable, we have $\sharp T^{\mathbf{F}}(x, P) > m$, which implies that there are at least $m+1$ $\langle T_1 \ldots T_n\rangle$-successors $\langle v_{11}, \ldots, v_{1n}\rangle$, …, $\langle v_{m+1,1}, \ldots, v_{m+1,n}\rangle$ of x such that $(\langle v_{i1}, \ldots, v_{in}\rangle, P) \in DC(x)$ for all $1 \leq i \leq m+1$. Thus, there must be two $\langle T_1 \ldots T_n\rangle$-successors $\langle v_{i1}, \ldots, v_{in}\rangle$, $\langle v_{j1}, \ldots, v_{jn}\rangle$ among these $m+1$ ones, such that $(\langle \delta(\pi(v_{i1})), \ldots, \delta(\pi(v_{in}))\rangle, \langle \delta(\pi(v_{j1})), \ldots, \delta(\pi(v_{jn}))\rangle) \in \neq^{\mathbf{D}}$ does not hold (otherwise $\sharp T^{\mathcal{T}}(\pi(x), P) > m$ would hold). This implies that $(\langle v_{i1}, \ldots, v_{in}\rangle, \langle v_{j1}, \ldots, v_{jn}\rangle, \neq) \notin DC(x)$. Hence the $\leqslant_P$-rule can be applied without violating $(*)$.

- The $choose_P$-rule: If $\{\leqslant m T_1, \ldots, T_n.P, \geqslant m T_1, \ldots, T_n.P\} \cap \mathfrak{L}(x) \neq \emptyset$, then $\{\leqslant m T_1, \ldots, T_n.P, \geqslant m T_1, \ldots, T_n.P\} \cap \mathfrak{L}(\pi(x)) \neq \emptyset$, and, if $\langle v_1, \ldots, v_n\rangle$ is a $\langle T_1 \ldots T_n\rangle$-successor of x, then $\langle \pi(x), \pi(v_j)\rangle \in \mathcal{E}_{\mathbf{D}}(T_j)$ for all $1 \leq j \leq n$ due to $(*)$. Since $\mathcal{T}$ is a tableau, (P16') from Definition 6.5 implies that

$P(\pi(v_1),\ldots,\pi(v_n)) \in DC^{\mathcal{T}}$ or $\neg P(\pi(v_1),\ldots,\pi(v_n)) \in DC^{\mathcal{T}}$. Hence the $choose_P$-rule can add an appropriate constraint $(\langle v_1,\ldots,v_n\rangle, E)$, where $E \in \{P, \neg P\}$, to $DC(x)$ due to $(*)$.

Why does this claim yield the completeness of the completion algorithm? For the initial completion forest including a node x_0 with $\mathcal{L}(x_0) = \{D\}$, we can give a function π that satisfies $(*)$ by setting $\pi(x_0) := s_0$ for some $s_0 \in \mathbf{S}_A$ with $D \in \mathcal{L}(s_0)$ (such an s_0 exists since $\mathcal{T}$ is a tableau for D). Whenever a rule is applicable to $\mathtt{F}$, it can be applied in a way that maintains $(*)$, and, from Lemma 6.8, we have that any sequence of rule applications must terminate. Since $(*)$ holds, any forest generated by these rule-applications must be clash-free. This can be seen as follows:

- $\mathtt{F}$ cannot contain a node x such that $\bot \in \mathcal{L}(x)$ because $\mathcal{L}(x) \subseteq \mathcal{L}(\pi(x))$, hence $\bot \in \mathcal{L}(\pi(x))$. This implies $\pi(x) \in \bot^{\mathcal{I}}$, which contradicts the interpretation of $\bot$.

- $\mathtt{F}$ cannot contain a node x such that $\{C, \sim C\} \in \mathcal{L}(x)$ because $\mathcal{L}(x) \subseteq \mathcal{L}(\pi(x))$ and hence (P1) of Definition 6.5 would be violated for $\pi(x)$.

- $\mathtt{F}$ cannot contain a node x with $(\leqslant nS.C) \in \mathcal{L}(x)$ and $n+1$ S-successors $y_1,\ldots,y_{n+1}$ of x with $C \in \mathcal{L}(y_i)$ and $y_i \neq y_j$ for $1 \leq i < j \leq n+1$ because $(\leqslant nS.C) \in \mathcal{L}(\pi(x))$ and $y_i \neq y_j$ implies $\pi(y_i) \neq \pi(y_j)$, $\sharp S^{\mathcal{T}}(\pi(x), C) > n$ would hold which contradicts (P9) of Definition 6.5.

- $\mathtt{F}$ cannot contain a node x with $(\leqslant m T_1,\ldots,T_n.P) \in \mathcal{L}(x)$ and $m+1$ $\langle T_1 \ldots T_n\rangle$-successors $\langle v_{11},\ldots,v_{1n}\rangle$, …, $\langle v_{m+1,1},\ldots,v_{m+1,n}\rangle$ of x with $(\langle v_{i1},\ldots,v_{in}\rangle, P) \in DC(x)$ for each $1 \leq i \leq m+1$, and $(\langle v_{i1},\ldots,v_{in}\rangle, \langle v_{j1},\ldots,v_{jn}\rangle, \neq) \in DC(x)$ for all $1 \leq i < j \leq m+1$, which implies that $(\langle \delta(\pi(v_{i1})),\ldots, \delta(\pi(v_{in}))\rangle$, $\langle \delta(\pi(v_{j1})),\ldots, \delta(\pi(v_{jn}))\rangle) \in \neq^{\mathbf{D}}$ would hold which contradicts (P15) of Definition 6.5.

- $\mathtt{F}$ cannot contain a node x where $DC(x)$ is not satisfied, otherwise δ is not a solution for the datatype constraints in $DC(x)$, which implies that δ is not a solution for datatype constraints about $\pi(x)$ in $DC^{\mathcal{T}}$, or it is not a solution for the number restrictions of $T_1,\ldots,T_n^{\mathcal{T}}(\pi(x), P)$ due to $(*)$. This contradicts (P0) of Definition 6.5.

- $\mathtt{F}$ cannot contain a node x with $\{\mathsf{o}\} \in \mathcal{L}(x)$ and $x \neq z_{\{\mathsf{o}\}}$ because $\{\mathsf{o}\} \in \mathcal{L}(\pi(x))$ and $\{\mathsf{o}\} \in \mathcal{L}(\pi(z_{\{\mathsf{o}\}}))$, hence $\{\mathsf{o}\} \in \mathcal{L}(\pi(x)) \cap \mathcal{L}(\pi(z_{\{\mathsf{o}\}}))$, while $x \neq$

$z_{\{\mathsf{O}\}}$ implies $\pi(x) \neq \pi(z_{\{\mathsf{O}\}})$ would hold which contradicts (P11) of Definition 6.5. □

As an immediate consequence of Lemmas 6.6, 6.8, 6.9 and 6.10, the completion algorithm always terminates, and answers with "D is *satisfiable* w.r.t. $\mathcal{R}$" iff D has a tableau T. Next, subsumption can be reduced to (un)satisfiability because $\mathcal{SHOQ}(\mathcal{G})$ is closed under negation (cf. Section 2.1.3). Finally, $\mathcal{SHOQ}(\mathcal{G})$ can internalise general concept inclusion axioms (cf. Theorem 2.8 on page 36). In the presence of nominals, however, we must also add $\exists O.o_1 \sqcap \ldots \sqcap \exists O.o_l$ to the concept internalising the general concept inclusion axioms to make sure that the universal role O indeed reaches all nominals o_i occurring in the input concept and terminology. Thus, we can decide these inference problems also w.r.t. TBoxes and RBoxes [75].

Theorem 6.11. *The completion algorithm presented in Definition 6.7 is a decision procedure for satisfiability and subsumption of $\mathcal{SHOQ}(\mathcal{G})$-concepts w.r.t. TBoxes and RBoxes.*

Combining this result with Theorem 2.7, we obtain the following theorem.

Theorem 6.12. *The completion algorithm presented in Definition 6.7 is a decision procedure for $\mathcal{SHOQ}(\mathcal{G})$ knowledge base satisfiability.*

In fact, we can generalise the way we extend the tableaux algorithm for $\mathcal{SHOQ}$ to the one for $\mathcal{SHOQ}(\mathcal{G})$ by so call $\mathcal{G}$-augmented tableaux algorithms.

Definition 6.13. ($\mathcal{G}$-augmented Tableaux Algorithm) Given a conforming datatype group $\mathcal{G}$ and a $\mathcal{G}$-combinable Description Logic $\mathcal{L}$, if there exists a tableaux algorithm $\mathtt{TA}_{\mathcal{L}}$ that is a decision procedure for $\mathcal{L}$-concept satisfiability w.r.t. RBoxes, then the *$\mathcal{G}$-augmented tableaux algorithm* (w.r.t. $\mathtt{TA}_{\mathcal{L}}$) $\mathtt{TA}_{\mathcal{L}(\mathcal{G})}$ is the same as $\mathtt{TA}_{\mathcal{L}}$, except that:

- the set of completion rules in $\mathtt{TA}_{\mathcal{L}(\mathcal{G})}$ is the union of the set of completion rules (or *$\mathcal{L}$-rules*) in $\mathtt{TA}_{\mathcal{L}}$ and the $\mathcal{G}$-rules listed in Figure 6.2 on page 143;
- the set of clash conditions in $\mathtt{TA}_{\mathcal{L}(\mathcal{G})}$ is the union of the set of clash conditions (or *$\mathcal{L}$-clash conditions*) in $\mathtt{TA}_{\mathcal{L}}$ and the $\mathcal{G}$-clash conditions listed in Figure 6.4 on page 144. ◇

The following theorem shows that $\mathtt{TA}_{\mathcal{L}(\mathcal{G})}$ is a decision procedure for $\mathcal{L}(\mathcal{G})$-concept satisfiability w.r.t. RBoxes.

Theorem 6.14. *Let $\mathcal{L}$ be a $\mathcal{G}$-combinable Description Logic, $\mathtt{TA}_{\mathcal{L}}$ a tableaux algorithm that is a decision procedure for $\mathcal{L}$-concept satisfiability w.r.t. to RBoxes. The $\mathcal{G}$-augmented tableaux algorithm $\mathtt{TA}_{\mathcal{L}(\mathcal{G})}$ is a decision procedure for $\mathcal{L}(\mathcal{G})$-concept satisfiability w.r.t. to RBoxes.*

Proof. This is mainly due to the locality of $\mathcal{G}$-rules and $\mathcal{G}$-clash conditions.

- $\mathcal{G}$-rules are applicable only when there exist datatype group-based concepts in the label of a node x, and they only affect the datatype query of x, its concrete successors and the edges connecting x and its concrete successors (cf. Definition 6.7). In other words, they will not affect the applicability of $\mathcal{L}$-rules.

- The $\mathcal{G}$-clash conditions will be triggered only when some local datatype constraint is not satisfied. Hence, $\mathcal{G}$-clash conditions and $\mathcal{L}$-clash conditions cover completely different aspects of the completion forest (or tree).

Therefore, $\mathcal{G}$-rules and $\mathcal{G}$-clash conditions can be seen as extra local checking about datatype constraints added into $\mathtt{TA}_{\mathcal{L}}$. As $\mathtt{TA}_{\mathcal{L}}$ is decidable, we only need to show that $\mathcal{G}$-rules and $\mathcal{G}$-clash conditions are sound and complete on checking local datatype constraints, and that $\mathcal{G}$-rules are terminating. All these are witnessed by the fact that the $\mathcal{G}$-augmented tableaux algorithm $\mathtt{TA}_{\mathcal{SHOQ}(\mathcal{G})}$ defined in Definition 6.7 on page 140 is sound, complete and terminating. □

Theorem 6.14 suggests that if we have a decidable tableaux algorithm for $\mathcal{SHOIQ}$-concept satisfiability w.r.t. RBoxes, we can construct a decidable $\mathcal{G}$-augmented tableaux algorithm for $\mathcal{SHOIQ}(\mathcal{G})$-concept satisfiability w.r.t. RBoxes.

In the next section, we will provide a $\mathcal{G}$-augmented tableaux algorithm to decide $\mathcal{SHIQ}(\mathcal{G})$-concept satisfiability w.r.t. RBoxes.

6.2.2 The $\mathcal{SHIQ}(\mathcal{G})$ DL

Syntax and Semantics

The $\mathcal{SHIQ}(\mathcal{G})$ DL extends the $\mathcal{SHIQ}$ DL with an arbitrary datatype group. As usual, we define $\mathcal{SHIQ}(\mathcal{G})$-roles, $\mathcal{SHIQ}(\mathcal{G})$-concepts and $\mathcal{SHIQ}(\mathcal{G})$-RBox as follows.

Definition 6.15. **($\mathcal{SHIQ}(\mathcal{G})$-roles)** Let $RN \in \mathbf{R}_A$, $R \in \mathbf{Rdsc}_A(\mathcal{SHIQ}(\mathcal{G}))$, $TN \in \mathbf{R}_\mathbf{D}$ and $T \in \mathbf{Rdsc}_\mathbf{D}(\mathcal{SHIQ}(\mathcal{G}))$. Valid $\mathcal{SHIQ}(\mathcal{G})$ abstract roles are defined by the abstract syntax:

$$R ::= RN \mid R^-;$$

where for some $x, y \in \Delta^{\mathcal{I}}$, $\langle x, y\rangle \in R^{\mathcal{I}}$ iff $\langle y, x\rangle \in R^{-\mathcal{I}}$. The inverse relation of roles is symmetric, and to avoid considering the roles such as R^{--}, we define a function Inv which returns the inverse of a role, more precisely

$$\mathsf{Inv}(R) := \begin{cases} RN^{-} & \text{if } R = RN, \\ RN & \text{if } R = RN^{-}. \end{cases}$$

Valid $\mathcal{SHIQ}(\mathcal{G})$ concrete roles are defined by the abstract syntax:

$$T ::= TN.$$

$\diamond$

Definition 6.16. **($\mathcal{SHIQ}(\mathcal{G})$-concepts)** Let $\mathsf{CN} \in \mathbf{C}$, $R \in \mathbf{Rdsc}_A(\mathcal{SHIQ}(\mathcal{G}))$, $T_1, \ldots, T_n \in \mathbf{Rdsc}_{\mathbf{D}}(\mathcal{SHIQ}(\mathcal{G}))$ and $T_i \not\sqsubseteq T_j, T_j \not\sqsubseteq T_i$ for all $1 \leq i < j \leq n$, $C, D \in \mathbf{Cdsc}(\mathcal{SHIQ}(\mathcal{G}))$, $E \in \mathbf{Dexp}(\mathcal{G})$, $n, m \in \mathbb{N}$, $n \geq 1$. Valid $\mathcal{SHIQ}(\mathcal{G})$-concepts are defined by the abstract syntax:

$$\begin{aligned} C \quad ::= \quad & \top \mid \bot \mid \mathsf{CN} \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C \mid \geqslant mR.C \mid \leqslant mR.C \\ & \exists T_1, \ldots, T_n.E \mid \forall T_1, \ldots, T_n.E \mid \geqslant mT_1, \ldots, T_n.E \mid \leqslant mT_1, \ldots, T_n.E. \end{aligned}$$

The semantics of datatype group-based $\mathcal{SHIQ}(\mathcal{G})$-concepts is given in Table 5.2 on page 118; the semantics of other $\mathcal{SHIQ}(\mathcal{G})$-concepts is given in Table 2.2 on page 46.

$\diamond$

As the $\mathcal{SHIQ}(\mathcal{G})$ DL provides the inverse role constructor, its RBoxes are more expressive than those of the $\mathcal{SHOQ}(\mathbf{D})$ and $\mathcal{SHOQ}(\mathcal{G})$ DLs.

Definition 6.17. **($\mathcal{SHIQ}(\mathcal{G})$ RBox)** Let $R_1, R_2 \in \mathbf{Rdsc}_A(\mathcal{SHIQ}(\mathcal{G}))$, $T_1, T_2 \in \mathbf{Rdsc}_{\mathbf{D}}(\mathcal{SHIQ}(\mathcal{G}))$, $SN \in \mathbf{R}$, a $\mathcal{SHIQ}(\mathcal{G})$ RBox $\mathcal{R}$ is a finite, possibly empty, set of role axioms:

- functional role axioms $\mathsf{Func}(SN)$;
- transitive role axioms $\mathsf{Trans}(R_1)$;[8]
- abstract role inclusion axioms $R_1 \sqsubseteq R_2$;
- concrete role inclusion axioms $T_1 \sqsubseteq T_2$.

[8]Note that a concrete role T can not be a transitive role.

We extend $\mathcal{R}$ to the *role hierarchy* of $\mathcal{SHIQ}(\mathcal{G})$ as follows:

$$\mathcal{R}^+ := (\mathcal{R} \cup \{\mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\}, \sqsubseteq^*)$$

where $\sqsubseteq^*$ is the transitive-reflexive closure of $\sqsubseteq$ over $\mathcal{R} \cup \{\mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\}$. ◇

A $\mathcal{G}$-augmented Tableaux Algorithm for $\mathcal{SHIQ}(\mathcal{G})$

Based on the tableaux algorithm for $\mathcal{SHIQ}$-concept satisfiability w.r.t. role hierarchies presented by Horrocks et al. [65], we can construct a $\mathcal{G}$-augmented tableaux algorithm to decide $\mathcal{L}(\mathcal{G})$-concept satisfiability problem w.r.t. a role hierarchy.

Definition 6.18. (**A $\mathcal{G}$-augmented Tableaux Algorithm for $\mathcal{SHIQ}(\mathcal{G})$**) Let $\mathcal{R}^+$ be a role hierarchy, D a $\mathcal{SHIQ}(\mathcal{G})$-concept in NNF, $\mathbf{R}_A^D$ the set of abstract roles in D or $\mathcal{R}^+$, together with their inverse, $\mathbf{R}_\mathbf{D}^D$ the set of concrete roles occurring in D or $\mathcal{R}^+$ and $\mathcal{G}$ a datatype group.

A tableaux algorithm for $\mathcal{SHIQ}(\mathcal{G})$ is similar to the one for $\mathcal{SHOQ}(\mathcal{G})$ defined in Definition 6.7 except that

- it works on a *completion tree* (cf. Section 2.2) $\mathbf{T}$ for D w.r.t. $\mathcal{R}^+$, and
- for (abstract) nodes x, y, the label on the edge $\langle x, y\rangle$, namely $\mathcal{L}(\langle x, y\rangle)$, could possibly contain inverse roles, and
- $S^\mathbf{T}(x, C)$ is defined in terms of 'neighbours', instead of 'successors', and
- this tableaux algorithm uses pair-wise blocking.

Given a completion tree, an (abstract) node y is called an R-*neighbour* of a node x if, for some R' with $R' \sqsubseteq^* R$, either y is a successor of x and $R' \in \mathcal{L}(\langle x, y\rangle)$, or y is a predecessor of x and $\mathsf{Inv}(R') \in \mathcal{L}(\langle x, y\rangle)$. For an abstract role S and a node x in $\mathbf{T}$, we define $S^\mathbf{T}(x, C)$ as

$$S^\mathbf{T}(x, C) := \{y \mid y \text{ is an } S\text{-neighbour of } x \text{ and } C \in \mathcal{L}(y)\}.$$

A node x is *directly blocked* if none of its ancestors are blocked, and it has ancestors x', y and y' such that

1. x is a successor of x' and y is a successor of y' and

$\sqcap$-rule:	if 1.$C_1 \sqcap C_2 \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.$\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$, then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-rule:	if 1.$C_1 \sqcup C_2 \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.$\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$, then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$
$\exists$-rule:	if 1.$\exists R.C \in \mathcal{L}(x)$, x is not blocked, and 2.x has no R-neighbour y with $C \in \mathcal{L}(y)$ then create a new node y with $\mathcal{L}(\langle x, y\rangle) = \{R\}$ and $\mathcal{L}(y) = \{C\}$
$\forall$-rule:	if 1.$\forall R.C \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.there is an R-neighbour y of x with $C \notin \mathcal{L}(y)$, then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$
$\forall_+$-rule:	if 1.$\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.there is some R with $\mathsf{Trans}(R)$ and $R \sqsubseteq S$, 3.and there is an R-neighbour y of x with $\forall R.C \notin \mathcal{L}(y)$, then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\forall R.C\}$
$\geqslant$-rule:	if 1.$\geqslant nS.C \in \mathcal{L}(x)$, x is not blocked, and 2.there are no n S-neighbour $y_1, \ldots, y_n$ of x with $C \in \mathcal{L}(y_i)$ and 3.$y_i \neq y_j$ for $1 \leq i < j \leq n$, then create n new nodes $y_1, \ldots, y_n$ with $\mathcal{L}(\langle x, y_i\rangle) = \{S\}$, $\mathcal{L}(y_i) = \{C\}$, and $y_i \neq y_j$ for $1 \leq i < j \leq n$.
$\leqslant$-rule:	if 1.$\leqslant nS.C \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.$S^{\mathbf{T}}(x, C) > n$ and there are two S-neighbours y, z of x with $C \in \mathcal{L}(y), C \in \mathcal{L}(z)$ y is not an ancestor of z, and not $y \neq z$ then1. $\mathcal{L}(y_i) \longrightarrow \mathcal{L}(y_i) \cup \mathcal{L}(y_j)$ and 2. add $y \neq y_i$ for each y with $y \neq y_j$, and 3. $\mathcal{L}(\langle x, y_j\rangle) \longrightarrow \emptyset$
choose-rule:	if 1.$\{\geqslant nS.C, \leqslant nS.C\} \cap \mathcal{L}(x) \neq \emptyset$, x is not indirectly blocked, and 2.y is an S-neighbour of x with $\{C, \sim C\} \cap \mathcal{L}(y) = \emptyset$, then $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{E\}$ for some $E \in \{C, \sim C\}$

Figure 6.5: The completion rules for $\mathcal{SHIQ}$

A node x of a completion forest $\mathtt{F}$ contains a *clash* if (at least) one of the following conditions holds:

(1) $\bot \in \mathcal{L}(x)$;

(2) for some $\mathsf{A} \in \mathbf{C}$, $\{\mathsf{A}, \neg\mathsf{A}\} \subseteq \mathcal{L}(x)$;

(3) for some abstract role S, $\leqslant nS.C \in \mathcal{L}(x)$ and there are $n+1$ S-neighbours $y_1, \ldots, y_{n+1}$ of x with $C \in \mathcal{L}(y_i)$ for each $1 \leq i \leq n+1$ and $y_i \neq y_j$ for each $1 \leq i < j \leq n+1$;

Figure 6.6: The clash conditions of $\mathcal{SHIQ}$

2. $\mathcal{L}(x) = \mathcal{L}(y)$ and $\mathcal{L}(x') = \mathcal{L}(y')$ and

3. $\mathcal{L}(\langle x', x\rangle) = \mathcal{L}(\langle y', y\rangle)$.

In this case we say that y blocks x.

A node is *indirectly blocked* if its predecessor is blocked, and in order to avoid wasted expansion after an application of the $\leqslant$-rule, a node y will also be taken to be indirectly blocked if it is a successor of a node x and $\mathcal{L}(\langle x, y\rangle) = \emptyset$.

The algorithm initialises the tree $\mathbf{T}$ the root node x_0 labeled with $\mathcal{L}(x_0) = \{D\}$, the inequality relation $\dot{\neq}$ is initialised with the empty relation and the $DC(x)$ of any (abstract) node x is initialised to the empty set. $\mathbf{T}$ is then expanded by repeatedly applying the completion rules, listed in Figure 6.5 (page 159) and Figure 6.2 (page 143), until $\mathbf{T}$ is complete, or some $\mathcal{L}(x)$ contains a clash (cf. Figure 6.6 on page 159 and Figure 6.4 on page 144).

If the completion rules can be applied in such a way that they yield a complete, clash-free completion tree, then the algorithm returns "D is *satisfiable* w.r.t. $\mathcal{R}^+$", and "D is *unsatisfiable* w.r.t. $\mathcal{R}^+$" otherwise. $\diamond$

According to Definition 6.17 on page 157, a role hierarchy is simply a syntactic variant of the set of role inclusion axioms in an RBox when inclusion axioms ($\mathcal{H}$) is present. According to Theorem 6.14 and the fact that $\mathcal{SHIQ}(\mathcal{G})$ can internalise TBoxes (cf. Theorem 2.8 on page 36), we have the following theorem.

Theorem 6.19. *The $\mathcal{G}$-augmented tableaux algorithm $\mathtt{TA}_{\mathcal{SHIQ}(\mathcal{G})}$ presented in Definition 6.18 is a decision procedure for $\mathcal{SHIQ}(\mathcal{G})$-concept satisfiability and subsumption problems w.r.t. TBoxes and RBoxes.*

6.2.3 The $\mathcal{SHIO}(\mathcal{G})$ DL

Having introduced $\mathcal{SHOQ}(\mathcal{G})$ and $\mathcal{SHIQ}(\mathcal{G})$, now we briefly describe $\mathcal{SHIO}(\mathcal{G})$, which extends $\mathcal{SHIO}$ with an arbitrary datatype group. $\mathcal{SHIO}$-roles are the same as $\mathcal{SHIQ}$-roles, $\mathcal{SHIO}$-RBoxes are the same as $\mathcal{SHIQ}$-RBoxes and $\mathcal{SHIO}$-concepts are defined as follows.

Definition 6.20. **($\mathcal{SHIO}(\mathcal{G})$-concepts)** Let $\mathsf{CN} \in \mathbf{C}$, $R \in \mathbf{Rdsc}_A(\mathcal{SHIO}(\mathcal{G}))$, T_1, $\ldots, T_n \in \mathbf{Rdsc}_{\mathbf{D}}(\mathcal{SHIO}(\mathcal{G}))$ and $T_i \not\sqsubseteq T_j, T_j \not\sqsubseteq T_i$ for all $1 \leq i < j \leq n$, $C, D \in \mathbf{Cdsc}(\mathcal{SHIO}(\mathcal{G}))$, $o \in \mathbf{I}$, $E \in \mathbf{Dexp}(\mathcal{G})$, $n, m \in \mathbb{N}$, $n \geq 1$. Valid $\mathcal{SHIO}(\mathcal{G})$-concepts are defined by the abstract syntax:

$$
\begin{aligned}
C \;::=\; & \top \mid \bot \mid \mathsf{CN} \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \{o\} \mid \exists R.C \mid \forall R.C \mid \\
& \exists T_1, \ldots, T_n.E \mid \forall T_1, \ldots, T_n.E \mid \geqslant m T_1, \ldots, T_n.E \mid \leqslant m T_1, \ldots, T_n.E.
\end{aligned}
$$

The semantics of datatype group-based $\mathcal{SHIO}(\mathcal{G})$-concepts is given in Table 5.2 on page 118; the semantics of other $\mathcal{SHIO}(\mathcal{G})$-concepts is given in Table 2.2 on page 46. ◇

According to Theorem 6.14, we can construct a $\mathcal{G}$-augmented tableaux algorithm, based on the tableaux algorithm for $\mathcal{SHIO}$-concept satisfiability w.r.t. RBoxes presented by Jladik and Model [78], to decide $\mathcal{SHIO}(\mathcal{G})$-concept satisfiability w.r.t. RBoxes. Due to Theorem 2.7 (page 35) and Theorem 2.8 (page 36), such an algorithm is also a decision procedure for $\mathcal{SHIO}(\mathcal{G})$-knowledge base satisfiability.

We end this section by concluding that we can construct decidable $\mathcal{G}$-augmented tableaux algorithms for a family of DLs, of the form $\mathcal{L}(\mathcal{G})$, which satisfy the following conditions:

1. $\mathcal{L}$ is a $\mathcal{G}$-combinable Description Logic,
2. $\mathcal{G}$ is a conforming datatype group, and
3. there exists a decidable tableaux algorithm for $\mathcal{L}$-concept satisfiability w.r.t. RBoxes (or role hierarchies).

We have investigated several example $\mathcal{G}$-augmented tableaux algorithms, viz. those for $\mathcal{SHOQ}(\mathcal{G})$, $\mathcal{SHIQ}(\mathcal{G})$ and $\mathcal{SHIO}(\mathcal{G})$, which can be used to provide reasoning support for OWL-E under the ‘divide and conquer’ strategy.

6.3 OWL-Eu: A Smaller Extension of OWL DL

Until now, we have discussed OWL-E as a decidable extension of OWL DL that provides customised datatypes and predicates, but we may also consider the possibility of having a smaller extension of OWL DL. The alternative extension we are going to present in this section is called OWL-Eu, which is a unary restriction of OWL-E.

The Small Extension Requirement

When we extend OWL DL to support customised datatypes and predicates, some people prefer the so called *small extension requirement*, which is two folded: on the one hand, an extension should be a substantial *extension* to support customised datatypes and predicates; on the other hand, following W3C’s ‘one small step at a time’ strategy, the extension should only be *small*.

Abstract Syntax	DL Syntax	Semantics
rdfs:Literal	$\top_{\mathrm{D}}$	$\Delta_{\mathbf{D}}$
owlx:DatatypeBottom	$\bot_{\mathrm{D}}$	$\emptyset$
u a unary predicate URIref	u	$u^{\mathbf{D}}$
not(u)	$\overline{u}$	if $u \in \mathbf{D}_{\mathcal{G}}$, $\Delta_{\mathbf{D}} \setminus u^{\mathbf{D}}$ if $u \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}$, $(\mathsf{dom}(u))^{\mathbf{D}} \setminus u^{\mathbf{D}}$ if $u \notin \Phi_{\mathcal{G}}$, $\Delta_{\mathbf{D}} \setminus u^{\mathbf{D}}$
oneOf("s_1"^^d_1 ... "s_n"^^d_n)	{"s_1"^^$d_1, \ldots,$ "s_n"^^d_n}	{("s_1"^^d_1)$^{\mathbf{D}}$} $\cup \ldots \cup$ {("s_n"^^d_n)$^{\mathbf{D}}$}
and(p, q)	$p \wedge q$	$p^{\mathbf{D}} \cap q^{\mathbf{D}}$
or(p, q)	$P \vee q$	$p^{\mathbf{D}} \cup q^{\mathbf{D}}$

Table 6.4: OWL-Eu unary datatype expressions

In this section, we provide OWL-Eu, a unary restriction of OWL-E, as a small extension of OWL DL.

From OWL DL to OWL-Eu

OWL-Eu only allows *unary datatype groups*; i.e., given a datatype group $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_{\mathcal{G}}, \mathsf{dom})$, $\mathbf{M}_p$ contains only pairs of predicate URIrefs and unary datatype predicates and thus $\Phi_{\mathcal{G}}$ contains only unary supported predicate URIrefs. Accordingly, OWL-Eu only supports unary $\mathcal{G}$-datatype expressions. Customised datatypes, such as 'greaterThan20' and 'cameraPrice' presented in Example 5.7 are expressible in OWL-Eu, while customised datatype predicates with arities greater than one, such as 'sumNoGreaterThan15' or 'multiplyBy1.6' in Example 5.7, are not expressible in OWL-Eu.

The only difference between OWL DL and OWL-Eu is that the latter one extends data ranges in the former one (listed in Table 3.4) to unary datatype expressions listed in Table 6.4, where u is a unary datatype predicate URIref, "s_i"^^d_i are typed literals, p, q are datatype expressions. Therefore, unary datatype expressions can be used as data ranges in datatype range axioms and datatype-related class descriptions (cf. Table 3.5 and 3.6).

Example 6.4 OWL-Eu: Matchmaking

The PCs with $memorySizeInMb$ greater than 512, unit $priceInPound$ less than 700 and $deliveryDate$ earlier than 15/03/2004 can be expressed by the following OWL-Eu class description

```
Class(RequiredPC complete PC
    restriction(memorySizeInMb
        someValuesFrom(owlx:integerGreatThanx=512))
    restriction(priceInPound
```

```
        someValuesFrom(owlx:integerLessThanx=700))
    restriction(deliveryDate
        someValuesFrom(owlx:dateEarlierThanx=2004-03-15))).
DatatypeProperty(memorySizeInMb Functional)
DatatypeProperty(priceInPound Functional)
DatatypeProperty(deliveryDate Functional)
```

Note that (i) the key word `someValuesFrom`, instead of `someTuplesSatisfy` is used in the above class description (cf. Example 6.1), and that (ii) functional property axioms are used in place of qualified number restrictions. ♢

Reasoning with OWL-Eu

The underpinning of OWL-Eu is the $\mathcal{SHOIN}(\mathcal{G}_1)$ DL, where '1' means it only allows unary datatype groups. To support OWL-Eu, similarly to how we provide reasoning support for OWL-E, we can use the 'divide and conquer' strategy and exploit decision procedures for $\mathcal{SHON}(\mathcal{G}_1)$, $\mathcal{SHIN}(\mathcal{G}_1)$ and $\mathcal{SHOI}(\mathcal{G}_1)$, which are just simplified versions of $\mathcal{SHOQ}(\mathcal{G})$, $\mathcal{SHIQ}(\mathcal{G})$ and $\mathcal{SHIO}(\mathcal{G})$, respectively.[9] Note that the $\mathcal{SHIN}(\mathcal{G}_1)$ DL fully covers OWL Lite.

According to Theorem 6.14 (page 156), Theorem 2.7 (page 35) and Theorem 2.8 (page 36) if we have a decidable tableaux algorithm for $\mathcal{SHOIN}$-concept satisfiability w.r.t. role hierarchies, we can easily extend it and provide a $\mathcal{G}$-augmented tableaux algorithm for $\mathcal{SHOIN}(\mathcal{G}_1)$-knowledge base satisfiability.

In Chapter 7, we will introduce the architecture of our datatype framework for providing DL inferencing services for OWL and OWL-E (and therefore OWL-Eu).

[9] Note that '$\mathcal{Q}$' stands for qualified number restrictions, while '$\mathcal{N}$' stands for non-qualified number restrictions.

Chapter Achievements

- We provide OWL-E and its unary restriction OWL-Eu, both of which support customised datatypes and datatype predicates, as decidable extensions of OWL DL.
- OWL-E is an extension of both OWL DL and DAML+OIL. It overcomes all the limitations of OWL datatyping discussed in Chapter 3.
- OWL-Eu is only a small extension of OWL DL, it extends OWL data ranges to support unary $\mathcal{G}$-datatype expressions.
- Theorem 6.14 shows that if we have a tableaux algorithm for the concept satisfiability problem w.r.t. to RBoxes of a $\mathcal{G}$-combinable Description Logic $\mathcal{L}$, then we can provide a $\mathcal{G}$-augmented tableaux algorithm to decide $\mathcal{L}(\mathcal{G})$-concept satisfiability w.r.t. RBoxes.
- We can provide reasoning support for OWL DL, DAML+OIL, OWL-Eu and OWL-E by using the decision procedures for the $\mathcal{SHOQ}(\mathcal{G})$, $\mathcal{SHIQ}(\mathcal{G})$ and $\mathcal{SHIO}(\mathcal{G})$ DLs under the 'divide and conquer' strategy.

Chapter 7

Framework Architecture

Chapter Aims

- To investigate a general DL interface for OWL DL and OWL-E.
- To propose a framework architecture to provide flexible and decidable DL reasoning services for DLs integrated with datatype groups.

Chapter Plan

Having provided a framework for combining DLs with customised datatypes and predicates in previous chapters, this chapter further investigates a flexible architecture for our framework.

7.1 General DL API

A key component of our framework architecture is a general DL API for OWL DL and OWL-E. This API is an extension of DIG/1.1 [10], which is a simple API for a general DL system developed by the DL Implementation Group (DIG) [36, 37]. There is a commitment from the implementors of the leading DL reasoners (FaCT [59, 40], FaCT++ [140, 41], Racer [51, 124], Cerebra [27] and Pellet [120]) to provide implementations conforming to the DIG specifications. In the following, Section 7.1.1

will describe the current version of DIG specification, DIG/1.1, which is not designed for OWL but for DAML+OIL. Section 7.1.2 will present our extension of DIG/1.1.

7.1.1 DIG/1.1 Interface

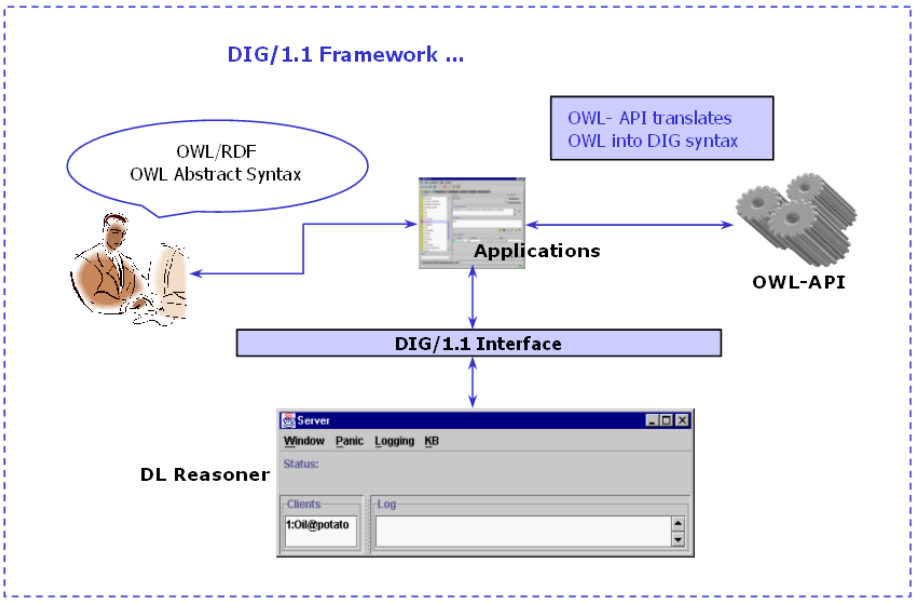

Figure 7.1: DIG/1.1 framework architecture

DIG/1.1 is effectively an XML Schema for the $\mathcal{SHOIQ}(\mathcal{D}_1)$ description language (Table 7.1 on page 167)[1] along with tell/ask/response functionality. Figure 7.1 on page 166 presents the DIG/1.1 framework architecture. Applications can take an OWL DL ontology, either in OWL/RDF syntax or in OWL abstract syntax (cf. Section 3.2.2) and use the OWL-API [11][2] to translate it into an ontology in DIG/1.1 syntax. Then the clients (applications) communicate with a DIG/1.1 server (a Description Logic reasoner) through the use of HTTP `POST` request, and the DIG server should return `200 OK`, unless there is a low-level error.

DIG/1.1 Communications

There are four kinds of requests from clients and a server:

[1] Notations of Table 7.1: $\mathtt{C}, \mathtt{C}_i$ are concepts, $\mathtt{R}$, is a role, $\mathtt{F}, \mathtt{F}_i$ are features, $\mathtt{A}$ is an attribute.

[2] OWL-API can also translate OWL/RDF syntax into OWL abstract syntax, cf. `http://owl.man.ac.uk/api.shtml`.

Primitive Concepts	<top/> <bottom/> <catom name = "CN"/>
Boolean Constructors	<and> C_1 ... C_n </and> <or> C_1 ... C_n </or> <not> C </not>
Abstract Role Restrictions	<some> R C </some> <all> R C </all> <atmost num = "n"> R C </atmost> <atleast num = "n"> R C </atleast> <iset> I_1 ... I_n </iset>
Unary Predicate Restrictions	<defined> A </defined> <stringmin val = "s"> A </stringmin> <stringmax val = "s"> A </stringmax> <stringequals val = "s"> A </stringequals> <stringrange min = "s" max = "t"> A </stringrange> <intmin val = "i"> A </intmin> <intmax val = "i"> A </intmax> <intequals val = "i"> A </intequals> <intrange min = "i" max = "j"> A </intrange>
Role Constructors	<ratom name = "RN"/> <feature name = "FN"/> <attribute name = "AN"/> <inverse> R </inverse> <chain> F_1 ... F_n A </chain>
Individuals	<individual name = "IN"/>

Table 7.1: DIG/1.1 description language

- Clients can find out which reasoner is actually behind the interface by sending an IDENTIFIER request. The server then returns the name and the version of the reasoner, together with the list of constructors, TELL and ASK operations supported by the reasoner.

- Clients can request to create or release knowledge bases.

- Clients can send TELL requests using the TELL language presented in Table 7.2 on page 168, where I, I_i are individual objects and V is either an integer value or a string value. The TELL requests are monotonic, i.e., DIG/1.1 does not support removing the TELL requests that have been sent – the only work-around is to release the knowledge base and start again. A TELL request must be made in the context of a particular knowledge base (by using the knowledge base URI). The server will response using the basic responses presented in Table 7.4 on

KB Control	<releaseKB/>
Concept Axioms	<defconcept name = "CN"/> <impliesc> C_1 C_2 </impliesc> <equalc> C_1 C_2 </equalc> <disjoint> C_1 ... C_n </disjoint>
Role Axioms	<defrole name = "RN"/> <deffeature name = "FN"/> <defattribute name = "AN"/> <impliesr> R_1 R_2 </impliesr> <equalr> R_1 R_2 </equalr> <domain> R C </domain> <range> R C </range> <transitive> R </transitive> <functional> R </functional> <rangeint> A </rangeint> <rangestring> A </rangestring>
Individual Axioms	<defindividual name = "IN"/> <instanceof> I C </instanceof> <related> I_1 R I_2 </related> <value> I A V </value>

Table 7.2: DIG/1.1 TELL language

page 170.

- Clients can send ASK requests using the ASK language presented in Table 7.3 on page 169. An ASK request can contain multiple queries (with different IDs) and must be made in the context of a particular knowledge base (by using the knowledge base URI). The server will respond using the RESPONSE language presented in Table 7.4 on page 170.

The following example shows how to use the TELL, ASK and RESPONSE languages of DIG/1.1.

Example 7.1 DIG/1.1: the TELL, ASK and RESPONSE Statements
The following simplified[3] TELL statements define three classes, Animal, Elephant and AdultElephant that we have seen in Example 3.1 on page 53:

```
<tells ...>
  <releaseKB/>
  <defconcept name = "Animal"/>
  <defconcept name = "Elephant"/>
  <equalc>
```

[3]To save space, we ignore the name space declarations in this example.

Primitive Queries	<allConceptNames/> <allRoleNames/> <allIndividualNames/>
Satisfiability Queries	<satisfiable> C </satisfiable> <subsumes> C_1 C_2 </subsumes> <disjoint> C_1 C_2 </disjoint>
Concept Hierarchy Queries	<parents> C </parents> <children> C </children> <ancestors> C </ancestors> <descendants> C </descendants> <equivalents> C </equivalents>
Role Hierarchy Queries	<rparents> R </rparents> <rchildren> R </rchildren> <rancestors> R </rancestors> <rdescendants> R </rdescendants>
Individual Queries	<instances> C </instances> <types> I </types> <instance> I C </instance> <roleFilters> I R </roleFilters> <relatedIndividuals> R </relatedIndividuals> <toldValues> I A </toldValues>

Table 7.3: DIG/1.1 ASK language

```
    <catom name = "AdultElephant"/>
    <and>
      <catom name = "Elephant"/>
      <intmin val = "21"> <attribute name = "age"/> </intmin>
    </and>
  </equalc>
  <defattribute name = "age"/>
</tells>
```

Then we use the following ASK statement to define two queries: the first asks about the satisfiability of the AdultElephant concept; the second asks for all the concepts subsuming the given description, i.e., Elephants with age greater than 30.

```
<asks ...>
  <satisfiable id = "q1">
    <catom name = "AdultElephant"/>
  </satisfiable>
  <ancestors id = "q2">
    <and>
      <catom name = "Elephant"/>
      <intmin val = "31"> <attribute name = "age"/> </intmin>
    </and>
  </ancestors>
```

Basic	<error/>
Boolean	<true/> <false/>
Concept Set	<conceptSet> <synonyms> C_{11} ... C_{1n} </synonyms> <synonyms> C_{n1} ... C_{nm} </synonyms> </conceptSet>
Role Set	<roleSet> <synonyms> R_{11} ... R_{1n} </synonyms> <synonyms> R_{n1} ... R_{nm} </synonyms> </roleSet>
Individual set	<individualSet> I_1 ... I_n </individualSet>
Individual Pair Set	<individualPairSet> <individualPair> I_1 I_2 </individualPair> <individualPair> J_1 J_2 </individualPair> </individualPairSet>
Values	<sval> s </sval> <ival> i </ival>

Table 7.4: DIG/1.1 RESPONSE language

</asks>

The following RESPONSE statements show a possible response to the query above: the answer to the first query is a boolean, while the answer to the second query is a collection of concepts.

```
<responses ...>
  <true id = "q1"/>
  <conceptSet id = "q2">
    <synonyms>
      <catom name = "Animal"/>
    </synonyms>
    <synonyms>
      <catom name = "Elephant"/>
    </synonyms>
    <synonyms>
      <catom name = "AdultElephant"'/>
    </synonyms>
  </conceptSet>
</responses>
```

♢

Limitations

DIG/1.1 is designed to support DAML+OIL; it has the following limitations w.r.t. OWL DL[4] and OWL-E:

1. The underpinning Description Logic for DIG/1.1 is $\mathcal{SHOIQ}(\mathcal{D}_1)$, while OWL DL and OWL-E correspond to the $\mathcal{SHOIN}(\mathbf{D}^+)$ and $\mathcal{SHOIQ}(\mathcal{G})$ DLs.

   (a) In DIG/1.1, datatype properties are features (functional roles, cf. Definition 2.10 on page 42) while in OWL DL and OWL-E, they are roles (not necessarily with the functional restriction).

   (b) As far as datatype-related concept descriptions are concerned, the DIG/1.1 description language only supports (very limited) unary predicate exists restrictions,[5] while OWL DL provides datatype exists ($\exists T.d$), datatype value ($\forall T.d$), datatype atleast ($\geqslant nT.d$) and datatype atmost ($\leqslant nT.d$) restrictions, and OWL-E provides expressive predicate exists ($\exists T_1, \ldots, T_n.E$), expressive predicate value ($\forall T_1, \ldots, T_n.E$), expressive predicate qualified atleast ($\geqslant mT_1, \ldots, T_n.E$) and expressive predicate qualified atmost ($\leqslant mT_1, \ldots, T_n.E$) restrictions.

   (c) DIG/1.1 does not support the datatype expression axioms that OWL-E provides.

2. DIG/1.1 uses unique name assumption (UNA, cf. Section 2.1.2 on page 33), while neither OWL DL nor OWL-E use this assumption.

3. DIG/1.1 supports only values of the integer and string datatypes, while both OWL DL and OWL-E provide typed literals (cf. Definition 3.7 on page 71).

7.1.2 DIG/OWL-E Interface

To overcome the limitations of DIG/1.1 presented at the end of the last section, we propose DIG/OWL-E, which supports OWL-E (viz. $\mathcal{SHOIQ}(\mathcal{G})$) and extends the DIG/1.1 description language with the following constructors (see Table 7.5):[6]

[4] Section 3.2 of [35] also provides some discussions on issues arising from translating OWL to DIG.

[5] DIG/1.1 only supports a few integer and string unary predicates; cf. Table 7.1 on page 167.

[6] Notations of Table 7.5: `V`$_i$ are typed literals, `E`$_i$ are datatype expressions or datatype expression names, `U`$_i$ are unary datatype predicates or their relativised negations, and `T`$_i$ are concrete roles.

Typed Literals	<typedValue lxform = "L" datatype = "DN"/>
Datatype Expressions	<dttop/> <dtbottom/> <predicate name = "PN"/> <dtnot name = "PN"/> <vset> V_1 ... V_n </vset> <dtand> E_1 ... E_n </dtand> <dtor> E_1 ... E_n </dtor> <dtdomain> U_1 ... U_n </dtdomain> <dtexpression name = "EN"/>
Concrete Roles	<dtratom name = "TN"/>
Datatype Expression-related Concept Descriptions	<dtsome> T_1 ... T_n E </dtsome> <dtall> T_1 ... T_n E </dtall> <dtatmost num = "n"> T_1 ... T_n E </dtatmost> <dtatleast num = "n"> T_1 ... T_n E </dtatleast>

Table 7.5: New constructors in the DIG/OWL-E description language

1. **Typed Literals** (cf. Definition3.7 on page 71): A <typedValue> element represents a typed literal; e.g., <typedValue lxform = "s" datatype = "u"/> represents the typed literal "s"^^u.

2. **Datatype Expressions** (cf. Definition 5.12 on page 110):

   (a) <dttop/> and <dtbottom/> correspond to rdfs:Literal and owlx:Datatype-Bottom in OWL-E, respectively (cf. Definition 5.4 on page 104).

   (b) A <predicate> element introduces a predicate URI reference, and a <dtnot> element represents a negated predicate URI reference.

   (c) A <vset> element represents an enumerated datatype (cf. Definition 3.11 on page 74), i.e., a datatype defined by enumerating all its member typed literals.

   (d) A <dtnot> element represents the relativised negation of a unary supported predicate URIref (cf. Definition 5.12 on page 110).

   (e) The <dtand>, <dtor> and <dtdomain> elements corresponds to the `and`, `or` and `domain` constructors defined in Definition 5.12, respectively.

3. **Concrete Roles**:[7] A <dtratom> element represents a concrete role. Note that attributes (<attribute> elements) are simply functional concrete roles.

[7] We use the term 'concrete roles' following [75]. In OWL DL and OWL-E, they are also called 'datatype properties'; cf. Section 3.3.3 on page 75.

4. **Datatype Expression-related Concept Descriptions** (cf. Table 5.2): <dtsome>, <dtall>, <dtatleast> and <dtatmost> elements corresponds to expressive predicate exists restriction ($\exists T_1, \ldots, T_n.E$), expressive predicate value restriction ($\forall T_1, \ldots, T_n.E$), expressive predicate qualified atleast restriction ($\leqslant mT_1, \ldots, T_n.E$) and expressive predicate qualified atmost restriction ($\leqslant mT_1, \ldots, T_n.E$), respectively; note that E can either be a datatype expression or a datatype expression name.

Example 7.2 A Concept Description in DIG/OWL-E

The AdultElephant concept can be defined in DIG/OWL-E as follows (cf. the DIG/1.1 version of the AdultElephant concept in Example 7.1):

```
<tells ...>
  <equalc>
    <catom name = "AdultElephant"/>
    <and>
      <catom name = "Elephant"/>
      <dtsome>
        <attribute name = "age"/>
        <predicate id = "owlx:integerGreaterThanx=20"/>
      </dtsome>
    </and>
  </equalc>
  <defconcept name = "Elephant"/>
  <defattribute name = "age"/>
</tells>
```

where owlx:integerGreaterThanx=20 is the URIref for the integer predicate $>^{int}_{[20]}$. Here we use the <dtsome> construct, instead of the <intmin> construct; the main benefit is that we can now use arbitrary predicate URIrefs, or even datatype expressions. ♢

DIG/OWL-E extends the DIG/1.1 TELL language by introducing the following new axioms (see Table 7.6 on page 174):

1. **Datatype Expression Axiom** (cf. Section 6.1): A <defdtexpression> element represents a datatype expression axiom, which introduces a name "EN" for a datatype expression E. Therefore, the datatype expression name can be used in datatype expression-related concept descriptions and the concrete role range axioms.

2. **Concrete Role Axioms**: Many of the concrete role axioms are very similar to abstract role axioms. We could have modified the existing abstract role axioms

Datatype Expression Axiom	<defdtexpression name = "EN"> E </defdtexpression>
Concrete Role Axioms	<defcrole name = "RN" /> <impliescr> T_1 T_2 </impliescr> <equalcr> T_1 T_2 </equalcr> <crdomain> T C </crdomain> <crrange> T E </crrange> <crfunctional> T </crfunctional>
Individual Axioms	<sameindividual> I_1 ... I_n </sameindividual> <diffindividual> I_1 ... I_n </diffindividual>

Table 7.6: New axioms in the DIG/OWL-E TELL language

to accommodate the concrete role axioms. We propose otherwise in order to maintain backward compatibility. Another advantage is that we can easily disallow asserting that an abstract role is a sub-role of (or equivalent to) a concrete one, or the other way around. Note that the concrete role range axiom (represented by a <crrange> element) is quite different from the abstract role range axiom (represented by a <range> element) in that the range in the former one is a datatype expression or a name of a datatype expression, instead of a concept.

3. **Same and Different Individual Axioms**: A <sameindividual> element asserts that two individual names are interpreted as the same individual, while a <diffindividual> element asserts that two individual names are interpreted as different individuals. Note that these two individual axioms are convenient, but not necessary, for people to use OWL and OWL-E.[8]

Example 7.3 A Datatype Expression Axiom in DIG/OWL-E

The sumLessThan15 customised datatype predicate mentioned in Example 1.5 on page 22 can be defined by the following DIG/OWL-E datatype expression axiom:

```
<tells>
  <defdtexpression name = "sumNoGreaterThan15">
    <dtand>
      <predicate id = "owlx:integerAddition" />
      <dtdomain>
        <dtnot id = "owlx:integerGreaterThanx=15" />
        <predicate id = "owlx:integerGreaterThanx=0" />
        <predicate id = "owlx:integerGreaterThanx=0" />
        <predicate id = "owlx:integerGreaterThanx=0" />
```

[8] cf. Section 2.1.2 on page 33.

Individual Query	<relatedIndividualValues> T </relatedIndividualValues>

Table 7.7: New queries in the DIG/OWL-E ASK language

Individual Value Pair Sets	<individualValuePairSet> <indvalPair> I_1 V_1 </indvalPair> <indvalPair> I_1 V_2 </indvalPair> </individualValuePairSet>
Typed Value Sets	<typedValueSet> V_1 ... V_n </typedValueSet>

Table 7.8: New responses in the DIG/OWL-E RESPONSE language

```
      </dtdomain>
    </dtand>
  </defdtexpression>
</tells>
```

where owlx:integerAddition is the URIref of the integer predicate $+^{int}$ mentioned in Section 5.1 on page 102. The datatype expression sumNoGreaterThan15 is a conjunction, where the first conjunct is the predicate URIref owlx:integerAddition, and the second conjunct is a domain constraint, which sets the domains of all its four arguments: the first one (corresponding to the sum of addition) should be integers that are no greater than 15, and the rest are positive integers. Based on sumNoGreaterThan15, we can define the SmallItem concept as follows:

```
<tells>
  <defconceptname = "Item"/>
  <defconceptname = "SmallItem"/>
  <equalc>
  <catom name = "SmallItem"/>
    <and>
    <catom name = "Item"/>
      <dtsome>
        <dtratom name = "hlwSumInCM"/>
        <dtratom name = "heightInCM"/>
        <dtratom name = "lengthInCM"/>
        <dtratom name = "widthInCM"/>
        <dtexpression name = "sumNoGreaterThan15"/>
      </dtsome>
    </and>
  </equalc>
</tells>
```

♢

DIG/OWL-E extends the DIG/1.1 `ASK` language by introducing a new form of individual query. With a `<relatedIndividualValues>` element, clients can request the instances (pairs of individual and typed values) of a concrete role. Accordingly, DIG/OWL-E extends the DIG/1.1 `RESPONSE` language (see Table 7.8) with individual value pair sets (`<individualValuePairSet>` elements) and typed value sets. Furthermore, a response to an identifier request in DIG/OWL-E should include the list of base datatype URIrefs, supported predicate URIrefs and the supported datatype expression constructors.

7.2 Architecture

We now propose a revised DIG framework architecture, which is designed to be more flexible and compatible with the OWL DL and OWL-E languages. The proposed framework architecture (see Figure 7.2) is different from the existing DIG/1.1 framework architecture we introduced in Section 7.1.1 (see Figure 7.1 on page 166) mainly in the following aspects:

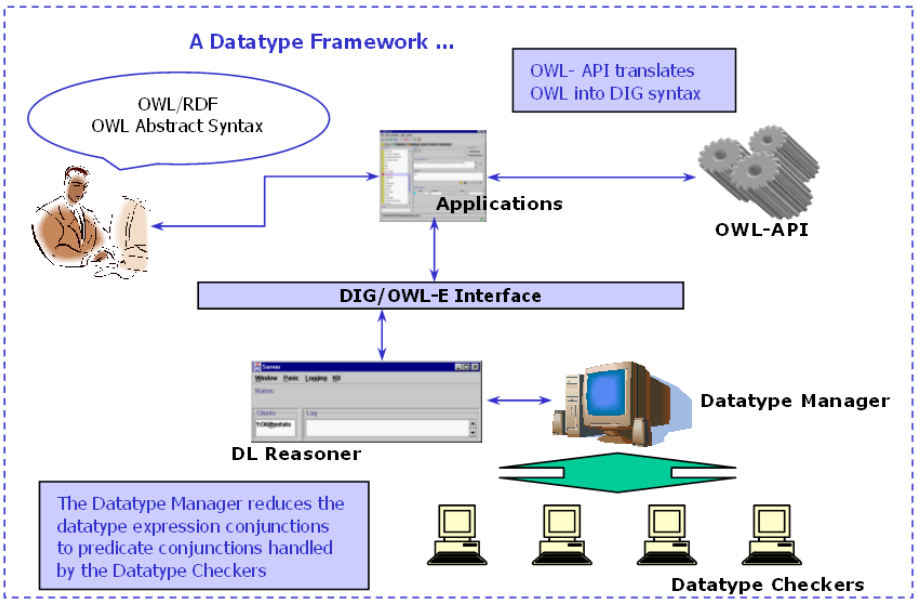

Figure 7.2: Framework architecture

1. We use DIG/OWL-E instead of DIG/1.1 as the general DL reasoner interface in the new architecture.

2. The datatype reasoning is not performed in the DL reasoner, but by one or more datatype reasoners. This architecture makes the framework more flexible, as when new datatypes and predicates are needed, we only need to update our datatype reasoners, but not the DL reasoner; in other words, we have a general-purpose DL reasoner.

There are two kinds of datatype reasoners in the framework. The first kind of datatype reasoner is called a datatype manager, which provides decision procedures to reduce the satisfiability problem of datatype expression conjunctions to the satisfiability problems of various datatype predicate conjunctions. The second kind of datatype reasoner is called a datatype checker. A datatype checker decides the satisfiability problem of datatype predicate conjunctions, where the datatype predicates are defined over a base datatype in a datatype group. More technical details of datatype managers and datatype checkers will be presented in the next section.

As shown in Figure 7.2, when a client sends an identification request to the DIG/OWL-E server, the server returns the names and versions of the DL reasoner, the datatype manager and the datatype checkers. In addition, the server returns the description language that the DL reasoner supports, the datatype expressions that the datatype manager supports and the base datatype URIrefs as well as supported predicate URIrefs that the datatype checkers provide.

The client can then upload the knowledge base using the `TELL` language and query using the `ASK` language. To answer the query, the DL reasoner runs as usual and depends on the datatype manager to decide the satisfiability problem of datatype expression conjunctions whenever necessary. The datatype manager reduces the datatype expression conjunctions to predicate conjunctions and then passes them to the proper datatype checkers to verify their satisfiability. Finally, the DL reasoner uses the `RESPONSE` language to return the answer to the client.

Whenever we need to support a new form of datatype expression, we update our datatype manager; whenever we need to provide some new datatype predicates that are defined over a new base datatype, we add a new datatype checker. In either case, we do not have to update the DL reasoner.

7.3 Datatype Reasoners

The soundness of our proposed framework depends on the proper design of the DL reasoner and datatype reasoners. In Chapter 6, we have presented the tableaux algorithms for the $\mathcal{G}$-family DLs, which allows us to make use of datatype reasoners to answer expression conjunctions of the form (6.1). In this section, we present how to design datatype managers and datatype checkers.

7.3.1 Datatype Checkers

We start with datatype checkers. Let $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_\mathcal{G}, \mathsf{dom})$ be a datatype group, $d \in \mathbf{D}_\mathcal{G}$ a base datatype URIref. According to Lemma 5.9 on page 108, if the corresponding concrete domain of w is admissible, we can use a datatype checker for d to decide the satisfiability of finite predicate conjunctions over $\mathsf{sub\text{-}group}(w, \mathcal{G})$. Formally, a datatype checker for d is defined as follows.

Definition 7.1. (**Datatype Checker**) Given a conforming datatype group $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_\mathcal{G}, \mathsf{dom})$ and a base datatype URIref $d \in \mathbf{D}_\mathcal{G}$, a *datatype checker for* d, written as $\mathtt{DC}_d$, is a program that takes as input a finite predicate conjunction $\mathscr{C}$

$$\mathscr{C} = \bigwedge_{j=1}^{k} p_j(v_1^{(j)}, \ldots, v_{n_j}^{(j)}),$$

where $p_j \in \mathsf{sub\text{-}group}(d, \mathcal{G})$ with arity n_j and $v_i^{(j)}$ are variables, and answers *satisfiable* if $\mathscr{C}$ is satisfiable and *unsatisfiable* otherwise. $\diamond$

In general, we can reuse existing constraint solvers for the corresponding predicate conjunctions as our datatype checkers.

7.3.2 Datatype Manager

As shown in the framework architecture (see Figure 7.2), a datatype manager works between a DL reasoner and a set of datatype checkers. We assume that datatype managers defined as follows support all forms of OWL-E datatype expressions.

Intuitively, a datatype manager transforms an input datatype expression into a disjunction of predicate conjunctions and then divides each of the predicate conjunction into several sub-conjunctions. Each of these sub-conjunctions contains predicates of certain base datatype. If there exist variables being used across these sub-conjunctions,

then the corresponding disjunct is *unsatisfiable*, as the value spaces of base datatypes are disjoint with each other; otherwise, the datatype manager can send these sub-conjunctions to appropriate datatype checkers to decide their satisfiabilities. If all the sub-conjunctions of a predicate conjunction is *satisfiable*, then it is *satisfiable*, and if any one of the predicate conjunction is *satisfiable*, then the input datatype expression is *satisfiable*.

Definition 7.2. (**Datatype Manager**) Given a conforming datatype group $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_\mathcal{G}, \mathsf{dom})$, the *datatype manager for* $\mathcal{G}$, written as $\mathtt{DM}_\mathcal{G}$, is a program that takes as input a datatype query Ψ of the form of (6.1) and answers *satisfiable* if Ψ is satisfiable and *unsatisfiable* otherwise.

Let $p \in \Phi_\mathcal{G}, q \notin \Phi_\mathcal{G}, d \in \mathbf{D}_\mathcal{G}, d_1, \ldots, d_n \in \mathsf{sub\text{-}group}(d, \mathcal{G})$ and $a(d_i) = 1$ for all $1 \leq i \leq n$, and P, Q are $\mathcal{G}$-datatype expression with the same arity. $\mathtt{DM}_\mathcal{G}$ decides the satisfiability problem of a $\mathcal{G}$-datatype expression conjunction Ψ in the following steps:

1. $\mathtt{DM}_\mathcal{G}$ transforms Ψ into Ψ_1 by eliminating the negation constructor $\neg$ for datatype expressions within Ψ as follows :

$$\begin{aligned}
\neg p &\equiv \overline{p} \sqcup \overline{(\underbrace{d, \ldots, d}_{n \text{ times}})} \text{ (where } \mathsf{dom}(p) = (\underbrace{d, \ldots, d}_{n \text{ times}})\text{)},\\
\neg \overline{p} &\equiv p \sqcup \overline{(\underbrace{d, \ldots, d}_{n \text{ times}})} \text{ (where } \mathsf{dom}(p) = (\underbrace{d, \ldots, d}_{n \text{ times}})\text{)},\\
\neg q &\equiv \overline{q}\\
\neg \overline{q} &\equiv q
\end{aligned}$$

$$\begin{aligned}
\neg(d_1 \ \ldots \ d_n) &\equiv \overline{(d_1 \ \ldots \ d_n)}\\
\neg\overline{(d_1 \ \ldots \ d_n)} &\equiv (d_1 \ \ldots \ d_n)\\
\neg(P \sqcap Q) &\equiv \neg P \sqcup \neg Q\\
\neg(P \sqcup Q) &\equiv \neg P \sqcap \neg Q.
\end{aligned}$$

2. $\mathtt{DM}_\mathcal{G}$ transforms Ψ_1 into Ψ_2 by rewriting $(d_1, \ldots, d_n)$ and $\overline{(d_1, \ldots, d_n)}$ in Ψ_1 as follows:

   - $(d_1, \ldots, d_n)(v_1, \ldots, v_n) \equiv d_1(v_1) \sqcap \ldots \sqcap d_n(v_n)$;

- $\overline{(d_1, \ldots, d_n)}(v_1, \ldots, v_n) \equiv C_{d_1}(v_1) \sqcup \ldots \sqcup C_{d_n}(v_n)$, and

$$C_{d_i}(v_i) = \begin{cases} \overline{d}(v_i) & \text{if } d_i = d, \\ \overline{d_i}(v_i) \sqcup \overline{d}(v_i) & \text{otherwise.} \end{cases}$$

  where $d = \mathsf{dom}(d_i)$ for $1 \leq i \leq n$.

3. $\mathtt{DM}_{\mathcal{G}}$ transforms Ψ_2 into Ψ_3 by eliminating the negated predicate names $\overline{p}$ for $p \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}$ as follows:

$$\overline{p} \equiv e \quad \text{(where } E(e) = E(\overline{p})\text{)}.$$

4. $\mathtt{DM}_{\mathcal{G}}$ transforms Ψ_3 into Ψ_4 by rewriting $\neq (v_{i1}, \ldots, v_{in};\, v_{j_1}, \ldots, v_{j_n})$ as follows:

$$\neq (v_{i1}, \ldots, v_{in};\, v_{j_1}, \ldots, v_{j_n}) \;\equiv\; \neq (v_{i_1}, v_{j_1}) \sqcup \ldots \sqcup \neq (v_{i_n}, v_{j_n}).$$

5. $\mathtt{DM}_{\mathcal{G}}$ transforms Ψ_4 into its disjunctive normal form Ψ_5.

6. $\mathtt{DM}_{\mathcal{G}}$ checks each disjunct M of Ψ_5 as follows:

   (a) Let $v_1, \ldots, v_n$ be a tuple of variables and p a predicate URIref. If both $p(v_1, \ldots, v_n)$ and $\overline{p}(v_1, \ldots, v_n)$ are in M, then M is *unsatisfiable*.

   (b) Let v be a variable. If v occurs as an argument of both p and $\overline{d}$ in M, and $p \in \mathsf{sub\text{-}group}(d, \mathcal{G})$, then M is *unsatisfiable*.

   (c) Let v be a variable. If v occurs as an argument of both p_1 and p_2 in M, where $p_1 \in \mathsf{sub\text{-}group}(d_1, \mathcal{G}), p_2 \in \mathsf{sub\text{-}group}(d_2, \mathcal{G})$ and $d_1 \neq d_2$, then M is *unsatisfiable*.

   (d) Otherwise, since $\Phi_{\mathcal{G}} = \bigcup_{\forall d \in \mathbf{D}_{\mathcal{G}}} \mathsf{sub\text{-}group}(d, \mathcal{G})$, $\mathtt{DM}_{\mathcal{G}}$ disregards the unsupported predicate URI references and splits M into conjunctions N, $M_{d_1}, \ldots, M_{d_k}$, where k is the number of sub-groups used in M and $\{d_1, \ldots, d_k\} \subseteq \mathbf{D}_{\mathcal{G}}$, such that

   - N contains all the value inequality predicate conjunction;
   - for each p in M_{d_h}, $p \in \mathsf{sub\text{-}group}(d_h, \mathcal{G})$.

   For each $\neq (v_{i_s}, v_{j_s})$, if both the variables v_{i_s}, v_{j_s} occurs in M_{d_h}, $\mathtt{DM}_{\mathcal{G}}$ will add $\neq_{d_h} (v_{i_s}, v_{j_s})$ as a conjunct into M_{d_h}.

$\mathtt{DM}_{\mathcal{G}}$ then uses the corresponding datatype checker $\mathtt{DC}_{d_h}$ to decide the satisfiability of M_{d_h}. Finally, M is *satisfiable* if all datatype checkers return *satisfiable*; $\mathtt{DM}_{\mathcal{G}}$ returns *satisfiable* if one of the disjuncts M of Ψ_5 is *satisfiable*; otherwise, $\mathtt{DM}_{\mathcal{G}}$ returns *unsatisfiable*.

Lemma 7.3. *Given a conforming datatype group $\mathcal{G} = (\mathbf{M}_p, \mathbf{D}_{\mathcal{G}}, \mathsf{dom})$, the type manager $\mathtt{DM}_{\mathcal{G}}$ decides the satisfiability of datatype queries Ψ of the form of (6.1).*

Proof: In Definition 7.2, the first five steps use equivalent transformations to rewrite the input $\mathcal{G}$-predicate expression conjunction Ψ into a disjunction of predicate conjunctions Ψ_5.

In step 1, the equivalent transformations are based on the semantics of negated predicate names and datatype expressions.

- Let p be a supported predicate URIref and $p \in \mathsf{sub\text{-}group}(d, \mathcal{G})$, we have

$$\begin{aligned}(\neg p)^{\mathbf{D}} &= (\Delta_{\mathbf{D}})^n \setminus p^{\mathbf{D}} \\ &= (\Delta_{\mathbf{D}})^n \setminus E(p) \\ &= ((\Delta_{\mathbf{D}})^n \setminus (\mathsf{dom}(p))^{\mathbf{D}}) \cup ((\mathsf{dom}(p))^{\mathbf{D}} \setminus E(p)) \\ &= \overline{(\underbrace{d, \ldots, d}_{n \text{ times}})}^{\mathbf{D}} \cup \overline{p}^{\mathbf{D}},\end{aligned}$$

  so we have $\neg p \equiv \overline{p} \sqcup \overline{(\underbrace{d, \ldots, d}_{n \text{ times}})}$. Similarly, we have $\neg \overline{p} \equiv p \sqcup \overline{(\underbrace{d, \ldots, d}_{n \text{ times}})}$.

- Let q be an un-supported predicate URIref, i.e., $q \notin \Phi_{\mathcal{G}}$. According to Definition 5.4, we have $\overline{q}^{\mathbf{D}} = (\Delta_{\mathbf{D}})^n \setminus q^{\mathbf{D}}$, so $\overline{q} \equiv \neg q$.

- According to Definition 5.12, we have $\neg(d_1, \ldots, d_n)^{\mathbf{D}} = (\Delta_{\mathbf{D}})^n \setminus (d_1, \ldots, d_n)^{\mathbf{D}} = \overline{(d_1, \ldots, d_n)}^{\mathbf{D}}$, so we have $\neg(d_1, \ldots, d_n) \equiv \overline{(d_1, \ldots, d_n)}$.

- Let P, Q be $\mathcal{G}$-datatype expressions with the same arity. According to De Morgan's Law, we have $\neg(P \sqcap Q) \equiv \neg P \sqcup \neg Q$ and $\neg(P \sqcup Q) \equiv \neg P \sqcap \neg Q$.

In step 2, let $d \in \mathbf{D}_{\mathcal{G}}$, $d_1, \ldots, d_n \in \mathsf{sub\text{-}group}(d, \mathcal{G})$ and $a(d_i) = 1$ for each $1 \leq i \leq n$. By definition, we have $(d_1, \ldots, d_n)(v_1, \ldots, v_n) \equiv d_1(v_1) \sqcap \ldots \sqcap d_n(v_n)$. As for its negation, we have

$$\begin{aligned}& \overline{(d_1, \ldots, d_n)}(v_1, \ldots, v_n) \\ \equiv\ & \neg(d_1, \ldots, d_n)(v_1, \ldots, v_n) \\ \equiv\ & \neg d_1(v_1) \sqcup \ldots \sqcup \neg d_n(v_n).\end{aligned}$$

There are two possibilities here:

- if $d_i = d$, then $(\neg d_i)^{\mathbf{D}} = (\neg d)^{\mathbf{D}} = \overline{d}^{\mathbf{D}}$, hence we have $\neg d_i \equiv \overline{d}$;
- otherwise $(\neg d_i)^{\mathbf{D}} = \Delta_{\mathbf{D}} \setminus d_i^{\mathbf{D}} = (\Delta_{\mathbf{D}} \setminus d^{\mathbf{D}}) \cup (d^{\mathbf{D}} \setminus d_i^{\mathbf{D}}) = \overline{d}^{\mathbf{D}} \cup \overline{d_i}^{\mathbf{D}}$; hence we have $\neg d_i \equiv \overline{d} \sqcup \overline{d_i}$.

In step 3, the equivalent transformations are guaranteed by the condition 2 of a conforming datatype group. As for step 4 and 5, the equivalent transformations are obvious.

Finally step 6, which checks every disjunct M of Ψ_5, where M is simply a predicate conjunction.

- Sub-step (6a) is obvious, a tuple of variables cannot satisfy both p and $\overline{p}$.
- In sub-step (6b), assume we have a solution δ for M. Since $p \in \mathsf{sub\text{-}group}(d, \mathcal{G})$, we have $\delta(v) \in d^{\mathbf{D}}$; due to $\overline{d}(v)$, we have $\delta(v) \in \overline{d}^{\mathbf{D}}$. Since $d^{\mathbf{D}} \cap \overline{d}^{\mathbf{D}} = \emptyset$, we have a contradiction; therefore, we have no solution for M, i.e., M is *unsatisfiable*.
- In sub-step (6c), assume we have a solution δ for M. Since $p_1 \in \mathsf{sub\text{-}group}(d_1, \mathcal{G})$, we have $\delta(v) \in d_1^{\mathbf{D}}$; similarly as $p_2 \in \mathsf{sub\text{-}group}(d_2, \mathcal{G})$ we have $\delta(v) \in d_2^{\mathbf{D}}$. According to Definition 5.5 on page 104, $d_1^{\mathbf{D}} \cap d_2^{\mathbf{D}} = \emptyset$, hence there is a contradiction; therefore, we have no solution for M, i.e., M is *unsatisfiable*.
- In sub-step (6d), $\mathtt{DM}_{\mathcal{G}}$ only concerns the supported predicates because the only possible contradictions caused by unsupported predicate URIrefs have been considered in sub-step (6a). $\mathtt{DM}_{\mathcal{G}}$ splits M into $M_{d_1}, \ldots, M_{d_k}$ and N. For each inequality constraint $\neq (v_{i_s}, v_{j_s})$, there are two possibilities:
  - if both v_{i_s} and v_{j_s} occur in same M_{d_h}, $\mathtt{DM}_{\mathcal{G}}$ adds $\neq_{d_h} (v_{i_s}, v_{j_s})$ into M_{d_h} to ensure, for any solution δ_h of M_{d_h}, $\delta_h(v_{i_s}) \neq \delta_h(v_{i_s})$;
  - if v_{i_s} and v_{j_s} do not occur in some M_{d_h}, they will not be mapped to the same data value, due to the disjointness of the interpretations of base datatypes.

  Finally, $\mathtt{DM}_{\mathcal{G}}$ can then uses the corresponding datatype checker $\mathtt{DC}_{d_h}$ to decide the satisfiability of each M_{d_h}. Since $M_{d_1}, \ldots, M_{d_k}$ are conjuncts of M, M is satisfiable iff all the corresponding datatype checkers return *satisfiable*. Finally, $\mathtt{DM}_{\mathcal{G}}$ returns *satisfiable* if one of the disjuncts M of Ψ_5 is satisfiable. □

7.4 Flexibility

We end the chapter by a conclusion of the flexibility of our framework in the following points of view.

Users The framework enables the users to use customised datatypes and datatype predicates. In addition, it provides minimum checking for unsupported datatype predicates.

DIG/OWL-E interface Our general DL API does not need to be updated when new datatype predicates are supported by datatype reasoners.

DL reasoner In the framework, the DL reasoner can implement $\mathcal{G}$-augmented tableaux algorithms for a wide range of $\mathcal{G}$-combined DLs, including express ones like $\mathcal{SHIQ}(\mathcal{G})$, $\mathcal{SHOQ}(\mathcal{G})$ and $\mathcal{SHIO}(\mathcal{G})$, and less express but more efficient ones like the datatype group extensions of DL-Lite [26] and $\mathcal{ELH}$ [20]. To support new DLs, we only need to upgrade the DL reasoner.

Datatype Manager To support new forms of datatype expressions, we only need to upgrade the datatype manager and ensure the datatype query of the form of (6.1) are still decidable.

Datatype Checkers To support datatype predicates of some new base datatypes, we only have to add new datatype checkers.

Chapter Achievements

- DIG/OWL-E extends DIG/1.1 by using $\mathcal{SHOIQ}(\mathcal{G})$ as the underpinning Description Logic, which is expressive enough to cover DAML+OIL, OWL DL and OWL-E.
- DIG/OWL-E supports $\mathcal{G}$-datatype expressions and allows the use of arbitrary datatype predicates in $\mathcal{G}$-datatype expressions.
- Our proposed framework is highly extendable to support new datatype predicates, new forms of datatype expressions and new decidable Description Logics.

Chapter 8

Implementation

Chapter Aims

- To present a prototype as a test bed for verifying the validity of our approach and for suggesting new directions to investigate.
- To show the flexibility of the framework architecture.
- To describe the matchmaking use case which exploits our prototype.

Chapter Plan

This chapter presents an implementation of the tableaux algorithm for $\mathcal{SHIQ}(\mathcal{G})$ presented in Chapter 6 and the datatype manager algorithm presented in Chapter 7.

8.1 A Datatype Extension of FaCT

The goal of this section is to describe a datatype extension of the FaCT reasoner [60].[1] FaCT was the first sound and complete DL system to demonstrate the usefulness of expressive Description Logics for developing practical applications [100]. Our implementation is designed for two purposes. Firstly, it is meant to be a light-weight test bed for the $\mathcal{G}$-augmented tableaux algorithm of the $\mathcal{SHIQ}(\mathcal{G})$ DL presented in Definition 6.18. Secondly, it aims to show the flexibility of the framework architecture

[1] 'FaCT' for 'Fast Classification of Terminologies'.

presented in Chapter 7. As a concept-proof prototype, it provides neither an implementation of the DIG/OWL-E interface nor any optimisations for datatype reasoning. In other words, performance is not its main concern.

8.1.1 System Overview

As per the framework presented in Chapter 7, our prototype is actually a hybrid reasoner that has three highly independent components: (i) a DL reasoner that is built based on the FaCT reasoner and supports the $\mathcal{SHIQ}(\mathcal{G})$ DL (see Section 6.2.2), (ii) a datatype manager which decides the satisfiability of datatype expression conjunctions, and (iii) two simple datatype checkers for integers and strings. The DL reasoner asks the datatype manager to check the satisfiability of datatype queries of the form of (6.1). The datatype manager asks the two datatype checkers to check the satisfiability of predicate conjunctions of the form of (5.6). All the three components of our system is implemented in Lisp, although in principle they do *not* have to be implemented in same the programming language.

8.1.2 Extended DL Reasoner

The extended FaCT DL reasoner supports datatype group-based concept descriptions. It implements the $\mathcal{G}$-rules described in Figure 6.2 on page 143. We usually apply these rules (in the following order: $\exists_P$-rule, $\geqslant_P$-rule, $\forall_P$-rule, $choose_p$-rule and $\leqslant_P$-rule) on an abstract node just before we further check its abstract successors and query the datatype manager to check resulting datatype constraints. It may need to query the datatype manager again if new datatype constraints are added later, e.g., from some of its abstract successors via some inverse roles.

The DL reasoner is independent of the forms of datatype expressions that the datatype manager supports. It does not have to understand the syntax of datatype expressions, and it simply leaves them untouched and passes them to the datatype manager. Therefore, the DL reasoner does not have to be modified even if the datatype manager is upgraded to support some forms of new datatype expressions.

We extend the FaCT syntax by introducing datatype expressions (see Table 8.1) and datatype expression-related concept descriptions (see Table 8.2), where we use a positive integer n to indicate the number of concrete roles used in the concept descriptions.

The syntax of TBox and RBox axioms remains the same as FaCT. Users can now

Extended FaCT Syntax	OWL-E Abstract Syntax
TOPD	rdfs:Literal
BOTTOMD	owlx:DatatypeBottom
(and E_1 ... E_n)	and($E_1, \ldots, E_n$)
(or E_1 ... E_n)	or($E_1, \ldots, E_n$)
(neg p)	neg(p)
(domain d_1 ... d_n)	domain($d_1, \ldots, d_n$)

Table 8.1: FaCT datatype expressions

Extended FaCT Syntax	DL Standard Syntax
TOP	$\top$
BOTTOM	$\bot$
(and $\mathsf{C}_1, \ldots, \mathsf{C}_n$)	$\mathsf{C_1} \sqcap \ldots \sqcap \mathsf{C_n}$
(or $\mathsf{C}_1, \ldots, \mathsf{C}_n$)	$\mathsf{C_1} \sqcup \ldots \sqcup \mathsf{C_n}$
(not C)	$\neg\mathsf{C}$
(some R C)	$\exists R.\mathsf{C}$
(all R C)	$\forall R.\mathsf{C}$
(atleast m R C)	$\geqslant mR.\mathsf{C}$
(atmost m R C)	$\leqslant mR.\mathsf{C}$
(dt-some n T_1 ... T_n E)	$\exists T_1, \ldots, T_n.E$
(dt-all n T_1 ... T_n E)	$\forall T_1, \ldots, T_n.E$
(dt-atleast n m T_1 ... T_n E)	$\geqslant m T_1, \ldots, T_n.E$
(dt-atmost n m T_1 ... T_n E)	$\leqslant m T_1, \ldots, T_n.E$

Table 8.2: FaCT concepts

use datatype expression-related concept descriptions in TBox axioms. Note that the set of abstract role names and the set of concrete role names should be disjoint, otherwise the system will report an error. Users can define concrete role inclusion axioms and functional axioms. Note that if users use an abstract role and a concrete role in a role inclusion axiom, the system will report an error. The syntax of TBox queries remains the same as FaCT too; i.e., this datatype extension of FaCT provides concept satisfiability, concept subsumption and classification checking.

8.1.3 Datatype Reasoning Components

The datatype reasoning components of the hybrid reasoner include a datatype manager and two simple datatype checkers (concrete domain reasoners). The datatype manager

reduces the satisfiability problem of datatype expression conjunctions to the satisfiability problem of predicate conjunctions that the datatype checkers can handle.

The datatype manager is independent of the kinds of datatype predicates that the datatype checkers support. Theoretically it can work with an arbitrary set of datatype checkers, as long as the datatype checkers satisfy the following conditions.

1. the domains of the base datatypes that the datatype checkers support are pairwise disjoint;
2. each datatype checker provides the following registration information:[2]
   (a) its satisfiability checking function,
   (b) its base datatype URIref,
   (c) the inequality predicate URIref for its base datatype,
   (d) the set of supported predicate URIrefs for its base datatype;
3. each registered datatype checker supports the syntax of predicate conjunctions that the datatype manager uses in its queries.[3]

Table 8.3 lists the registration information of the two datatype checkers implemented in our hybrid reasoner.

	Datatype Checker ♯1	**Datatype Checker ♯2**
Sat. Checking Func.	sint-dcf-sat	sstr-dcf-sat
Base Datatype URIref	xsd:integer	xsd:string
Inequality Pred. URIref	owlx:integerInequality	owlx:stringInequality
Other Supported Pred. URIrefs	owlx:integerEquality owlx:integerLessThan owlx:integerGreaterThan owlx:integerLessThanOrEqualTo owlx:integergreaterThanOrEqualTo owlx:integerInequalityx=n owlx:integerEqualityx=n owlx:integerLessThanx=n owlx:integerGreaterThanx=n owlx:integerLessThanOrEqualTox=n owlx:integergreaterThanOrEqualTox=n	owlx:stringEquality

Table 8.3: Registration information of our datatype checkers

[2]Here we assume that each datatype checker supports only one base datatype, but it is easy to extend it to the case where a datatype checker supports multiple base datatypes.

[3]In general, it should not be difficult to translate the syntax between the datatype manager and datatype checkers.

It is straightforward, but important, to observe that the system is very flexible. For example, to support new datatypes and predicates, we simply need to add datatype checkers that satisfy the above three conditions.

8.2 Case Study: Matchmaking

We invite the reader to consider a use case of our hybrid reasoner. Section 1.3 has shown an example of using datatype reasoning in matchmaking. This section will further describe how to use our prototype to support matchmaking.

8.2.1 Matchmaking

Let us consider the following scenario: agents advertise services with their capabilities through a registry and query the registry for services with specified capabilities. Matchmaking is a process that takes a query as input and return all advertisements which may potentially satisfy the capabilities specified in the query.

We can use the $\mathcal{SHIQ}(\mathcal{G})$ DL to describe service capabilities for both advertisement and query. More precisely, the capability (either in advertisements or queries) of a service can be represented as an OWL-E class or class restriction, e.g. the capability that memory size should be either 256Mb or 512Mb, can be represented the datatype expression-related concept $\exists memoryUnitSizeInMb.(=_{256} \vee =_{512})$.

Usually, we are not only interested in finding the exact match, viz., there could be several degrees of matching. Following [84], we consider five levels of matching:

1. **Exact** If the capabilities of an advertisement A and a request R are equivalent classes, we call it an exact match, noted as $C_A \equiv C_R$.

2. **PlugIn** If the capability of a request R is a sub-class of that of an advertisement A, we call it a PlugIn match, noted as $C_R \sqsubseteq C_A$.

3. **Subsume** If the capability of a request R is a super-class of that of an advertisement A, we call it a Subsume match, noted as $C_A \sqsubseteq C_R$.

4. **Intersection** If the intersection of the capabilities of an advertisement A and a request R are satisfiable, we call it a Intersection match, noted as $\neg(C_A \sqcap C_R \sqsubseteq \bot)$.

5. **Disjoint** Otherwise, we call it a Disjoint (failed) match , noted as $C_A \sqcap C_R \sqsubseteq \bot$.

8.2.2 Working Examples

To gain a further insight into the above five levels of matching, it is often helpful to have some working examples. Suppose, in a scenario of computer selling, that an agent would like to buy a PC with the following capabilities:

- the $processor$ must be Pentium4;
- the $memoryUnitSizeInMb$ must be 128;
- the $priceInPound$ must be less than 500.

This can be represented by the following $\mathcal{SHIQ}(\mathcal{G})$ concept:

$$\begin{aligned} C_{R1} \equiv\ & \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \\ & \exists memoryUnitSizeInMb. =^{int}_{[128]} \sqcap \exists priceInPound. <^{int}_{[500]} \end{aligned}$$

$$\begin{aligned}
\text{Exact match: } C_{A1} \equiv\ & \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \\
& \geqslant 1 memoryUnitSizeInMb. =^{int}_{[128]} \sqcap \exists priceInPound. <^{int}_{[500]} \\
\text{PulgIn match: } C_{A2} \equiv\ & \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \\
& \geqslant 1 memoryUnitSizeInMb.(=^{int}_{[128]} \vee =^{int}_{[256]}) \\
& \sqcap \exists priceInPound. <^{int}_{[700]} \\
\text{Subsume match: } C_{A3} \equiv\ & \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \\
& \geqslant 1 memoryUnitSizeInMb. =^{int}_{[128]} \sqcap \exists priceInPound. <^{int}_{[500]} \\
& \sqcap \forall orderDate.((\geq^{int}_{[20040801]} \wedge \leq^{int}_{[20040831]}) \vee \\
& (\geq^{int}_{[20040901]} \wedge \leq^{int}_{[20040930]})) \sqcap \forall orderDate, deliverDate. <^{int} \\
\text{Intrsect. match: } C_{A4} \equiv\ & \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \\
& \geqslant 1 memoryUnitSizeInMb. =^{int}_{[128]} \sqcap \exists priceInPound. >^{int}_{[400]} \\
& \sqcap \leqslant 1 priceInPound.\top_{\mathrm{D}} \sqcap\ \exists CPUFreqInGHz. =^{real}_{[2.8]} \\
\text{Disjoint match: } C_{A5} \equiv\ & \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \\
& \geqslant 2 memoryUnitSizeInMb. =^{int}_{[256]} \sqcap \exists priceInPound. <^{int}_{[500]} \\
& \leqslant 2 memoryUnitSizeInMb. =^{int}_{[256]} \\
& \sqcap \forall HardDiskBrand, USBKeyBrand. =^{str}
\end{aligned}$$

Figure 8.1: Example matching advertisements

Figure 8.1 presents five example matching advertisements for C_{R1} in five different matching levels. Among them, C_{A1} is the exact match. In realistic situations, however, it is not to easy have an exact match, since advertisements might provide more general

or more specific information. For example, C_{A2} states that $priceInPound$ is only less than 700 and that the $memoryUnitSizeInMb$ can be either 128 or 256 (represented by the datatype expression $=^{int}_{[128]} \vee =^{int}_{[256]}$). C_{A3} adds two restrictions on $orderDate$s: firstly, the order date must be in August and September of 2004, which is represented by the datatype expression $(\geq^{int}_{[20040801]} \wedge \leq^{int}_{[20040831]}) \vee (\geq^{int}_{[20040901]} \wedge \leq^{int}_{[20040930]})$; secondly, $orderDate$s should be sooner than (represented by the binary predicate $<^{int}$) $DeliverDate$s, indicating that PCs will be delivered on some date *after* orders are made. As a result, C_{A2} and C_{A3} are PlugIn match and Subsume match of C_{R1}, respectively. C_{A4} says the $priceInPound$ is greater than 400, and the $CPUFreqInGHz$ of their PCs is 2.8;[4] it is an Intersection match. Finally, C_{A5} advertises that their PCs have exactly two memory chips, with the $memoryUnitSizeInMb$ of each chip is 256, and the $HardDiskBrand$ and $USBKeyBrand$ in their PCs are the same (represented by the binary predicate $=^{str}$); hence it is a disjoint (failed) match.

8.2.3 Matching Algorithm

We use the algorithm presented in [84] to detect the above five levels of matching. Firstly, we use our prototype to classify the hierarchy for all the advertised services and then compute the taxonomy position of the capability request C_R. Advertisements with capability equivalent to C_R are considered to be Exact matches, those subsumed by but not equivalent to C_R are considered to be PlugIn matches, and those subsuming but not equivalent to C_R are considered to be Subsume match. We can then compute the taxonomy position of the negation of the capability request $\neg C_R$. Advertisements with capability subsuming but not equivalent to $\neg C_R$ are considered to be Intersection matches, while those subsumed by $\neg C_R$ are considered to be Disjoint (failed) matches.

We have done some tests with our prototype for the matchmaking use case. In terms of functionality, our implemented matchmaking algorithm has achieved its purpose: it can respond to an input capability request with the results of matched advertisements of all the five degrees. Performance is relatively poor: e.g., classifying a TBox with 20 concepts (each of which has at least two datatype expression-related sub-concepts), a test that requires 235 subsumption tests, takes 34.8 seconds, i.e., 0.148 second per subsumption test.

This is, however, to be expected. Firstly, reasoning with datatypes is generally

[4]Note that $=^{real}_{[2.8]}$ is not a supported predicate for our prototype: our prototype will not reject it and even provides minimum checking for it; i.e., if $=^{real}_{[2.8]}$ and $\overline{=^{real}_{[2.8]}}$ are both in a predicate conjunction, this conjunction is *unsatisfiable*.

hard when there is more than one base datatype (our prototype supports both integer and string). This is because full negations of datatype expressions can introduce non-determinism. For example, when we check if C_{A1} is subsumed by C_{R1}, we actually check if $C_{A1} \sqcap \neg C_{R1}$ is unsatisfiable. The tableaux algorithm we use requires the negated normal form (NNF) of the input concept. Therefore, we transform $\neg C_{R1}$ into its NNF form $\sim C_{R1}$:

$$\begin{aligned} \sim C_{R1} \equiv\ & \neg \mathsf{PC} \sqcup \forall processor.\neg \mathsf{Pentium4} \sqcup \\ & \forall memoryUnitSizeInMb.(\neg =^{int}_{[128]}) \sqcup\ \forall priceInPound.(\neg <^{int}_{[500]}) \end{aligned}$$

which is further transformed (according to Definition 7.2 on page 179) into

$$\begin{aligned} \sim C_{R1} \equiv\ & \neg \mathsf{PC} \sqcup \forall processor.\neg \mathsf{Pentium4} \sqcup \\ & \forall memoryUnitSizeInMb.(\neg \neq^{int}_{[128]} \vee \neg integer) \sqcup \\ & \forall priceInPound.(\geq^{int}_{[500]} \vee \neg integer) \end{aligned}$$

with the result that disjunctions of datatype expressions are introduced.

Secondly, in our prototype, we did not cache the normalised forms of datatype expressions. Therefore, when the datatype manager receives a query from the DL reasoner, it always normalises all the datatype expressions no matter whether they have been normalised before.

8.2.4 Speeding Up the Answer

Some optimisation techniques can be used to improve the performance of our prototype. Firstly, we can apply known optimisation techniques for concrete domain reasoning [141, 144, 49], such as incremental concrete domain reasoning and dependency-directed backtracking, in our framework. Secondly, we could devise some new optimisation techniques that are related to the new features of our framework.

Incremental Datatype Checker The idea is that a datatype checker should be able to keep track of previous internal states and only check if the additional constraints would introduce contradictions. For example, if a datatype checker has once determined that

$$p_1(\vec{v_1}) \wedge p_2(\vec{v_2})$$

is *satisfiable*, when it receives a new query as follows

$$p_1(\vec{v_1}) \wedge p_2(\vec{v_2}) \wedge p_3(\vec{v_3}),$$

it should not check it all over again, but recall the internal state for $p_1(\vec{v_1}) \wedge p_2(\vec{v_2})$ and check if adding $p_3(\vec{v_3})$ will cause any contradictions.

A key question here is how to efficiently keep track of previous internal states, since datatype checkers may result in keeping many unused previous states. Proper strategies should be introduced so that datatype checkers know when to keep a copy of an internal state and how long they should keep the state.

Dependency-Directed Backtracking An adaptation of dependency-directed backtracking (DDB) to support datatype reasoning is described in [141]. The idea is that a DDB-enabled datatype checker should be able to identify all minimal, inconsistent sets of predicates (also referred to as *clash culprits*). These minimal sets define the necessary dependencies for backtracking. If this is not supported by the datatype checker, or is not computationally feasible, DDB must be disabled after a CD clash.

If a datatype checker is not able to identify all minimal inconsistent sets, [49] proposes that using a technique similar to semantic backtracking can help to keep the recorded predicates and their dependencies as small as possible. The main idea is that when we want to check whether adding $p(\vec{v})$ causes a contradiction, we try $\neg p(\vec{v})$ first. If $\neg p(\vec{v})$ causes a clash, the current internal state already entails $p(\vec{v})$; therefore, the backtracking dependencies of $p(\vec{v})$ can be safely ignored. If $\neg p(\vec{v})$ does not cause a clash, $p(\vec{v})$ is added to the current internal state. In case the datatype checker signals an inconsistency, the last predicate added to its current internal state is guaranteed to be a clash culprit. A possible problem here is that, when we have multiple base datatypes in a datatype group, full negation would introduce more disjunctions; hence, it is unclear whether this relaxed version will actually work in this situation.

Other Optimisations Further optimisation techniques might be useful for handling datatype expressions. For example, we could further improve the performance by caching the normalised datatype expressions in the datatype manager. A key question is how to cache reusable normalised datatype expressions and avoid caching unused ones.

We did not investigate the above ideas yet — they will be the subject of our future work.

Chapter Achievements

- We briefly described our implementation for the $\mathcal{SHIQ}(\mathcal{G})$ DL.
- We demonstrated the usefulness and flexibility of our framework.
- We discussed possible ways to improve the performance of our prototype.

Chapter 9

Discussion

This thesis concludes with a review of the work presented and an assessment of the extent to which the objectives set out in Chapter 1 have been met. The significance of the major results is summarised, and directions for future work are suggested.

9.1 Thesis Overview

This thesis proposes solutions for the two problems that hold back the wider adoption of DL-based SW ontology languages (cf. Chapter 3). To solve the first problem, it proposes RDFS(FA), a novel sub-language of RDF(S), as a firm semantic foundation for the latest DL-based SW ontology languages. To solve the second problem, it proposes two decidable extensions of OWL DL, viz. OWL-E and OWL-Eu, that support customised datatypes and datatype predicates.

Furthermore, it presents a DL reasoning framework to represent customised datatypes and datatype predicates with datatype expressions, and to provide a wide range of $\mathcal{G}$-combined DLs, including the very expressive OWL-E and OWL-Eu DLs, that are decidable and support datatype expressions. The framework can be used to provide decision procedures for $\mathcal{G}$-combined DLs, including those that are closely related to OWL DL, DAML+OIL, OWL-Eu and OWL-E. An important feature of the proposed framework is its flexibility: the hybrid reasoner in the framework is highly extensible to support new datatypes and datatype predicates, new forms of datatype expressions and new decidable Description Logics.

To sum up, the thesis has demonstrated that Description Logics can provide clear semantics, decision procedures and flexible reasoning services for SW ontology languages, including those that provide customised datatype and datatype predicates.

9.2 Significance of Major Contributions

The objectives set out in Section 1.4 have been successfully fulfilled by the following major contributions of the thesis:

- the design of *RDFS(FA)*, a sub-language of RDF(S) with DL-style model theoretic semantics, which provides a firm foundation for using DL reasoning in the Semantic Web and thus solidifies RDF(S)'s proposed role as the foundation of the Semantic Web;
- the *datatype group approach*, which specifies a formalism to unify datatypes and datatype predicates and to provide a wide range of (including very expressive) *decidable* Description Logics integrated with customised datatypes and datatype predicates;
- the design of *OWL-E*, a decidable extension of OWL DL that provides customised datatypes and datatype predicates based on datatype groups, and its unary restriction *OWL-Eu*, which is a much smaller extension of OWL DL;
- the design of practical *tableaux algorithms* for a wide range of DLs that are combined with arbitrary conforming datatype groups, including those of a family of DLs that are closely related to OWL DL, DAML+OIL, OWL-Eu and OWL-E;
- a flexible *framework architecture* to support decidable Description Logic reasoning services for Description Logics integrated with datatype groups.

RDFS(FA)

The first objective of the thesis was to propose a novel modification of RDF(S) as a firm semantic foundation for the latest DL-based SW ontology languages. Such a modification of RDF(S) should satisfy the requirements on language layering and applications presented in Section 4.1.1; i.e., its semantics should be compatible with that of OWL DL, and it should still provide the main features of RDFS.

RDFS(FA) satisfies all these requirements. Firstly, strata 0-2 in RDFS(FA) have a standard model theoretic semantics such that more expressive FOL ontology languages, such as the W3C standard OWL, can be layered on top of them and are compatible with RDFS(FA)'s metamodeling architecture. The bidirectional one-to-one mapping presented in Table 4.1 (page 96) between RDFS(FA) axioms in strata 0-2

and OWL DL axioms enables RDFS(FA)-agents and OWL DL-agents to communicate with each other (cf. Theorem 4.4 on page 96) in a much easier way.

Furthermore, RDFS(FA) clarifies the vision of the Semantic Web: RDF is *only* a standard syntax for SW annotations and languages; the meaning of annotation comes from either external agreements (such as Dublin Core) or ontologies, both of which are supported by RDFS(FA). As an ontology language, RDFS(FA) provides a UML-like layered style for using RDFS, which makes it more intuitive and easier to understand and use. Data-valued annotation properties in RDFS(FA) enable SW applications to make use of URIrefs of ontology elements, such as classes, in results of various ontology inferences.

Note that RDFS(FA) does not allow cross strata abstract properties (such as a property relating an individual in stratum 0 and a class in stratum 3); hence, users who prefer this kind of non-layered style can/should use the full RDFS. RDFS(FA), instead, provide annotation properties to allow that 'anyone can say anything about anything' (such that users can, e.g., annotate an individual in stratum 0 with a typed literal that represents a class URIref in stratum 3).

Summarily, RDFS(FA) solidifies RDF(S)'s intended role as the foundation of the Semantic Web by restoring the broken link between RDF(S) and OWL DL and by clarifying the vision of the Semantic Web.

OWL-E and OWL-Eu

The second objective of the thesis was to propose some extensions of OWL DL, so as to support customised datatypes and datatype predicates. Such extensions of OWL DL should satisfy the requirements presented in Section 6.1.1. In other words, these extensions should be decidable and should provide customised datatypes and datatype predicates; furthermore, it is desirable that they should also be extensions of DAML+OIL and should overcome the limitations of OWL DL presented in the 'Limitations of OWL Datatyping' section on page 77.

OWL-E satisfies all these requirements. Firstly, using $\mathcal{SHOIQ}(\mathcal{G})$ as its underpinning, OWL-E is a decidable extension of both OWL DL and DAML+OIL, which provides customised datatypes and predicates; in fact, all the basic reasoning services of OWL-E are decidable (cf. Theorem 6.3). Secondly, as a $\mathcal{G}$-combined DL, OWL-E overcomes the limitations of OWL DL with respect to negated datatypes and datatype domain. Finally, OWL-E allows users to define names (in fact, URIrefs) for $\mathcal{G}$-datatype expressions, including enumerated datatypes.

OWL-Eu, as a unary restriction of OWL-E, is a small extension of OWL DL.[1] The design of OWL-Eu is motivated by the W3C's 'one small step at a time' strategy — the *only* extension it presents is to extend OWL data ranges to support unary $\mathcal{G}$-datatype expressions, which can be used ro represent unary customised datatype predicates. OWL-Eu is much easier for existing OWL tools to adapt to, as unary $\mathcal{G}$-datatype expressions can be treated as unrecognised data ranges (unrecognised in general, but can make sense if they are recognised by certain supporting tools).

Note that OWL-Eu does not provide arbitrary datatype expressions (such as 'sum no greater than 15', 'multiply by 1.6' or even simple binary comparison predicates such as '$<$') and qualified number restrictions. Users who need these features thus can/should use OWL-E instead.

To sum up, as customised datatypes and datatype predicates are necessary in SW and ontology applications, OWL-E and OWL-Eu are two useful and decidable extensions to OWL DL.

A DL Reasoning Framework

The third objective of the thesis was to provide a DL reasoning framework, which (i) supports customised datatypes and datatype predicates, (ii) integrates a family of decidable DLs, including very expressive ones, with customised datatypes and datatype predicates, and (iii) provides decision procedures and flexible reasoning services for some members of this family that are closely related to OWL and the proposed extensions. The DL reasoning framework proposed in the thesis satisfies all these requirements.

Datatype Group Firstly, the datatype group approach provides a general formalism to unify existing ontology-related datatype and predicate formalisms (such as the concrete domain approach and OWL datatyping) and to overcome their limitations (cf. Section 5.1.3). Most importantly, $\mathcal{G}$-datatype expressions, in the unified formalism, can be used to represent customised datatypes, including XML Schema user-derived datatypes by restriction and union (cf. Section 3.3.1), and datatype predicates (cf. Example 5.7).

$\mathcal{G}$-combinable and $\mathcal{G}$-combined DLs Secondly, a scheme for integrating an arbitrary

[1] The underpinning of OWL-Eu is $\mathcal{SHOIN}(\mathcal{G}_1)$, which does not provide datatype qualified number restrictions; hence, OWL-Eu is not an extension of DAML+OIL.

conforming datatype group into $\mathcal{G}$-combinable Description Logics has been provided. An outstanding feature of the $\mathcal{G}$-combined DLs is that they provide customised datatypes and datatype predicates. Theorem 5.19 shows that all members of the family of $\mathcal{G}$-combined DLs, including $\mathcal{SHOIQ}(\mathcal{G})$, are decidable; interestingly, it also generalises (almost) all the known decidability results, on concept satisfiability w.r.t. TBoxes and RBoxes, of feature-chain-free DLs combined with admissible concrete domains.

$\mathcal{G}$-augmented Tableaux Algorithm Thirdly, given an arbitrary $\mathcal{G}$-combinable DL $\mathcal{L}$, a $\mathcal{G}$-augmented tableaux algorithm for $\mathcal{L}(\mathcal{G})$ can be constructed, based on a tableaux algorithm that is a decision procedure for $\mathcal{L}$-concept satisfiability w.r.t. to RBoxes (cf. Theorem 6.14). As there are no published tableaux algorithms for $\mathcal{SHOIQ}$, we stick to the 'divide and conquer' strategy and use the $\mathcal{G}$-augmented tableaux algorithms of the $\mathcal{SHOQ}(\mathcal{G})$, $\mathcal{SHIQ}(\mathcal{G})$ and $\mathcal{SHIO}(\mathcal{G})$ DLs to provide reasoning support for SW ontology languages OWL DL, DAML+OIL, OWL-Eu and OWL-E. Theorem 6.14 ensures that once we have a tableaux algorithm for $\mathcal{SHOIQ}$, we can easily upgrade it to one for $\mathcal{SHOIQ}(\mathcal{G})$.

Framework Architecture Finally, the proposed framework architecture ensures that the framework can provide flexible reasoning services for $\mathcal{G}$-combined DLs $\mathcal{L}(\mathcal{G})$ if the corresponding $\mathcal{G}$-augmented tableaux algorithms exist. From the viewpoint of users, the DIG/OWL-E interface (cf. Section 7.1.2) allows applications to use arbitrary $\mathcal{G}$-datatype expressions, so as to construct customised datatypes and predicates. From the viewpoint of reasoners, the hybrid reasoner is highly extensible to support new datatype predicates, new forms of datatype expressions and new $\mathcal{G}$-combined Description Logics.

9.3 Future Work

The proposal of RDFS(FA), OWL-E and OWL-Eu and the study of our DL reasoning framework have suggested several promising paths for further research.

Reasoning Support for RDFS(FA) Although Theorem 4.4 indicates that we can use DL reasoners to reason with RDFS(FA) axioms in strata 0-2,[2] this thesis has not presented a detailed solution for reasoning with RDFS(FA).

[2] Or even in every three adjacent strata.

The Design of OWL FA Although having suggested that it is possible to have a new sub-language of OWL — OWL FA, this thesis has not specified any details of this new sub-language. OWL FA could further enjoy useful features, such as meta-classes, meta-properties and data-value annotation properties, of RDFS(FA).

A Tableaux Algorithm for $\mathcal{SHOIQ}$ How to design a tableaux algorithm of the $\mathcal{SHOIQ}(\mathcal{G})$ DL has been an open question for quite a while. Once we have it, we can design a $\mathcal{G}$-augmented tableaux algorithm for $\mathcal{SHOIQ}(\mathcal{G})$, the $\mathcal{G}$-combined DL that underpins OWL-E.

Complexity Although it has proved the decidability of the $\mathcal{SHOIQ}(\mathcal{G})$ DL, this thesis has not provided any complexity result about it. Nor has the thesis provided any complexity results for the $\mathcal{G}$-augmented tableaux algorithms, such as those for $\mathcal{SHOQ}(\mathcal{G})$, $\mathcal{SHIQ}(\mathcal{G})$ and $\mathcal{SHIO}(\mathcal{G})$.

Optimisations As full negations of even datatype predicates can introduce nondeterminism, reasoning with $\mathcal{G}$-datatype expressions is generally hard. It could be helpful to investigate if the known optimisation techniques for concrete domains can be applied in $\mathcal{G}$-augmented tableaux algorithms and the datatype manager algorithm (cf. Definition 7.2).

More Datatype Predicates More research is needed to investigate some less well understood predicates (such as the string predicates concatenation), predicates that are defined over multiple disjoint base datatypes (such as the string predicate length), predicates that do not fit RDF and OWL datatyping, such as XML Schema user-derived datatypes by list and some datatype-related functions and operators provided in XPath [28] and XQuery [18, 93].

Other future work includes investigations of the application of datatype groups in various extensions (such as the Semantic Web Rule Language [72, 71]) or variants (such as OWL Flight [33]) of OWL DL, so as to facilitate the use of datatypes and datatype predicates in such languages.

Solving the problems that discourages potential users from adopting DL-based Semantic Web ontology languages is an encouraging result for both the DL and SW communities. It is hoped that RDFS(FA), OWL-E, OWL-Eu and the proposed reasoning framework will provide a firm foundation for ongoing research into realising the Semantic Web and, in general, into reasoning support for ontology applications.

Bibliography

[1] H. Alvestrand. Rfc 3066 - tags for the identification of languages. Technical report, IETF, Jan 2001. URL `http://www.isi.edu/in-notes/rfc3066.txt`.

[2] Carlos Eduardo Areces. *Logic Engineering - The Case of Description and Hybrid Logics*. PhD thesis, Universiteit van Amsterdam, 2000.

[3] F. Baader. Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 446–451, Sydney (Australia), 1991.

[4] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.-J. Profitlich. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or: Making KRIS get a move on. In *Proceedings of the Third International Conference on the Principles of Knowledge Representation and Reasoning (KR'92)*, pages 270–281, 1992. Also available as DFKI RR-93-03.

[5] F. Baader, C. Lutz, H. Sturm, and F. Wolter. Fusions of Description Logics and Abstract Description Systems. *Journal of Artificial Intelligence Research (JAIR)*, 16:1–58, 2002.

[6] F. Baader, D. L. McGuiness, D. Nardi, and P. Patel-Schneider, editors. *Description Logic Handbook: Theory, implementation and applications*. Cambridge University Press, 2002.

[7] F. Baader and W. Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95. Cambridge University Press, 2003.

[8] Franz Baader and Philipp Hanschke. A Schema for Integrating Concrete Domains into Concept Languages. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 452–457, 1991.

[9] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics for the Semantic Web. *KI – Künstliche Intelligenz*, 16(4):57–59, 2002. ISSN 09331875.

[10] Sean Bechhofer. The DIG Description Logic Interface: DIG/1.1. URL `http://dl-web.man.ac.uk/dig/2003/02/interface.pdf`, Feb 2003.

[11] Sean Bechhofer, Phillip Lord, and Raphael Volz. Cooking the Semantic Web with the OWL API. In *2nd International Semantic Web Conference (ISWC2003), Sanibel Island, Florida*, Oct 2003.

[12] Sean Bechhofer, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein eds. OWL Web Ontology Language Reference. URL `http://www.w3.org/TR/owl-ref/`, Feb 2004.

[13] R. Benjamins and D. Fensel. The Ontological Engineering Initiative $(KA)^2$. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98), IOS Press, 1998.*, 1998.

[14] S. Bergamaschi, S. Castano, S. De Capitani di Vimercati, S. Montanari, and M. Vincici. An Intelligent Approach to Information Integration. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98), IOS Press, 1998.*, 1998.

[15] Tim Berners-Lee. Realising the Full Potential of the Web. W3C Document, URL `http://www.w3.org/1998/02/Potential.html`, Dec 1997.

[16] Tim Berners-lee. Semantic Web Road Map. W3C Design Issues. URL `http://www.w3.org/DesignIssues/Semantic.html`, Oct. 1998.

[17] Paul V. Biron and Ashok Malhotra. Extensible Markup Language (XML) Schema Part 2: Datatypes – W3C Recommendation 02 May 2001. Technical report, World Wide Web Consortium, 2001. Available at `http://www.w3.org/TR/xmlschema-2/`.

[18] Scott Boag, Don Chamberlin, Mary F. Fernndez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Recommentdation, URL `http://www.w3.org/TR/xquery/`, July 2004.

[19] Ronald J. Brachman and Hector J. Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 34–37, Austin, Texas, August 1984. American Association for Artificial Intelligence.

[20] Sebastian Brandt. On Subsumption and Instance Problem in $\mathcal{ELH}$ w.r.t. General TBoxes. In *Proceedings of the 2004 International Workshop on Description Logics (DL2004)*, CEUR-WS, 2004.

[21] Dan Brickley. OWL CR feedback: owl:Class 'vs' rdfs:Class causing pain. Is owl:Class really needed? URL `http://lists.w3.org/Archives/Public/public-webont-comments/2003Sep/0009.html`, Sept 2003. Discussion in the `public-webont-comments@w3.org` mailing list.

[22] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. URL `http://www.w3.org/TR/rdf-schema/`, 2004. Series Editor: Brian McBride.

[23] J. Broekstra, M. Klein, S. Decker, D. Fensel, F. van Harmelen, and I. Horrocks. Enabling knowledge representation on the web by extending rdf schema. In *Proceedings of the tenth World Wide Web conference (WWW10)*, Hong Kong, May 2001.

[24] M. Buchheit, F. M. Donini, and A. Schaerf. Decidable Reasoning in Terminological Knowledge Representation Systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

[25] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the Decidability of Query Containment under Constraints. In *Proc. of the 17th ACM SIGACT SIGMOD SIGART Sym. on Principles of Database Systems (PODS'98)*, pages 149–158, 1998.

[26] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Riccardo Rosati, and Guido Vetere. DL-Lite: Practical Reasoning for Rich DLs. In *Proc. of*

the 2004 Description Logic Workshop (DL 2004). CEUR Electronic Workshop Proceedings, http://ceur-ws.org/Vol-104/, 2004.

[27] Cerebra. http://www.networkinference.com, 2002.

[28] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. W3C Recommentdation, URL http://www.w3.org/TR/xpath, Nov 1999.

[29] Roger L. Costello. Re: Even more Fuzzy about FunctionalProperty!, Mar. 2003. http://lists.w3.org/Archives/Public/www-rdf-logic/2003Mar/0047.html. Discussion in the www-rdf-logic@w3.org mailing list.

[30] Alain Coucho. Improving Web Searching Using Descriptive Graphs. In *Natural Language Processing and Information Systems: 9th International Conference on Applications of Natural Language to Information Systems, NLDB 2004, Salford, UK, June 23-25, 2004.*, 2004.

[31] S. Cranefield and M. Purvis. UML as an ontology modelling language. In *IJCAI-99 Workshop on Intelligent Information Integration*, 1999.

[32] DCMI. Dublin Core Metadata Element Set, Version 1.1: Reference Description. DCMI Recommendation, URL http://dublincore.org/documents/dces/, June 2003.

[33] Jos de Bruijn, Axel Polleres, Rubn Lara, and Dieter Fensel. OWL Flight, WSML Working Draft 23-08-2004. URL http://www.wsmo.org/2004/d20/d20.3/v0.1/20040823/, Aug 2004.

[34] Giuseppe De Giacomo and Maurizio Lenzerini. TBox and ABox Reasoning in Expressive Description Logics. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 316–327. Morgan Kaufmann, San Francisco, California, 1996.

[35] Ian Dickinson. Implementation Experience with the DIG 1.1 Specification. Technical report, Semantic and Adaptive Systems Dept, HP Labs Bristol, 2004.

[36] DIG. http://dl.kr.org/dig/, 2002.

[37] DIG. SourceForge DIG Interface Project. http://sourceforge.net/projects/dig/, 2004.

[38] Francesco M. Donini. Complexity of Reasoning. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 96–136. Cambridge University Press, 2003. ISBN 0-521-78176-0.

[39] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in Description Logics. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford, California, 1996. URL `citeseer.nj.nec.com/article/donini97reasoning.html`.

[40] FaCT. `http://www.cs.man.ac.uk/~horrocks/FaCT/`, 1998.

[41] FaCT++. `http://owl.man.ac.uk/factplusplus/`, 2003.

[42] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In R. Dieng, editor, *Proc. of the 12th European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW'00)*, number 1937 in Lecture Notes in Artificial Intelligence, pages 1–16. Springer-Verlag, 2000. URL `download/2000/oilnutshell.pdf`.

[43] A. Gangemi, D. Pisanelli, and G. Steve. Ontology Alignment: Experiences with Medical Terminologies. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98), IOS Press, 1998.*, 1998.

[44] Christine Golbreich, Olivier Dameron, Bernard Gibaud, and Anita Burgun. Web Ontology Language Requirements w.r.t Expressiveness of Taxonomy and Axioms in Medicine. In *Proceedings of the Second International Semantic Web Conference (ISWC 2003)*, 2003.

[45] Joint W3C/IETF URI Planning Interest Group. URIs, URLs, and URNs: Clarifications and Recommendations 1.0. URL `http://www.w3.org/TR/uri-clarification/`, 2001. W3C Note.

[46] T. R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[47] M. Gruninger and M. Fox. The Logic of Enterprise Modelling. In J. Brown and D. O'Sullivan, editors, *Reengineering the Enterprise*. Chapman and Hall, 1995.

[48] Nicola Guarino. Semantic matching: Formal ontological distinctions for information organization, extraction, and integration. In Maria Teresa Pazienza, editor, *Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology, International Summer School, SCIE-97, Frascati, Italy*, pages 139–170, 1997. URL `citeseer.ist.psu.edu/guarino97semantic.html`.

[49] Volker Haarslev and Ralf Mller. Practical Reasoning in RACER with a Concrete Domain for Linear Inequations. In *Proceedings of the International Workshop on Description Logics (DL-2002), Toulouse, France.*, pages 91–98, April. 2002.

[50] Volker Haarslev, Ralf Mller, and Michael Wessel. The Description Logic ALCNHR+ Extended with Concrete Domains: A Practically Motivated Approach. In *Proceedings of International Joint Conference on Automated Reasoning, IJCAR'2001, R. Gor, A. Leitsch, T. Nipkow (Eds.),Siena, Italy, Springer-Verlag, Berlin.*, pages 29–44, Jun. 2001.

[51] Volker Haarslev and Ralf Möller. RACER System Description. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083, 2001.

[52] Philipp Hanschke. Specifying Role Interaction in Concept Languages. In *Proceedings of the Third International Conference on the Principles of Knowledge Representation and Reasoning (KR'92)*, pages 318–329, 1992.

[53] Patrick Hayes. RDF Semantics. Technical report, W3C, Feb 2004. W3C recommendation, URL `http://www.w3.org/TR/rdf-mt/`.

[54] J. Heinsohn, D. Kudenko, B. Nebel, and H.-J. Profitlich. An Empirical Analysis of Terminological Representation Systems. *Artificial Intelligence*, 68:367–397, 1994.

[55] B. Hollunder and F. Baader. Qualifying Number Restrictions in Concept Languages. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, KR-91*, pages 335–346, Boston (USA), 1991.

[56] B. Hollunder, W. Nutt, and M. Schmidt-Schaug. Subsumption algorithms for concept languages. In *Proceedings of the Ninth European Conference on Artificial Intelligence (ECAI'90)*, pages 348–353, 1990.

[57] I. Horocks, D.Fensel, J.Broestra, S.Decker, M.Erdmann, C.Goble, F.van Harmelen, M.Klein, S.Staab, R.Studer, and E.Motta. The Ontology Inference Layer OIL. Technical report, OIL technical report, Aug. 2000.

[58] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, The University of Manchester, 1997. URL `http://www.cs.man.ac.uk/~horrocks/Publications/download/1997/phd-2sss.ps.gz` .

[59] I. Horrocks. Using an Expressive Description Logic: FaCT or Fiction? In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.

[60] I. Horrocks. FaCT and iFaCT. In *Proceedings of the 1999 Description Logic Workshop (DL'99)*, pages 133–135, 1999.

[61] I. Horrocks. Implementation and Optimisation Techniques. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 306–346. Cambridge University Press, 2003. ISBN 0-521-78176-0.

[62] I. Horrocks and P. F. Patel-Schneider. Comparing subsumption optimizations. In *Proceedings of the 1998 Description Logic Workshop (DL'98)*, 1998.

[63] I. Horrocks and U. Sattler. A Description Logic with Transitive and Converse Roles and Role Hierarchies. LTCS-Report 98-05, LuFg Theoretical Computer Science, RWTH Aachen, Germany, 1998.

[64] I. Horrocks, U. Sattler, and S. Tobies. A PSPACE-algorithm for deciding $\mathcal{ALCI}_{R^+}$-satisfiability. LTCS-Report 98-08, LuFg Theoretical Computer Science, RWTH Aachen, Germany, 1998.

[65] I. Horrocks, U. Sattler, and S. Tobies. A Description Logic with Transitive and Converse Roles, Role Hierarchies and Qualifying Number Restrictions. LTCS-Report LTCS-99-08, LuFG Theoretical Computer Science,

RWTH Aachen, 1999. Revised version. See http://www-lti.informatik.rwth-aachen.de/Forschung/Reports.html.

[66] I. Horrocks, U. Sattler, and S. Tobies. Practical Reasoning for Expressive Description Logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer-Verlag, 1999.

[67] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with Individuals for the Description Logic $\mathcal{SHIQ}$. In David MacAllester, editor, *CADE-2000*, number 1831 in LNAI, pages 482–496. Springer-Verlag, 2000.

[68] Ian Horrocks and Peter F. Patel-Schneider. The generation of DAML+OIL. In *Proceedings of the 2001 Description Logic Workshop (DL 2001)*, pages 30–35. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-49/`, 2001.

[69] Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, number 2870 in Lecture Notes in Computer Science, pages 17–29. Springer, 2003. ISBN 3-540-20362-1.

[70] Ian Horrocks and Peter F. Patel-Schneider. Three theses of representation in the semantic web. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 39–47. ACM, 2003. ISBN 1-58113-680-3.

[71] Ian Horrocks and Peter F. Patel-Schneider. A Proposal for an OWL Rules Language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 723–731. ACM, 2004.

[72] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language — Combining OWL and RuleML. W3C Member Submission, `http://www.w3.org/Submission/SWRL/`, May 2004.

[73] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. Reviewing the Design of DAML+OIL: An Ontology Language for the Semantic Web. In *Proc.*

of the 18th Nat. Conf. on Artificial Intelligence (AAAI 2002), pages 792–797. AAAI Press, 2002. ISBN 0-26251-129-0.

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

[75] Ian Horrocks and Ulrike Sattler. Ontology reasoning in the $\mathcal{SHOQ}$(D) description logic. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 199–204, 2001.

[76] Ian Horrocks and Ulrike Sattler. Optimised Reasoning for $\mathcal{SHIQ}$. In *Proc. of the 15th Eur. Conf. on Artificial Intelligence (ECAI 2002)*, pages 277–281, July 2002. ISBN 1-58603-257-7.

[77] U. Hustadt, B. Motik, and U. Sattler. Reducing $\mathcal{SHIQ}^-$ Description Logic to Disjunctive Datalog Programs. In D. Dubois, C. Welty, and M.-A. Williams, editors, *Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR2004)*, pages 152–162. AAAI Press, 2004.

[78] Jan Jladik and Jörg Model. Tableau Systems for $\mathcal{SHI}$ and $\mathcal{SHIQ}$. In *Proceedings of the 2004 International Workshop on Description Logics (DL2004), Whistler, British Columbia, Canada*, June 2004.

[79] T. S. Kaczmarek, R. Bates, and G. Robins. Recent developments in NIKL. In *The 5th National Conference of the AAAI*, 1986.

[80] Atanas Kiryakov, Borislav Popov, Damyan Ognyanoff, Dimitar Manov, Angel Kirilov, and Miroslav Goranov. Semantic Annotation, Indexing, and Retrieval. In *Proceedings of the Second International Semantic Web Conference (ISWC 2003)*, 2003.

[81] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. URL `http://www.w3.org/TR/rdf-concepts/`, 2004. Series Editor: Brian McBride.

[82] Ora Lassila and Ralph R.Swick. Resource Description Framework (RDF) Model and Syntax Specification – W3C Recommendation 22 February 1999. Technical report, World Wide Web Consortium, 1999.

[83] Fritz Lehmann. Machine-Negotiated, Ontology-Based EDI (Electronic Data Interchange). In *Proceedings of the Workshop at NIST on Electronic Commerce, Current Research Issues and Applications*, pages 27–45, 1994. ISBN 3-540-60738-2.

[84] Lei Li and Ian Horrocks. A Software Framework For Matchmaking Based on Semantic Web Technology. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 331–339. ACM, 2003. ISBN 1-58113-680-3.

[85] C. Lutz. NExpTime-complete Description Logics with Concrete Domains. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artifical Intelligence, pages 45–60, Siena, Italy, 2001. Springer Verlag.

[86] Carsten Lutz. Complexity of Terminological Reasoning Revisited. In *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning LPAR'99*, Lecture Notes in Artificial Intelligence, pages 181–200. Springer-Verlag, September 6 – 10, 1999.

[87] Carsten Lutz. NExpTime-complete Description Logics with Concrete Domains. Technical report, LuFG Theoretical Computer Science, RWTH Aachen, Germany, 2000.

[88] Carsten Lutz. *The Complexity of Reasoning with Concrete Domains*. PhD thesis, Teaching and Research Area for Theoretical Computer Science, RWTH Aachen, 2001.

[89] Carsten Lutz. Description logics with concrete domains—a survey. In *Advances in Modal Logics Volume 4*. World Scientific Publishing Co. Pte. Ltd., 2002.

[90] Carsten Lutz. PSPACE Reasoning with the Description Logic $\mathcal{ALCF}(\mathcal{D})$. *Logic Journal of the IGPL*, 10(5):535–568, 2002.

[91] Bob MacGregor. Re: Why isn't FunctionalProperty a subClassOf owl:ObjectProperty?, Mar. 2003. `http://lists.w3.org/Archives/Public/www-rdf-logic/2003Mar/0049.html`. Discussion in the `www-rdf-logic@w3.org` mailing list.

[92] Andreas Maier, Hans-Peter Schnurr, and York Sure. Ontology-based information integration in the automotive industryy. In *Proceedings of the Second International Semantic Web Conference (ISWC 2003)*, 2003.

[93] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft, URL `http://www.w3.org/TR/xpath-functions/`, July 2004.

[94] Frank Manola and Eric Miller. RDF Primer, W3C Recommendation. URL `http://www.w3.org/TR/rdf-primer/`, 2004. Series Editor: Brian McBride.

[95] D. McGuinness. Ontologies for Electronic Commerce. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, 1999.

[96] D.L. McGuinness. Ontological Issues for Knowledge-Enhanced Search. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98), IOS Press, 1998.*, 1998.

[97] E. Mena, V. Kashyap, A. Illarramendi, and A. Sheth. Domain Specific Ontologies for Semantic Information Brokering on the Global Information Infrastructure. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98), IOS Press, 1998.*, 1998.

[98] G. Miller. WordNet: A Lexical Database for English. *Comm. ACM*, 38(11), November 1995.

[99] Marvin Minsky. A Framework for Representing Knoledge. In J. Haugeland, editor, *Mind Design*. The MIT Press, 1981.

[100] Ralf Moller and Volker Haarslev. Description Logic Systems. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 282–305. Cambridge University Press, 2003. ISBN 0-521-78176-0.

[101] B. Nebel. Terminological Cycles: Semantics and Computational Properties. In J. F. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, pages 331–361. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1991.

[102] Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*. Springer-Verlag, 1990.

[103] Bernhard Nebel. Terminological Reasoning is Inherently Intractable. *Artificial Intelligence*, 43:235–249, 1990.

[104] W. Nejdl, M. Wolpers, and C. Capella. The RDF Schema Specification Revisited. In *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik, Modellierung 2000*, Apr. 2000.

[105] Natasha Noy. Representing Classes As Property Values on the Semantic Web. W3C Working Draft, URL `http://www.w3.org/TR/2004/WD-swbp-classes-as-values-20040721/`, Jul. 2004.

[106] Jeff Z. Pan. Web Ontology Reasoning in the SHOQ(Dn) Description Logic. In *Carlos Areces and Maartin de Rijke, editors,Proceedings of the Methods for Modalities 2 (M4M-2)*, Nov 2001. ILLC, University of Amsterdam.

[107] Jeff Z. Pan. Reasoning Support for OWL-E (Extended Abstract). In *Proc. of Doctoral Programme in the 2004 International Joint Conference of Automated Reasoning (IJCAR2004)*, July 2004.

[108] Jeff Z. Pan and Ian Horrocks. Metamodeling Architecture of Web Ontology Languages. In *Proceeding of the Semantic Web Working Symposium (SWWS)*, July 2001.

[109] Jeff Z. Pan and Ian Horrocks. Extending Datatype Support in Web Ontology Reasoning. In *Proc. of the 2002 Int. Conference on Ontologies, Databases and Applications of SEmantics (ODBASE 2002)*, Oct 2002.

[110] Jeff Z. Pan and Ian Horrocks. Metamodeling Architecture of Web Ontology Languages. In Isabel Cruz, Stefan Decker, Jérôme Euzenat, and Deborah McGuinness, editors, *The Emerging Semantic Web*, Frontiers in artificial intelligence and applications. IOS press, Amsterdam (NL), 2002. ISBN 1-58603-255-0.

[111] Jeff Z. Pan and Ian Horrocks. Reasoning in the SHOQ(Dn) Description Logic. In Ian Horrocks and Sergio Tessaris, editors, *Proc. of the 2002 Int. Workshop on Description Logics (DL-2002)*, Apr. 2002.

[112] Jeff Z. Pan and Ian Horrocks. RDFS(FA): A DL-ised Sub-language of RDFS. In *Proc. of the 2003 International Workshop on Description Logics (DL2003)*, 2003.

[113] Jeff Z. Pan and Ian Horrocks. RDFS(FA) and RDF MT: Two Semantics for RDFS. In *Proc. of the 2nd International Semantic Web Conference (ISWC2003)*, 2003.

[114] Jeff Z. Pan and Ian Horrocks. Web Ontology Reasoning with Datatype Groups. In *Proc. of the 2nd International Semantic Web Conference (ISWC2003)*, 2003.

[115] Peter F. Patel-Schneider. DLP. In *Description Logics*, 1999.

[116] Peter F. Patel-Schneider. Layering the Semantic Web: Problems and Directions. . In *2002 International Semantic Web Conference*, Jun 2002.

[117] Peter F. Patel-Schneider. Two Proposals for a Semantic Web Ontology Language. In *2002 International Description Logic Workshop*, Apr 2002.

[118] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. Technical report, W3C working draft, Mar. 2003. URL `http://www.w3.org/TR/2003/WD-owl-semantics-20030331/`.

[119] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. Technical report, W3C, Feb. 2004. W3C Recommendation, URL `http://www.w3.org/TR/2004/REC-owl-semantics-20040210/`.

[120] Pellet. `http://www.mindswap.org/2003/pellet/`, 2003.

[121] PICS. Platform for Internet Content Selectivity. `http://www.w3.org/PICS/`, 1997.

[122] D.M. Pisanelli, A. Gangemi, and G. Steve. WWW-available Conceptual Integration of Medical Terminologies: the ONIONS Experience. In *Proceedings of the 1997 AMIA Fall Symposium*, 1997.

[123] M. Ross Quillian. Word Concepts: A Theory and Simulation of Some Basic Capabilities. *Behavioral Science*, 12:410–430, 1967.

[124] Racer. `http://www.sts.tu-harburg.de/~r.f.moeller/racer/`, 1999.

[125] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification. W3C Recommendation, `http://www.w3.org/TR/html4/`, dEC. 1999.

[126] A. Rector. Re: [UNITS, OEP] FAQ : Constraints on data values range. URL `http://lists.w3.org/Archives/Public/public-swbp-wg/2004Apr/0216.html`, 2004. Discussion in the `public-swbp-wg@w3.org` mailing list.

[127] A. Rector, S. Bechhofer, C.A. Coble, I. Horrocks, W.A. Nowlan, and W.D. Solomon. The GRAIL Concept Modelling Language for Medical Terminology. *AI in Medicine*, 9:139–171, 1997.

[128] A. Schaerf. Reasoning With Individuals in Concept Languages. *Data and Knowledge Engineering*, 13(2):141–176, 1994.

[129] K. Schild. A correspondence theory for terminological logics: Preliminary report. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 466–471, 1991.

[130] K. Schild. Terminological cycles and the propositional μ-calculus. In *Proceedings of the Fourth International Conference on the Principles of Knowledge Representation and Reasoning (KR'94)*, pages 509–520, 1994.

[131] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

[132] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, Chichester,UK, 1986.

[133] Barry Smith, Jacob Khler, and Anand Kumar. On the Application of Formal Principles to Life Science Data: a Case Study in the Gene Ontology. In *Data Integration in the Life Sciences: First International Workshop, DILS 2004, Leipzig, Germany, March 25-26, 2004.*, 2004.

[134] K.A. Spackman. Manageing Clinical Terminology Hierarchies Using Algorithmic Calculation of Sybsupmtion: Experience with SNOMED-RT. *J. of the Amer. Med. Informatics Ass.*, 2000. Fall Symposium Special Issue.

[135] P.-H. Speel, F. van Raalte, P. E. van der Vet, and N. J. I. Mars. Runtime and Memory Usage Performance of Description Logics. In *Proc. of the 1st Int. KRUSE Symposium*, pages 13–27, 1995.

[136] R. Stevens, I. Horrocks, C. Goble, and S. Bechofer. Building a bioinformatics ontology using oil. *IEEE Inf. Techn. in Biomedicine*, 6(2):135–141, 2002.

[137] Nenad Stojanovic. On the Query Refinement in the Ontology-Based Searching for Information. In *WM 2003: Professionelles Wissesmanagement - Erfahrungen und Visionen, Beiträge der 2. Konferenz Professionelles Wissensmanagement, April 2003 in Luzern*, 2003.

[138] Sergio Tessaris. *Questions and answers: reasoning and querying in Description Logic*. PhD thesis, The University of Manchester, 1997. `URL http://www.cs.man.ac.uk/~tessaris/papers/phd-thesis.ps.gz .`

[139] Stephan Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen, 2001.

[140] Dmitry Tsarkov and Ian Horrocks. Efficient Reasoning with Range and Domain Constraints. In *Proc. of the 2004 Description Logic Workshop (DL 2004)*, pages 41–50, 2004.

[141] Anni-Yasmin Turhan and Volker Haarslev. Adapting Optimization Techniques to Description Logics with Concrete Domains. In *Proceedings of the International Workshop in Description Logics 2000 (DL2000), Aachen, Germany*, pages 247–256, 2000.

[142] M. Uschold and M. Gruninger. Ontologies: Principles, Methods and Applications. *The Knowledge Engineering Review*, 1996.

[143] M. Uschold, M. King, S. Moralee, and Y.Zorgios. The Enterprise Ontology. *The Knowledge Engineering Review*, 1998.

[144] Anni-Yasmin Turhan Volker Haarslev, Ralf Mller. Exploiting Pseudo Models for TBox and ABox Reasoning in Expressive Description Logics. In *Proceedings of International Joint Conference on Automated Reasoning, IJCAR'2001, R. Gor, A. Leitsch, T. Nipkow (Eds.),Siena, Italy, Springer-Verlag, Berlin.*, pages 61–75, Jun. 2001.

[145] Christopher A. Welty. The Ontological Nature of Subject Taxonomies. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98), IOS Press, 1998.*, 1998.

[146] Jan Wielemaker, Guus Schreiber, and Bob J. Wielinga. Prolog-based Infrastructure for RDF: Scalability and Performance. In *Proc. of the 2nd International Semantic Web Conference (ISWC2003)*, 2003.