RDF Validation Requirements - Evaluation and Logical Underpinning

Thomas Bosch[1], Andreas Nolle[2], Erman Acar[3], and Kai Eckert[3]

[1] GESIS – Leibniz Institute for the Social Sciences, Germany
thomas.bosch@gesis.org,
[2] Albstadt-Sigmaringen University, Germany
nolle@hs-albsig.de
[3] University of Mannheim, Germany
{erman, kai}@informatik.uni-mannheim.de

Abstract. There are many case studies for which the formulation of RDF constraints and the validation of RDF data conforming to these constraint is very important. As a part of the collaboration with the W3C and the DCMI working groups on RDF validation, we identified major RDF validation requirements and initiated an RDF validation requirements database which is available to contribute at http://purl.org/net/rdf-validation. The purpose of this database is to collaboratively collect case studies, use cases, requirements, and solutions regarding RDF validation. Although, there are multiple constraint languages which can be used to formulate RDF constraints (associated with these requirements), there is no standard way to formulate them. This paper serves to evaluate to which extend each requirement is satisfied by each of these constraint languages. We take reasoning into account as an important pre-validation step and therefore map constraints to DL in order to show that each constraint can be mapped to an ontology describing RDF constraints generically.

Keywords: RDF Validation, RDF Validation Requirements, RDF Constraints, Constraint Languages, Evaluation, Linked Data, Semantic Web

1 Introduction

The W3C organized the RDF Validation Workshop[4], where experts from industry, government, and academia discussed first use cases for RDF constraint formulation and validation. In 2014, two working groups on RDF validation have been established: the W3C RDF Data Shapes[5] and the DCMI RDF Application Profiles working groups[6]. Bosch and Eckert [1] collected the findings of these working groups and initiated a database of requirements to formulate and validate RDF constraints. The database is available for contribution at

[4] http://www.w3.org/2012/12/rdf-val/
[5] http://www.w3.org/2014/rds/charter
[6] http://wiki.dublincore.org/index.php/RDF-Application-Profiles

http://purl.org/net/rdf-validation. The intention associated with this database is to collaboratively collect case studies, use cases, requirements, and solutions regarding RDF validation in a comprehensive and structured way. The requirements are classified to better evaluate existing solutions. We mapped each requirement to formulate RDF constraints directly to an RDF constraint type.

A `constraint language` is a language which is used to formulate constraints. The W3C Data Shapes working group defines `constraint` as a component of a schema what needs to be satisfied[7]. There is no constraint language which can be seen as the standalone standard. However, there are multiple constraint languages (each having its own syntax and semantics) which can be used to express RDF constraints, such as existential and universal quantification, cardinality restrictions, and exclusive-or of properties. The five most popular constraint languages are Description Set Profiles (DSP)[8], Resource Shapes (ReSh)[9], Shape Expressions (ShEx)[10], the SPARQL Inferencing Notation (SPIN)[11], and the Web Ontology Language (OWL 2)[12].

In this paper, we describe each requirement within the RDF validation requirements database in detail (sections 2-75). Additional descriptions can be found directly in the database. Each requirement corresponds to an RDF constraint type which may be expressible by multiple constraint languages. For each requirement, we represent some examples in different constraint languages.

We evaluated to which extend the most promising five constraint languages fulfill each of the overall 74 requirements to formulate RDF constraints (section 82.1). We distinguished if a constraint is fulfilled by OWL 2 QL or if the more expressive OWL 2 DL is needed. We also take reasoning into account, as reasoning may be performed prior to validating constraints.

In order to define an ontology to describe RDF constraints generically, it is needed to define the terminology for the formulation of RDF constraints and to classify them. We identified four dimensions to classify constraints:

- `Universality:` specific constraints vs. generic constraints
- `Complexity:` simple constraints vs. complex constraints
- `Context:` property constraints vs. class constraints
- `DL Expressivity:` constraints expressible in DL vs. constraints not expressible in DL

As there are already five promising constraint languages, our purpose is not to invent a new constraint language. We rather developed a very simple ontology (only three classes, three object properties, and three data properties) which is universal enough to describe any RDF constraint expressible by any

[7] https://www.w3.org/2014/data-shapes/wiki/Glossary
[8] http://dublincore.org/documents/2008/03/31/dc-dsp/
[9] http://www.w3.org/Submission/shapes/
[10] http://www.w3.org/Submission/shex-primer/
[11] http://spinrdf.org/
[12] http://www.w3.org/TR/owl2-syntax/

RDF constraint language. We call this ontology the `RDF constraints ontology (RDF-CO)`[13].

`Specific constraints` are expressed by specific constraint languages like DSP, OWL 2, ReSh, ShEx, and SPIN. `Generic constraints` are expressed by the RDF-CO. As RDF-CO describes constraints generically, it does not distinguish constraints according to the dimension `universality`. The majority of constraints can be expressed in DL. In contrast, there are constraints which cannot be expressed in DL, but are also expressible in the RDF-CO. `Complex constraints` are built by combining `simple constraints` or complex constraints. DL statements which represents complex constraints are created out of DL statements representing composed constraints (if expressible in DL). Simple constraints may be applied to either properties (`properties constraints`) or classes (`class constraints`). There are no terms representing simple and complex constraints in the RDF-CO, since context classes (associated simple constraints hold for individuals of these classes) of simple constraints may just be reused by further constraints. As a consequence, the distinction of property and class constraints is sufficient to describe all possible RDF constraints.

In this paper, we investigate which constraints can be expressed in DL and which not. If a constraint can be expressed in DL, we added the mapping to DL and to the generic constraint in order to logically underpin associated requirements. If a constraint cannot be expressed in DL, we only added the mapping to the generic constraint. Therefore, we show that each constraint can be mapped to a generic constraint. In section 82.2 we classify the constraints according to the dimensions to classify constraints.

2 Subsumption

`Subsumption` (DL terminology: `concept inclusion`) corresponds to the requirement `R-100-SUBSUMPTION`. A subclass axiom SubClassOf(CE1 CE2) states that the class expression CE1 is a subclass of the class expression CE2. Roughly speaking, this states that CE1 is more specific than CE2.

2.1 Simple Example

All mothers are parents. The concept `Mother` is subsumed by the concept `Parent`:

$$\text{Mother} \sqsubseteq \text{Parent}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Mother	-	-	Parent	$\sqsubseteq$	

[13] Available at: https://github.com/boschthomas/RDF-CO

2.2 Simple Example

Jedis feel the force:

$$\text{Jedi} \sqsubseteq \text{FeelingForce}$$

Expressed by multiple constraint languages:

```
# OWL2:
Jedi rdfs:subClassOf FeelingForce .
```

```
# ReSh:
the extension ext:extendsShape may be used
```

```
# ShEx:
FeelingForce {
    feelingForce (true) }
Jedi {
    & FeelingForce ,
    attitute ('good') }
```

Data matching the shapes `FeelingForce` and `Jedi`:

```
Yoda
    feelingForce true ;
    attitute 'good' .
```

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Jedi	-	-	FeelingForce	$\sqsubseteq$	

2.3 Complex Example

If an individual is rich, then this individual is not poor:

$$\text{Rich} \sqsubseteq \neg\ \text{Poor}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	$\neg$ Poor	-	-	Poor	$\neg$	
class	Rich	-	-	$\neg$ Poor	$\sqsubseteq$	

3 Class Equivalence

`Class Equivalence` corresponds to the requirement `R-3-EQUIVALENT-CLASSES`. Concept equivalence asserts that two concepts have the same instances [4]. While synonyms are an obvious example of equivalent concepts, in practice one more often uses concept equivalence to give a name to complex expressions [4]. Concept equivalence is indeed subsumption from left and right ($A \sqsubseteq B$ and $B \sqsubseteq A$ implies $A \equiv B$).

3.1 Simple Example

$$\text{Person} \equiv \text{Human}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Person	-	-	Human	$\sqsubseteq$	
class	Human	-	-	Person	$\sqsubseteq$	

4 Sub Properties

`Sub Properties` (DL terminology: `role inclusion`) correspond to the requirements `R-54-SUB-OBJECT-PROPERTIES` and `R-54-SUB-DATA-PROPERTIES`. Subproperty axioms are analogous to subclass axioms. These axioms state that the property expression PE1 is a subproperty of the property expression PE2 — that is, if an individual x is connected by PE1 to an individual or a literal y, then x is also connected by PE2 to y.

4.1 Simple Example

$$\text{parentOf} \sqsubseteq \text{ancestorOf}$$

States that `parentOf` is a sub-role of `ancestorOf` , i.e., every pair of individuals related by `parentOf` is also related by `ancestorOf`.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	parentOf	ancestorOf	$\top$	$\sqsubseteq$	

4.2 Simple Example

$$\text{hasDog} \sqsubseteq \text{hasPet}$$

4.3 Complex Example

$$Person \sqcap \forall hasAge.\leqslant_{10} \sqsubseteq Person \sqcap \forall hasEvenAge.\leqslant_{10}$$

5 Object Property Paths

`Object Property Paths` (or Object Property Chains and in DL terminology complex role inclusion axiom or role composition) corresponds to the requirement `R-55-OBJECT-PROPERTY-PATHS`. The more complex form of sub properties. This axiom states that, if an individual x is connected by a sequence of object property expressions OPE1, ..., OPEn with an individual y, then x is also connected with y by the object property expression OPE. Role composition can only appear on the left-hand side of complex role inclusions [4].

5.1 Simple Example

$$\text{brotherOf} \circ \text{parentOf} \sqsubseteq \text{uncleOf}$$

```
# OWL2:
uncleOf owl:propertyChainAxiom ( brotherOf parentOf ) .
```

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	brotherOf, parentOf	uncleOf	$\top$	$\sqsubseteq$	-

6 Disjoint Properties

`Disjoint Properties` corresponds to the requirement `R-9-DISJOINT-PROPERTIES`. A disjoint properties axiom states that all of the properties are pairwise disjoint; that is, no individual x can be connected to an individual y by these properties.

6.1 Simple Example

The object properties `parentOf` and `childOf` are disjoint:

$$\text{Disjoint(parentOf, childOf)}$$

or alternatively:

$$parentOf \sqsubseteq \neg childOf$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	parentOf	$\neg$childOf	$\top$	$\sqsubseteq$	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	parentOf	childOf	$\top$	$\neq$	

7 Intersection

`Intersection` (composition, conjunction) corresponds to the requirements `R-15-CONJUNCTION-OF-CLASS-EXPRESSIONS` and `R-16-CONJUNCTION-OF-DATA-RANGES`. DLs allow new concepts and roles to be built using a variety of different constructors. We distinguish concept and role constructors depending on whether concept or role expressions are constructed. In the case of concepts, one can further separate basic Boolean constructors, role restrictions and nominals/enumerations [4]. Boolean concept constructors provide basic boolean operations that are closely related to the familiar operations of intersection, union and complement of sets, or to conjunction, disjunction and negation of logical expressions [4].

$$\text{Mother} \equiv \text{Female} \sqcap \text{Parent}$$

Concept inclusions allow us to state that all mothers are female and that all mothers are parents, but what we really mean is that mothers are exactly the female parents. DLs support such statements by allowing us to form complex concepts such as the intersection (also called conjunction) which denotes the set of individuals that are both female and parents. A complex concept can be used in axioms in exactly the same way as an atomic concept, e.g., in the equivalence $\text{Mother} \equiv \text{Female} \sqcap \text{Parent}$.

7.1 Simple Example

$$\text{Female} \sqcap \text{Parent}$$

Complex concept of all individuals which are of the concept `Female` and of the concept `Parent`.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Female ⊓ Parent	-	-	Female, Parent	⊓	

7.2 Simple Example

$$\text{Mother} \equiv \text{Female} \sqcap \text{Parent}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Mother	-	-	Female, Parent	⊓	

8 Disjunction

`Disjunction` of classes or data ranges corresponds to the requirements `R-17-DISJUNCTION-OF-CLASS-EXPRESSIONS` and `R-18-DISJUNCTION-OF-DATA-RANGES`. Synonyms are union and inclusive or.

8.1 Simple Example

$$\text{Father} \sqcup \text{Mother} \sqcup \text{Child}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Father ⊔ Mother ⊔ Child	-	-	Father, Mother, Child	⊔	

8.2 Simple Example

$$\text{Parent} \equiv \text{Father} \sqcup \text{Mother}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Parent	-	-	Father, Mother	$\sqcup$	

9 Negation

`Negation` (complement) corresponds to the requirements `R-19-NEGATION-OF-CLASS-EXPRESSIONS` and `R-20-NEGATION-OF-DATA-RANGES`.

9.1 Simple Example

$$\neg Married$$

Set of all individuals that are not married.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	$\neg$ Married	-	-	Married	$\neg$	

9.2 Complex Example

$$\text{Female} \sqcap \neg \text{Married}$$

All female individuals that are not married.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	$\neg$ Married	-	-	Married	$\neg$	
class	Female $\sqcap$ $\neg$ Married	-	-	Female, $\neg$ Married	$\sqcap$	

10 Disjoint Classes

`Disjoint Classes` corresponds to the requirement `R-7-DISJOINT-CLASSES`. A disjoint classes axiom states that all of the classes are pairwise disjoint; that is, no individual can be at the same time an instance of these classes.

10.1 Simple Example

Individuals cannot be male and female at the same time:

$$\text{Male} \sqcap \text{Female} \sqsubseteq \bot$$

or alternatively:

$$Male \sqsubseteq \neg Female$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Male ⊓ Female	-	-	Male, Female	⊓	
class	⊤	-	-	Male ⊓ Female, ⊥	⊑	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Male	-	-	Female	≠	
class	Female	-	-	Male	≠	

10.2 Simple Example

One can either be a hologram or a human, but not both:

$$\text{Hologram} \sqcap \text{Human} \sqsubseteq \bot$$

or alternatively:

$$Hologram \sqsubseteq \neg Human$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Hologram ⊓ Human	-	-	Hologram, Human	⊓	
class	⊤	-	-	Hologram ⊓ Human, ⊥	⊑	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Hologram	-	-	Human	≠	
class	Human	-	-	Hologram	≠	

11 Existential Quantifications

`Existential Quantification on Properties` conforms to the requirement `R-86-EXISTENTIAL-QUANTIFICATION-ON-PROPERTIES`. In DL terminology the `existential quantification` is also called `existential restriction`. An existential class expression consists of a property expression and a class expression or a data range, and it contains all those individuals that are connected by the property expression to an individual that is an instance of the class expression or to literals that are in the data range.

11.1 Simple Example

$$\exists \text{ parentOf} . \top$$

Complex concept that describes the set of individuals that are parents of at least one individual (instance of $\top$).

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\exists$ parentOf . $\top$	parentOf	-	$\top$	$\exists$	

11.2 Simple Example

$$\exists \text{ parentOf} . \text{Female}$$

The complex concept describes those individuals that are parents of at least one female individual, i.e., those that have a daughter.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\exists$ parentOf . Female	parentOf	-	Female	$\exists$	

11.3 Simple Example

$$\text{Parent} \equiv \exists \text{ parentOf} . \top$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Parent	parentOf	-	$\top$	$\exists$	

11.4 Simple Example

$$\text{ParentOfSon} \equiv \exists \text{ parentOf} . \text{Male}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	ParentOfSon	parentOf	-	Male	$\exists$	

12 Universal Quantifications

`Universal Quantification on Properties` corresponds to the requirement `R-91-UNIVERSAL-QUANTIFICATION-ON-PROPERTIES`, which is also called `value restriction` in DL terminology.

$(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$ where $\cdot^{\mathcal{I}}$ is an interpretation function, $\Delta^{\mathcal{I}}$ is the domain, $a, b \in \Delta^{\mathcal{I}}$ are individuals C is a concept, R is a role.

12.1 Simple Example

$$\forall \text{ parentOf.Female}$$

The set of individuals all of whose children are female also includes those that have no children at all.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	∀ parentOf . Female	parentOf	-	Female	∀	

13 Property Domains

`Property Domain` corresponds to the requirements `R-25-OBJECT-PROPERTY-DOMAIN` and `R-26-DATA-PROPERTY-DOMAIN`. The constraint restricts the domain of object and data properties. In DL terminology this constraint is also called domain restrictions on roles. The purpose is to declare that a given property is associated with a class, e.g. to populate input forms with appropriate widgets but also constraint checking. In OO terms this is the declaration of a member, field, attribute or association. $\exists R.\top \sqsubseteq C$ is the object property restriction where R is the object property (role) whose domain is restricted to concept C.

13.1 Simple Example

$$\exists \text{ sonOf} \ . \ \top \sqsubseteq \text{Male}$$

Restricts the domain of `sonOf` to male individual.
syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	⊤	sonOf	-	Male	domain	

14 Property Ranges

`Property Range` corresponds to the requirements `R-28-OBJECT-PROPERTY-RANGE` and `R-35-DATA-PROPERTY-RANGE`. This constraint restricts the range of object and data properties In DL terminology it is also called range restrictions on roles. $\top \sqsubseteq \forall R.C$ is the range restriction to the object property R (restricted by the concept C).

14.1 Simple Example

$$\top \sqsubseteq \forall \text{ sonOf . Parent}$$

equivalent to:

$$\exists sonOf^{-}.\top \sqsubseteq Person$$

Restricts the range of `sonOf` to parents.

```
# OWL 2:
sonOf rdfs:range Parent .
```

```
# DSP:
:hasDogRange
        a dsp:DescriptionTemplate ;
        dsp:resourceClass owl:Thing ;
        dsp:statementTemplate [
            a dsp:NonLiteralStatementTemplate ;
            dsp:property sonOf ;
            dsp:nonLiteralConstraint [
                a dsp:NonLiteralConstraint ;
                dsp:valueClass Parent ] ] .
```

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	sonOf	-	Parent	range	

15 Class-Specific Property Range

`Class-Specific Property Range` corresponds to the requirements `R-29-CLASS-SPECIFIC-RANGE-OF-RDF-OBJECTS` and `R-36-CLASS-SPECIFIC-RANGE-OF-RDF-LITERALS`. The constraint restricts the range of object and data properties for individuals within a specific context (e.g. class, application profile). The values of each member property of a class may be limited by their value type, such as xsd:string or Person.

15.1 Simple Example

Only men can have `sonOf` relationships to parents:

$$\neg \text{Man} \sqsubseteq \neg \exists \text{ sonOf.Parent}$$

15.2 Simple Example

Only vulcans can have friendship relationships to cardassians

$$\neg\text{Vulcan} \sqsubseteq \neg\exists \text{ friendOf.Carsassian}$$

16 Minimum Unqualified Cardinality Restrictions

`Minimum Unqualified Cardinality Restrictions on Properties` corresponds to the requirements `R-81-MINIMUM-UNQUALIFIED-CARDINALITY-ON-PROPERTIES` and `R-211-CARDINALITY-CONSTRAINTS`. $\leqslant nR.\top$ is the minimum unqualified cardinality restriction where $n \in \mathbb{N}$ (written $\leqslant nR$ in short). A minimum cardinality restrictions contains all those individuals that are connected by a property to at least n different individuals/literals that are instances of a particular class or data range. If the class is missing, it is taken to be owl:Thing. If the data range is missing, it is taken to be rdfs:Literal. For unqualified cardinality restrictions, classes respective data ranges are not stated.

16.1 Simple Example

```
# ShEx:
:Jedi {
    :attitude ('good') }
:JediStudent {
    & :Jedi ,
    :studentOf {}{1} }
:JediMaster {
    & :Jedi ,
    :mentorOf {}{1,2} }
:SuperJediMaster {
    & :Jedi ,
    :mentorOf {}{3,} }
```

- Jedis have the attitude 'good'
- Jedi students are students of exactly 1 resource
- Jedi masters are mentoring at least 1 and at most 2 resources
- Super Jedi masters are mentoring at least 3 resources

```
# data:
:Yoda
    :attitude 'good' ;
    :mentorOf :MaceWindu , :Obi-Wan , :Luke .
:MaceWindu
    :attitude 'good' ;
    :studentOf :Yoda .
:Obi-Wan
    :attitude 'good' ;
    :studentOf :Yoda ;
    :mentorOf :Anakin .
:Anakin
    :attitude 'good' ;
    :studentOf :Obi-Wan .
:Luke
    :attitude 'good' ;
    :studentOf :Yoda .
```

```
# Individuals matching the ':SuperJediMaster' data shape:
:Yoda
```

$$Jedi \sqsubseteq \exists attitude.\{good\}$$
$$JediStudent \sqsubseteq Jedi \sqcap \geqslant 1 studentOf.\top \sqcap \leqslant 1 studentOf.\top$$
$$JediMasters \sqsubseteq Jedi \sqcap \geqslant 1 mentorOf.\top \sqcap \leqslant 2 mentorOf.\top$$
$$SuperJediMaster \sqsubseteq Jedi \sqcap \geqslant 3 mentorOf.\top$$

16.2 Simple Example

$$Captain \sqsubseteq \geqslant 1 commandsVessel.\top$$

Captains command at least one vessel.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Captain	commandsVessel	-	$\top$	$\geqslant$	1

17 Minimum Qualified Cardinality Restrictions

`Minimum Qualified Cardinality Restrictions on Properties` corresponds to the requirements `R-75-MINIMUM-QUALIFIED-CARDINALITY-ON-PROPERTIES` and `R-211-CARDINALITY-CONSTRAINTS` A minimum cardinality restrictions contains all those individuals that are connected by a property to at least n different individuals/literals that are instances of a particular class or data range. If the class is missing, it is taken to be owl:Thing. If the data range is missing, it is taken to be rdfs:Literal. For qualified cardinality restrictions, classes respective data ranges are stated. $\geqslant nR.C$ is a minimum qualified cardinality restriction where $n \in \mathbb{N}$.

17.1 Simple Example

$$\geqslant 2 \text{ childOf . Parent}$$

Set of individuals that are children of at least two parents.

```
# OWL 2:
owl:Thing
    a owl:Restriction ;
    owl:minQualifiedCardinality "2"^^xsd:nonNegativeInteger ;
    owl:onProperty childOf ;
    owl:onClass Parent .
```

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\geqslant$ 2 childOf . Parent	childOf	-	Parent	$\geqslant$	2

17.2 Simple Example

foaf:Person $\equiv \geqslant 2$ hasName.xsd:string

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	foaf:Person	hasName	-	xsd:string	$\geqslant$	2

17.3 Simple Example

```
# ShEx:
:Jedi {
    :attitude ('good') }
:JediStudent {
    & :Jedi ,
    :studentOf @:Jedi{1} }
:JediMaster {
    & :Jedi ,
    :mentorOf @:Jedi{1,2} }
:SuperJediMaster {
    & :Jedi ,
    :mentorOf @:Jedi{3,} }
```

- Jedis have the attitude 'good'
- Jedi students are students of exactly 1 Jedi
- Jedi masters are mentoring at least 1 and at most 2 Jedis
- Super Jedi masters are mentoring at least 3 Jedis

```
# data:
:Yoda
    :attitude 'good' ;
    :mentorOf :MaceWindu , :Obi-Wan , :Luke .
:MaceWindu
    :attitude 'good' ;
    :studentOf :Yoda .
:Obi-Wan
    :attitude 'good' ;
    :studentOf :Yoda ;
    :mentorOf :Anakin .
:Anakin
    :attitude 'good' ;
    :studentOf :Obi-Wan .
:Luke
    :attitude 'good' ;
    :studentOf :Yoda .
```

```
# Individuals matching the ':SuperJediMaster' data shape:
:Yoda

# Individuals matching the ':JediMaster' data shape:
:Obi-Wan
```

$mentorOf$ and $studentOf$ are taken to be inverse properties:

$$Jedi \sqsubseteq \exists attitude.\{good\}$$
$$JediStudent \sqsubseteq Jedi \sqcap \geqslant 1 studentOf.Jedi \sqcap \leqslant 1 studentOf.Jedi$$
$$JediMasters \sqsubseteq Jedi \sqcap \geqslant 1 mentorOf.Jedi \sqcap \leqslant 2 mentorOf.Jedi$$
$$SuperJediMaster \sqsubseteq Jedi \sqcap \geqslant 3 mentorOf.Jedi$$

17.4 Complex Example

$$\text{Father2Daughters} \equiv \text{Man} \sqcap (\geqslant 2\ \text{hasChild.Woman})$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\geqslant$ 2 hasChild.Woman	hasChild	-	Woman	$\geqslant$	2
class	Father2Daughters	-	-	Man, $\geqslant$ 2 hasChild.Woman	$\sqcap$	-

17.5 Simple Example

$$Captain \sqsubseteq \geqslant 1 commandsVessel.Vessel$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Captain	commandsVessel	-	Vessel	$\geqslant$	1

17.6 Complex Example

$$FederationCaptain \sqsubseteq Federation \sqcap \geqslant 1 commandsVessel.Vessel$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\geqslant$1 commandsVessel . Vessel	commandsVessel	-	Vessel	$\geqslant$	1
class	FederationCaptain	-	-	Federation, $\geqslant$1 commandsVessel . Vessel	$\sqcap$	-

18 Maximum Unqualified Cardinality Restrictions

`Maximum Unqualified Cardinality Restrictions on Properties` corresponds to the requirements `R-82-MAXIMUM-UNQUALIFIED-CARDINALITY-ON-PROPERTIES` and `R-211-CARDINALITY-CONSTRAINTS`. A maximum cardinality restriction contains all those individuals that are connected by a property to at most n different individuals/literals that are instances of a particular class or data range. If the class is missing, it is taken to be owl:Thing. If the data range is not present, it is taken to be rdfs:Literal. Unqualified means that the class respective the data range is not stated. $\geqslant nR.\top$ is a maximum unqualified cardinality restriction where $n \in \mathbb{N}$ (written $\geqslant nR$ in short).

18.1 Simple Example

```
# ShEx:
:Jedi {
    :attitude ('good') }
:JediStudent {
    & :Jedi ,
    :studentOf {}{1} }
:JediMaster {
    & :Jedi ,
    :mentorOf {}{1,2} }
:SuperJediMaster {
    & :Jedi ,
    :mentorOf {}{3,} }
```

- Jedis have the attitude 'good'
- Jedi students are students of exactly 1 resource
- Jedi masters are mentoring at least 1 and at most 2 resources
- Super Jedi masters are mentoring at least 3 resources

```
# data:
:Yoda
    :attitude 'good' ;
    :mentorOf :MaceWindu , :Obi-Wan , :Luke .
:MaceWindu
    :attitude 'good' ;
    :studentOf :Yoda .
:Obi-Wan
    :attitude 'good' ;
    :studentOf :Yoda ;
    :mentorOf :Anakin .
:Anakin
    :attitude 'good' ;
    :studentOf :Obi-Wan .
:Luke
    :attitude 'good' ;
    :studentOf :Yoda .
```

```
# Individuals matching the ':JediMaster' data shape:
:Obi-Wan
```

$$Jedi \sqsubseteq \exists attitude.\{good\}$$
$$JediStudent \sqsubseteq Jedi \sqcap \geqslant 1 studentOf.\top \sqcap \leqslant 1 studentOf.\top$$
$$JediMasters \sqsubseteq Jedi \sqcap \geqslant 1 mentorOf.\top \sqcap \leqslant 2 mentorOf.\top$$
$$SuperJediMaster \sqsubseteq Jedi \sqcap \geqslant 3 mentorOf.\top$$

19 Maximum Qualified Cardinality Restrictions

`Maximum Qualified Cardinality Restrictions on Properties` corresponds to the requirements `R-76-MAXIMUM-QUALIFIED-CARDINALITY-ON-PROPERTIES` and `R-211-CARDINALITY-CONSTRAINTS`. A maximum cardinality restriction contains all those individuals that are connected by a property to at most n different individuals/literals that are instances of a particular class or data range. If the class is missing, it is taken to be owl:Thing. If the data range is not present, it is taken to be rdfs:Literal. Qualified means that the class respective the data range is stated. $\leqslant nR.C$ is a maximum qualified cardinality restriction where $n \in \mathbb{N}$.

19.1 Simple Example

$$\leqslant 2 \text{ childOf . Parent}$$

Set of individuals that are children of at most two parents.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\leqslant 2$ childOf . Parent	childOf	-	Parent	$\leqslant$	2

19.2 Simple Example

```
# ShEx:
:Jedi {
    :attitude ('good') }
:JediStudent {
    & :Jedi ,
    :studentOf @:Jedi{1} }
:JediMaster {
    & :Jedi ,
    :mentorOf @:Jedi{1,2} }
:SuperJediMaster {
    & :Jedi ,
    :mentorOf @:Jedi{3,} }
```

- Jedis have the attitude 'good'
- Jedi students are students of exactly 1 Jedi
- Jedi masters are mentoring at least 1 and at most 2 Jedis
- Super Jedi masters are mentoring at least 3 Jedis

```
# data:
:Yoda
    :attitude 'good' ;
    :mentorOf :MaceWindu , :Obi-Wan , :Luke .
:MaceWindu
    :attitude 'good' ;
    :studentOf :Yoda .
:Obi-Wan
    :attitude 'good' ;
    :studentOf :Yoda ;
    :mentorOf :Anakin .
:Anakin
    :attitude 'good' ;
    :studentOf :Obi-Wan .
:Luke
    :attitude 'good' ;
    :studentOf :Yoda .
```

```
# Individuals matching the ':JediMaster' data shape:
:Obi-Wan
```

$$Jedi \sqsubseteq \exists attitude.\{good\}$$
$$JediStudent \sqsubseteq Jedi \sqcap \geqslant 1 studentOf.Jedi \sqcap \leqslant 1 studentOf.Jedi$$
$$JediMasters \sqsubseteq Jedi \sqcap \geqslant 1 mentorOf.Jedi \sqcap \leqslant 2 mentorOf.Jedi$$
$$SuperJediMaster \sqsubseteq Jedi \sqcap \geqslant 3 mentorOf.Jedi$$

20 Exact Unqualified Cardinality Restrictions

`Exact Unqualified Cardinality Restrictions on Properties` corresponds to the requirements `R-80-EXACT-UNQUALIFIED-CARDINALITY-ON-PROPERTIES` and `R-211-CARDINALITY-CONSTRAINTS`. An exact cardinality restriction contains all those individuals that are connected by a property to exactly n different individuals that are instances of a particular class or data range. If the class is missing, it is taken to be owl:Thing. If the data range is not present, it is taken to be rdfs:Literal. Unqualified means that the class respective data range is not stated. $\geqslant nR.\top \sqcap \leqslant nR.\top$ is an exact unqualified cardinality restriction where $n \in \mathbb{N}$.

20.1 Simple Example

```
# ShEx:
:JediStudent {
    :studentOf {}{1} }
```

```
# ReSh:
:JediStudent a rs:ResourceShape ;
    rs:property [
        rs:name "studentOf" ;
        rs:propertyDefinition :studentOf ;
        rs:valueShape [ a rs:ResourceShape] ;
        rs:occurs rs:Exactly-one ; ] .
```

- Jedis have the attitude 'good'
- Jedi students are students of exactly 1 resource

```
# data:
:Yoda
    :attitude 'good' ;
    :mentorOf :MaceWindu , :Obi-Wan , :Luke .
:MaceWindu
    :attitude 'good' ;
    :studentOf :Yoda .
:Obi-Wan
    :attitude 'good' ;
    :studentOf :Yoda ;
    :mentorOf :Anakin .
:Anakin
    :attitude 'good' ;
    :studentOf :Obi-Wan .
:Luke
    :attitude 'good' ;
    :studentOf :Yoda .
```

```
# Individuals matching the ':JediStudent' data shape:
:MaceWindu :Obi-Wan :Anakin :Luke
```

$$JediStudent \sqsubseteq Jedi \sqcap \geqslant 1 studentOf.\top \sqcap \leqslant 1 studentOf.\top$$

21 Exact Qualified Cardinality Restrictions

`Exact Qualified Cardinality Restrictions on Properties` corresponds to the requirements `R-74-EXACT-QUALIFIED-CARDINALITY-ON-PROPERTIES` and `R-211-CARDINALITY-CONSTRAINTS`. An exact cardinality restriction contains all those individuals that are connected by a property to exactly n different individuals that are instances of a particular class or data range. If the class is missing, it is taken to be owl:Thing. If the data range is not present, it is taken to be rdfs:Literal. Qualified means that the class respective data range is stated. $\geqslant nR.C \sqcap \leqslant nR.C$ is an exact qualified cardinality restriction where $n \in \mathbb{N}$.

21.1 Simple Example

```
# ShEx:
:Jedi {
    :attitude ('good') }
:JediStudent {
    :studentOf @:Jedi{1} }
```

```
# ReSh:
:Jedi a rs:ResourceShape ;
    rs:property [
        rs:name "attitude" ;
        rs:propertyDefinition :attitude ;
        rs:allowedValue "good" ;
        rs:occurs rs:Exactly-one ;
    ] .
:JediStudent a rs:ResourceShape ;
    rs:property [
        rs:name "studentOf" ;
        rs:propertyDefinition :studentOf ;
        rs:valueShape :Jedi ;
        rs:occurs rs:Exactly-one ;
    ] .
```

- Jedis have the attitude 'good'
- Jedi students are students of exactly 1 Jedi

```
# data:
:Yoda
    :attitude 'good' ;
    :mentorOf :MaceWindu , :Obi-Wan , :Luke .
:MaceWindu
    :attitude 'good' ;
    :studentOf :Yoda .
:Obi-Wan
    :attitude 'good' ;
    :studentOf :Yoda ;
    :mentorOf :Anakin .
:Anakin
    :attitude 'good' ;
    :studentOf :Obi-Wan .
:Luke
    :attitude 'good' ;
    :studentOf :Yoda .
```

```
# Individuals matching the ':JediStudent' data shape:
:MaceWindu :Obi-Wan :Anakin :Luke
```

$$Jedi \sqsubseteq \exists attitude.\{good\}$$
$$JediStudent \sqsubseteq Jedi \sqcap \geqslant 1 studentOf.Jedi \sqcap \leqslant 1 studentOf.Jedi$$

21.2 Complex Example

$$\text{Person} \sqsubseteq \geqslant 2 \text{ childOf . Parent} \sqcap \leqslant 2 \text{ childOf . Parent}$$

Every person is a child of exactly two parents.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\geqslant$ 2 childOf . Parent	childOf	-	Parent	$\geqslant$	2
property	$\leqslant$ 2 childOf . Parent	childOf	-	Parent	$\leqslant$	2
class	Person	-	-	$\geqslant$ 2 childOf . Parent, $\leqslant$ 2 childOf . Parent	$\sqcap$	-

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Person	childOf	-	Parent	=	2

22 Inverse Object Properties

`Inverse Object Properties` corresponds to the requirement `R-56-INVERSE-OBJECT-PROPERTIES`. In many cases properties are used bi-directionally and then accessed in the inverse direction, e.g. parent $\equiv$ child$^-$. There should be a way to declare value type, cardinality etc of those inverse relations without having to declare a new property URI. The object property OP1 is an inverse of the object property OP2. Thus, if an individual x is connected by OP1 to an individual y, then y is also connected by OP2 to x, and vice versa.

22.1 Simple Example

$$parentOf \equiv childOf^-$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	parentOf	childOf	-	inverse	

22.2 Simple Example

$$\text{captainOf} \equiv \text{hasCaptain}^-$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	captainOf	hasCaptain	-	inverse	

23 Transitive Object Properties

`Transitive Object Properties` corresponds to the requirement `R-63-TRANSITIVE-OBJECT-PROPERTIES`. Transitivity is a special form of complex role inclusion. An object property transitivity axiom states that the object property is transitive — that is, if an individual x is connected by the object property to an individual y that is connected by the object property to an individual z, then x is also connected by the object property to z.

23.1 Simple Example

$$\text{ancestorOf} \circ \text{ancestorOf} \sqsubseteq \text{ancestorOf}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	ancestorOf, ancestorOf	ancestorOf	-	$\sqsubseteq$	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	ancestorOf	-	-	transitive	

24 Symmetric Object Properties

`Symmetric Object Properties` corresponds to the requirement `R-61-SYMMETRIC-OBJECT-PROPERTIES`. A role is symmetric if it is equivalent to its own inverse [4]. An object property symmetry axiom states that the object property expression OPE is symmetric - that is, if an individual x is connected by OPE to an individual y, then y is also connected by OPE to x.

24.1 Simple Example

The property `marriedTo` is symmetric:

$$married To \equiv married To^{-}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	marriedTo	marriedTo	-	inverse	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	marriedTo	-	-	symmetric	

25 Asymmetric Object Properties

`Asymmetric Object Properties` corresponds to the requirement `R-62-ASYMMETRIC-OBJECT-PROPERTIES`. A role is asymmetric if it is disjoint from its own inverse [4]. An object property asymmetry axiom AsymmetricObjectProperty(OPE) states that the object property expression OPE is asymmetric - that is, if an individual x is connected by OPE to an individual y, then y cannot be connected by OPE to x.

25.1 Simple Example

The property `parentOf` is asymmetric:

$$parentOf \sqsubseteq \neg parentOf^{-}$$

alternatively:

$$parentOf \sqcap parentOf^{-} \sqsubseteq \bot$$

alternatively:

$$\text{disjoint(parentOf, parentOf}^{-}\text{)}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	$parentOf^{-}$	parentOf	-	inverse	
property	$\top$	parentOf	$parentOf^{-}$	-	$\neq$	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	parentOf	-	-	asymmetric	

25.2 Simple Example

Child parent relations (dbo:child) cannot be symmetric.

25.3 Simple Example

Person birth place relations (dbo:birthPlace) cannot be symmetric.

26 Class-Specific Reflexive Object Properties

Using DL terminology `Class-Specific Reflexive Object Properties` is called local reflexivity - a set of individuals (of a specific class) that are related to themselves via a given role [4].

26.1 Simple Example

$$\text{TalkingToThemselves} \sqsubseteq \exists \text{ talksTo .Self.}$$

Set of individuals that are talking to themselves.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	TalkingToThemselves	talksTo	-	Self	$\exists$	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\exists$ talksTo .Self	talksTo	-	-	reflexive	

27 Reflexive Object Properties

`Reflexive Object Properties` (reflexive roles, global reflexivity in DL) corresponds to the requirement `R-59-REFLEXIVE-OBJECT-PROPERTIES`. Global reflexivity can be expressed by imposing local reflexivity on the top concept [4].

27.1 Simple Example

Each individual knows itself:

$$\top \sqsubseteq \exists \text{ knows . Self}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\exists$ knows . Self	knows	-	Self	$\exists$	
class	$\top$	-	-	$\top$, $\exists$ knows . Self	$\sqsubseteq$	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	knows	-	-	reflexive	

28 Irreflexive Object Properties

`Irreflexive Object Properties` (irreflexive roles in DL) corresponds to the requirement `R-60-IRREFLEXIVE-OBJECT-PROPERTIES`. A role is irreflexive if it is never locally reflexive [4]. An object property irreflexivity axiom IrreflexiveObjectProperty(OPE) states that the object property expression OPE is irreflexive - that is, no individual is connected by OPE to itself.

28.1 Simple Example

$$\top \sqsubseteq \neg\ \exists married To.Self$$

alternatively:

$$\exists married To.Self \sqsubseteq \bot$$

alternatively:

$married To \sqcap married To^{-} \sqsubseteq \bot$ (without special self-concept Self)

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	∃ marriedTo . Self	knows	-	Self	∃	
class	¬ ∃ marriedTo . Self	-	-	∃ marriedTo . Self	¬	
class	⊤	-	-	⊤, ¬ ∃ marriedTo . Self	⊑	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	⊤	marriedTo	-	-	irreflexive	

28.2 Simple Example

A resource cannot be its own parent (dbo:parent).

28.3 Simple Example

A resource cannot be its own child (dbo:child).

28.4 Class-Specific Irreflexive Object Properties

A property is *irreflexive* if it is never locally reflexive [4]. An object property irreflexivity axiom states that the object property *OP* is irreflexive - that is, no individual is connected by *OP* to itself. *Class-Specific Irreflexive Object Properties* are object properties which are irreflexive within a given context, e.g. a class.

28.5 Simple Example

Persons cannot be married to themselves.

$$\text{Person} \sqsubseteq \neg\ \exists married To.Self$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	∃ marriedTo . Self	knows	-	Self	∃	
class	¬ ∃ marriedTo . Self	-	-	∃ knows . Self	¬	
class	Person	-	-	Person, ¬ ∃ marriedTo . Self	⊑	

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Person	marriedTo	-	-	irreflexive	

29 Context-Specific Property Groups

`Context-Specific Property Groups` corresponds to the requirement `R-66-PROPERTY-GROUPS` to group data and object properties.

29.1 Simple Example

```
# ShEx:
<Human> {
    (
        foaf:name xsd:string ,
        foaf:givenName xsd:string
    ) ,
    (
        foaf:mbox IRI ,
        foaf:homepage foaf:Document
    ) }
```

- 1. group: 1 foaf:name (range: xsd:string) and 1 foaf:givenName (range: xsd:string) and (,)
- 2. group: 1 foaf:mbox (range: IRI) and 1 foaf:homepage (range: foaf:Document)

$$\begin{aligned}
\text{Human} &\equiv \text{M} \sqcap \text{N} \\
\text{M} &\equiv \text{C} \sqcap \text{F} \\
\text{C} &\sqsubseteq \geqslant 1\ \text{name . string} \sqcap \leqslant 1\ \text{name . string} \\
\text{F} &\sqsubseteq \geqslant 1\ \text{givenName . string} \sqcap \leqslant 1\ \text{givenName . string} \\
\text{N} &\equiv \text{I} \sqcap \text{L} \\
\text{I} &\sqsubseteq \geqslant 1\ \text{mbox . string} \sqcap \leqslant 1\ \text{mbox . string} \\
\text{L} &\sqsubseteq \geqslant 1\ \text{homepage . Document} \sqcap \leqslant 1\ \text{homepage . Document}
\end{aligned}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Human	-	-	M, N	⊓	-
class	M	-	-	C, F	⊓	-
class	C	-	-	A, B	⊓	-
property	A	foaf:name	-	string	⩾	1
property	B	foaf:name	-	string	⩽	1
class	F	-	-	D, E	⊓	-
property	D	foaf:givenName	-	string	⩾	1
property	E	foaf:givenName	-	string	⩽	1
class	N	-	-	I, L	⊓	-
class	I	-	-	G, H	⊓	-
property	G	foaf:mbox	-	string	⩾	1
property	H	foaf:mbox	-	string	⩽	1
class	L	-	-	J, K	⊓	-
property	J	foaf:homepage	-	foaf:Document	⩾	1
property	K	foaf:homepage	-	foaf:Document	⩽	1

30 Context-Specific Exclusive OR of Properties

`Context-Specific Exclusive OR of Properties` correponds to the requirement `R-11-CONTEXT-SPECIFIC-EXCLUSIVE-OR-OF-PROPERTIES`. Exclusive or is a logical operation that outputs true whenever both inputs differ (one is true, the other is false). This constraint is generally expressed in DL as follows:

$$C \sqsubseteq (\neg A \sqcap B) \sqcup (A \sqcap \neg B)$$

and alternatively:

$$C \sqsubseteq (A \sqcup B) \sqcap \neg(A \sqcap B)$$

30.1 Simple Example

```
# ShEx:
<Human> { (
    foaf:name xsd:string |
    foaf:givenName xsd:string ) }
```

```
# OWL 2:
Human owl:disjointUnionOf ( :CC1 :CC2 ) .

CC1 rdfs:subClassOf [
    a owl:Restriction ;
    owl:onProperty foaf:name ;
    owl:someValuesFrom xsd:string ] .
CC2 rdfs:subClassOf [
    a owl:Restriction ;
    owl:onProperty foaf:givenName ;
    owl:someValuesFrom xsd:string ] .
```

– 1 foaf:name (range xsd:string) XOR 1 foaf:givenName (range xsd:string)

$$\text{Human} \sqsubseteq (\neg\ \text{A} \sqcap \text{B}) \sqcup (\text{A} \sqcap \neg\ \text{B})$$
$$\text{A} \sqsubseteq \geqslant 1\ \text{name}\ .\ \text{string} \sqcap \leqslant 1\ \text{name}\ .\ \text{string}$$
$$\text{B} \sqsubseteq \geqslant 1\ \text{givenName}\ .\ \text{string} \sqcap \leqslant 1\ \text{givenName}\ .\ \text{string}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Human	-	-	$\neg$ A $\sqcap$ B, A $\sqcap$ $\neg$ B	$\sqcup$	
class	A $\sqcap$ $\neg$ B	-	-	A, $\neg$ B	$\sqcap$	
class	$\neg$ A $\sqcap$ B	-	-	$\neg$ A, B	$\sqcap$	
class	$\neg$ A	-	-	A	$\neg$	
class	A	-	-	$\geqslant$ 1 name.string, $\leqslant$ 1 name.string	$\sqcap$	-
property	$\geqslant$ 1 name.string	name	-	string	$\geqslant$	1
property	$\leqslant$ 1 name.string	name	-	string	$\leqslant$	1
class	$\neg$ B	-	-	B	$\neg$	
class	B	-	-	$\geqslant$ 1 givenName.string, $\leqslant$ 1 givenName.string	$\sqcap$	-
property	$\geqslant$ 1 givenName.string	givenName	-	string	$\geqslant$	1
property	$\leqslant$ 1 givenName.string	givenName	-	string	$\leqslant$	1

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	= 1 name.string	name	-	string	=	1
property	= 1 givenName.string	givenName	-	string	=	1
class	Human	-	-	= 1 name.string, = 1 givenName.string	XOR	

30.2 Simple Example

```
# ShEx:
<FeelingForce> { (
    attackingBySword xsd:boolean |
    attackingByForce xsd:boolean ) }
```

```
# OWL 2 DL:
FeelingForce owl:disjointUnionOf ( CC1 CC2 ) .
CC1 rdfs:subClassOf [
    a owl:Restriction ;
    owl:onProperty attackingBySword ;
    owl:someValuesFrom xsd:boolean ] .
CC2 rdfs:subClassOf [
    a owl:Restriction ;
    owl:onProperty attackingByForce ;
    owl:someValuesFrom xsd:boolean ] .
```

This means that `FeelingForce` individuals are individuals having one `attackingBySword` relationship (range xsd:boolean) or one `attackingByForce` relationship (range xsd:boolean) but not both.

$$\begin{aligned} \text{A} &\equiv \exists \text{ attackingBySword.xsd:boolean} \\ \text{B} &\equiv \exists \text{ attackingByForce.xsd:boolean} \\ \text{FeelingForce} &\sqsubseteq (\neg \text{ A} \sqcap \text{B}) \sqcup (\text{A} \sqcap \neg \text{ B}) \end{aligned}$$

30.3 Simple Example

$$Person \sqsubseteq ((Male \sqcap \neg Female) \sqcup (\neg Male \sqcap Female))$$

30.4 Complex Example

```
# ShEx:
<Human> { (
    foaf:name xsd:string | foaf:givenName xsd:string+ ,
    foaf:familyName xsd:string ) }
```

- 1 foaf:name (range xsd:string) XOR 1-n foaf:givenName (range xsd:string)
- and
- 1 foaf:familyName (range xsd:string)

Individuals matching the 'Human' data shape:

```
:Han
    foaf:name "Han Solo" ;
    foaf:familyName "Solo" .
:Anakin
    foaf:givenName "Anakin" ;
    foaf:givenName "Darth" ;
    foaf:familyName "Skywalker" .
```

Individual not matching the 'Human' data shape:

```
:Anakin
    foaf:name "Anakin Skywalker" ;
    foaf:givenName "Anakin" ;
    foaf:familyName "Skywalker" .
```

31 Context-Specific Inclusive OR of Properties

`Context-Specific Inclusive OR of Properties` corresponds to the requirement `R-202-CONTEXT-SPECIFIC-INCLUSIVE-OR-OF-PROPERTIES`. Inclusive or is a logical connective joining two or more predicates that yields the logical value "true" when at least one of the predicates is true. The context can be a class, i.e., the constraint applies for individuals of this specific class.

31.1 Simple Example

- 1 foaf:name (range: xsd:string) OR 1 foaf:givenName (range: xsd:string)
- it is also possible that both 1 foaf:name and 1 foaf:givenName are stated
- context: class Human

$$\text{Human} \sqsubseteq \text{A} \sqcup \text{B}$$
$$\text{A} \sqsubseteq \geqslant 1 \text{ name . string} \sqcap \leqslant 1 \text{ name . string}$$
$$\text{B} \sqsubseteq \geqslant 1 \text{ givenName . string} \sqcap \leqslant 1 \text{ givenName . string}$$

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	= 1 name.string	name	-	string	=	1
property	= 1 givenName.string	givenName	-	string	=	1
class	Human	-	-	= 1 name.string, = 1 givenName.string	⊔	

32 Context-Specific Exclusive OR of Property Groups

`Context-Specific Exclusive OR of Property Groups` corresponds to the requirement: `R-13-DISJOINT-GROUP-OF-PROPERTIES-CLASS-SPECIFIC`. Exclusive or is a logical operation that outputs true whenever both inputs differ (one is true, the other is false). Only one of multiple property groups leads to valid data.

32.1 Simple Example

```
# ShEx:
<Human> {
    (
        foaf:name xsd:string ,
        foaf:givenName xsd:string )
    |
    (
        foaf:mbox IRI ,
        foaf:homepage foaf:Document ) }
```

- 1. group XOR 2. group
- 1. group: 1 foaf:name (range: xsd:string) and 1 foaf:givenName (range: xsd:string)
- 2. group: 1 foaf:mbox (range: IRI) and 1 foaf:homepage (range: foaf:Document)
- context: class Human

```
# valid data:
Thomas
    a Human ;
    foaf:mbox <thomas.bosch@gesis.org> ;
    foaf:homepage <http://purl.org/net/thomasbosch> .
```

```
# invalid data:
Thomas
    a Human ;
    foaf:name 'Thomas Bosch' ;
    foaf:givenName 'Thomas' ;
    foaf:mbox <thomas.bosch@gesis.org> ;
    foaf:homepage <http://purl.org/net/thomasbosch> .
```

$$\text{Human} \sqsubseteq (\neg\ \text{E} \sqcap \text{F}) \sqcup (\text{E} \sqcap \neg\ \text{F})$$
$$\text{E} \equiv \text{A} \sqcap \text{B}$$
$$\text{F} \equiv \text{C} \sqcap \text{D}$$
$$\text{A} \sqsubseteq \geqslant 1\ \text{name}\ .\ \text{string} \sqcap \leqslant 1\ \text{name}\ .\ \text{string}$$
$$\text{B} \sqsubseteq \geqslant 1\ \text{givenName}\ .\ \text{string} \sqcap \leqslant 1\ \text{givenName}\ .\ \text{string}$$
$$\text{C} \sqsubseteq \geqslant 1\ \text{mbox}\ .\ \text{IRI} \sqcap \leqslant 1\ \text{mbox}\ .\ \text{IRI}$$
$$\text{D} \sqsubseteq \geqslant 1\ \text{homepage}\ .\ \text{Document} \sqcap \leqslant 1\ \text{homepage}\ .\ \text{Document}$$

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Human	-	-	E, F	XOR	
class	E	-	-	= 1 name.string, = 1 givenName.string	⊓	
class	F	-	-	= 1 mbox.IRI, = 1 homepage.Document	⊓	
property	= 1 name.string	name	-	string	=	1
property	= 1 givenName.string	givenName	-	string	=	1
property	= 1 mbox.IRI	mbox	-	IRI	=	1
property	= 1 homepage.Document	homepage	-	Document	=	1

33 Context-Specific Inclusive OR of Property Groups

At least one property group must match for individuals of a specific context. Context may be a class, a shape, or an application profile.

33.1 Simple Example

- 1. group OR 2. group
- 1. group: 1 foaf:firstName (range: xsd:string) and 1 foaf:lastName (range: xsd:string)
- 2. group: 1 foaf:givenName (range: xsd:string) and 1 foaf:familyName (range: xsd:string)
- context: class Person

```
# valid data:
:Anakin
    a :Person ;
    foaf:firstName 'Anakin' ;
    foaf:lastName 'Skywalker' ;
    foaf:givenName 'Anakin' ;
    foaf:familyName 'Skywalker' .
```

```
# invalid data:
:Anakin
    a :Person .
```

$$\text{Human} \sqsubseteq \text{E} \sqcup \text{F}$$
$$\text{E} \equiv \text{A} \sqcap \text{B}$$
$$\text{F} \equiv \text{C} \sqcap \text{D}$$
$$\text{A} \sqsubseteq \geqslant 1 \text{ name . string} \sqcap \leqslant 1 \text{ name . string}$$
$$\text{B} \sqsubseteq \geqslant 1 \text{ givenName . string} \sqcap \leqslant 1 \text{ givenName . string}$$
$$\text{C} \sqsubseteq \geqslant 1 \text{ mbox . IRI} \sqcap \leqslant 1 \text{ mbox . IRI}$$
$$\text{D} \sqsubseteq \geqslant 1 \text{ homepage . Document} \sqcap \leqslant 1 \text{ homepage . Document}$$

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Human	-	-	E, F	$\sqcup$	
class	E	-	-	= 1 name.string, = 1 givenName.string	$\sqcap$	
class	F	-	-	= 1 mbox.IRI, = 1 homepage.Document	$\sqcap$	
property	= 1 name.string	name	-	string	=	1
property	= 1 givenName.string	givenName	-	string	=	1
property	= 1 mbox.IRI	mbox	-	IRI	=	1
property	= 1 homepage.Document	homepage	-	Document	=	1

34 Allowed Values

`Allowed Values` corresponds to the requirements: `R-30-ALLOWED-VALUES-FOR-RDF-OBJECTS` and `R-37-ALLOWED-VALUES-FOR-RDF-LITERALS`. It is a common

requirement to narrow down the value space of a property by an exhaustive enumeration of the valid values (both literals or resource). This is often rendered in drop down boxes or radio buttons in user interfaces. Allowed values for properties

- must be these IRIs,
- must be IRIs matching specific patterns,
- must be IRIs matching one of multiple patterns,
- must be (any) literals,
- must be literals of a list of allowed literals (e.g. "red" "blue" "green"),
- must be typed literals of this type (e.g. XML dataType).

34.1 Example

```
# DSP:
descriptionTemplate
    a dsp:DescriptionTemplate ;
    dsp:minOccur "0"^^xsd:nonNegativeInteger ;
    dsp:maxOccur "infinity"^^xsd:string ;
    dsp:resourceClass swrc:Book ;
    dsp:statementTemplate [
        a dsp:NonLiteralStatementTemplate ;
        dsp:minOccur "0"^^xsd:nonNegativeInteger ;
        dsp:maxOccur "infinity"^^xsd:string ;
        dsp:property dcterms:subject ;
        dsp:nonLiteralConstraint [
            a dsp:NonLiteralConstraint ;
            dsp:valueClass skos:Concept ;
            dsp:valueURI ComputerScience, SocialScience, Librarianship ] ] .
```

```
# OWL2:
dcterms:subject rdfs:range ObjectOneOf .
# EquivalentClasses( ObjectOneOf ObjectOneOf( ComputerScience SocialScience Librarianship ) )
ObjectOneOf owl:equivalentClass [
    a owl:Class ;
    owl:oneOf ( ComputerScience SocialScience Librarianship ) ] .
```

The range of the object property `dcterms:subject` must consist of the individuals `ComputerScience` `SocialScience` `Librarianship` which are of the class `skos:Concept`:

$$\top \sqsubseteq \forall \text{ dcterms:subject . skos:Concept} \sqcap \\ (\{ComputerScience\} \sqcup \{\text{SocialScience}\} \sqcup \{\text{Librarianship}\})$$

34.2 Simple Example

$$\text{Beatle} \equiv \{\text{john}\} \sqcup \{\text{paul}\} \sqcup \{\text{george}\} \sqcup \{\text{ringo}\}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Beatle	-	-	$\{john\}$, $\{paul\}$, $\{george\}$, $\{ringo\}$	$\sqcup$	

34.3 Simple Example

$$\{\text{ComputerScience}\} \sqcup \{\text{SocialScience}\} \sqcup \{\text{Librarianship}\}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	complex concept	-	-	{ComputerScience}, {SocialScience}, {Librarianship}	$\sqcup$	-

34.4 Simple Example

Jedis have blue, green, or white laser swords.

```
# DSP:
personDescriptionTemplate
    a dsp:DescriptionTemplate ;
    dsp:resourceClass Jedi ;
    dsp:statementTemplate [
        a dsp:LiteralStatementTemplate ;
        dsp:property laserSwordColor ;
        dsp:literalConstraint [
            a dsp:LiteralConstraint ;
            dsp:literal "blue" ;
            dsp:literal "green" ;
            dsp:literal "white"] ] .
```

```
# OWL2:
laserSwordColor rdfs:range laserSwordColors .
laserSwordColors
    a rdfs:Datatype .
    owl:oneOf ( "blue" "green" "white" ) .
```

```
# ReSh:
Jedi a rs:ResourceShape ;
    rs:property [
        rs:name "laserSwordColor" ;
        rs:propertyDefinition laserSwordColor ;
        rs:allowedValue "blue" , "green" , "white" ;
        rs:occurs rs:Exactly-one ; ] .
```

```
# ShEx:
Jedi {
    laserSwordColor ('blue' 'green' 'white') }
```

```
# Jedi individuals:
Yoda
    laserSwordColor 'blue' .
```

$$\text{Jedi} \equiv \exists \text{ laserSwordColor.}\{\text{blue, green, white}\}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Jedi	laserSwordColor	-	LaserSwordColor	$\exists$	-
class	LaserSwordColor	-	-	$\{blue\}$, $\{green\}$, $\{white\}$	$\sqcup$	-

35 Not Allowed Values

`Not Allowed Values` corresponds to the requirements `R-33-NEGATIVE-OBJECT-CONSTRAINTS` and `R-200-NEGATIVE-LITERAL-CONSTRAINTS`. A matching triple has any literal / object except those explicitly excluded.

35.1 Simple Example

Siths do not have blue, green, or white laser swords.

```
# OWL2:
laserSwordColor rdfs:range negativeLaserSwordColors .
NegativeLaserSwordColors
    a rdfs:Datatype .
    owl:complementOf laserSwordColors .
laserSwordColors
    a rdfs:Datatype .
    owl:oneOf ( "blue" "green" "white" ) .
```

```
# ShEx:
Sith {
    ! laserSwordColor ('blue' 'green' 'white') }
```

```
# Sith individuals:
DarthSidious
    laserSwordColor 'red' .
```

$$\text{Sith} \equiv \neg \exists \text{ laserSwordColor . } \{\text{blue, green, white}\}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	∃ laserSwordColor . LaserSwordColor	laserSwordColor	-	LaserSwordColor	∃	-
class	Sith	-	-	∃ laserSwordColor . LaserSwordColor	¬	-
class	LaserSwordColor	-	-	$\{blue\}$, $\{green\}$, $\{white\}$	⊔	-

36 Membership in Controlled Vocabularies

`Membership in Controlled Vocabularies` corresponds to the requirements `R-32-MEMBERSHIP-OF-RDF-OBJECTS-IN-CONTROLLED-VOCABULARIES` and `R-39-MEMBERSHIP-OF-RDF-LITERALS-IN-CONTROLLED-VOCABULARIES` Resources can only be members of listed controlled vocabularies.

36.1 Simple Example

```
# DSP:
bookDescriptionTemplate
    a dsp:DescriptionTemplate ;
    dsp:resourceClass swrc:Book ;
    dsp:statementTemplate [
        a dsp:NonLiteralStatementTemplate ;
```

```
        dsp:property dcterms:subject ;
        dsp:nonLiteralConstraint [
            a dsp:NonLiteralConstraint ;
            dsp:valueClass skos:Concept ;
            dsp:vocabularyEncodingScheme BookSubjects, BookTopics, BookCategories ] ] .
```

`skos:Concept` resources can only be members of the listed controlled vocabularies.

$$A \equiv Concept \sqcap B$$
$$B \equiv \forall inScheme.C$$
$$C \equiv ConceptScheme \sqcap (\{BookSubjects\} \sqcup \{BookTopics\} \sqcup \{BookCategories\})$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	subject	-	A	range	-
class	A	-	-	Concept, B	$\sqcap$	-
property	B	inScheme	-	C	$\forall$	-
class	C	-	-	ConceptScheme, D	$\sqcap$	-
class	D	-	-	$\{BookSubjects\}$, $\{BookTopics\}$, $\{BookCategories\}$	$\sqcup$	-

```
# valid data:
ArtficialIntelligence
    a swrc:Book ;
    dcterms:subject ComputerScience .
ComputerScience
    a skos:Concept ;
    dcam:memberOf BookSubjects ;
    skos:inScheme BookSubjects .
BookSubjects
    a skos:ConceptScheme .
```

```
# invalid data:
ArtficialIntelligence
    a swrc:Book ;
    dcterms:subject ComputerScience .
ComputerScience
    a skos:Concept ;
    dcam:memberOf BooksAboutBirds ;
    skos:inScheme BooksAboutBirds ;
    dcam:memberOf BookSubjects ;
    skos:inScheme BookSubjects .
BookSubjects
    a skos:ConceptScheme .
```

The related subject (`ComputerScience`) is a member of a controlled vocabulary (`BooksAboutBirds`) which is not part of the list of allowed controlled vocabularies.

37 IRI Pattern Matching

`IRI Pattern Matching` corresponds to the requirements

– R-21-IRI-PATTERN-MATCHING-ON-RDF-SUBJECTS,
– R-22-IRI-PATTERN-MATCHING-ON-RDF-OBJECTS and
– R-23-IRI-PATTERN-MATCHING-ON-RDF-PROPERTIES

indicating IRI pattern matching on subjects, properties, and objects.

Not expressible in DL!

38 Literal Pattern Matching

Literal Pattern Matching corresponds to the requirement R-44-PATTERN-MATCHING-ON-RDF-LITERALS. indicating pattern matching on literals.

38.1 Simple Example

```
# OWL 2 DL (functional-style syntax):
Declaration( Datatype( SSN ) )
DatatypeDefinition(
    SSN
    DatatypeRestriction( xsd:string xsd:pattern "[0-9]{3}-[0-9]{2}-[0-9]{4}" ) )
DataPropertyRange( hasSSN SSN )
```

```
# OWL 2 DL (turtle syntax):
SSN
    a rdfs:Datatype ;
    owl:equivalentClass [
        a rdfs:Datatype ;
        owl:onDatatype xsd:string ;
        owl:withRestrictions (
            [ xsd:pattern "[0-9]{3}-[0-9]{2}-[0-9]{4}" ] ) ] .
hasSSN rdfs:range SSN .
```

A social security number is a string that matches the given regular expression. The second axiom defines SSN as an abbreviation for a datatype restriction on xsd:string. The first axiom explicitly declares SSN to be a datatype. The datatype SSN can be used just like any other datatype; for example, it is used in the third axiom to define the range of the hasSSN property.

```
# valid data:
TimBernersLee
    hasSSN "123-45-6789"^^SSN .
```

```
# invalid data:
TimBernersLee
    hasSSN "123456789"^^SSN .
```

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	SSN	-	-	xsd:string	xsd:pattern	'[0-9]3-[0-9]2-[0-9]4'

38.2 Simple Example

There are multiple use cases associated with the requirement to match literals according to given patterns.

Luke's droids can only have the numbers "R2-D2" or "C-3PO". The universal restriction part of this constraint can be expressed by OWL 2 DL: `LukesDroids` $\sqsubseteq \forall$ `droidNumber.DroidNumber`. The restriction of the datatype `DroidNumber`, however, cannot be expressed in DL, but OWL 2 DL can be used anyway:

```
DroidNumber
    a rdfs:Datatype ;
    owl:equivalentClass [
        a rdfs:Datatype ;
        owl:onDatatype xsd:string ;
        owl:withRestrictions (
            [ xsd:pattern "R2-D2|C-3PO" ] ) ] .
```

The second axiom defines `DroidNumber` as an abbreviation for a datatype restriction on `xsd:string`. The first axiom explicitly declares `DroidNumber` to be a datatype. The datatype `DroidNumber` can be used just like any other datatype like in the universal restriction above. The literal pattern matching constraint validates `DroidNumber` literals according to the stated regular expression causing a constraint violation for the triples `Luke hasDroid Droideka` and `Droideka droidNumber "Droideka"^^DroidNumber`, but not for the triples `Luke hasDroid R2-D2` and `R2-D2 droidNumber "R2-D2"^^DroidNumber`.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	LukesDroids	droidNumber	-	DroidNumber	∀	-
class	DroidNumber	-	-	xsd:string	xsd:pattern	'R2-D2|C-3PO'

39 Negative Literal Pattern Matching

`Negative Literal Pattern Matching` corresponds to the requirement `R-44-PATTERN-MATCHING-ON-RDF-LITERALS` indicating negative pattern matching on literals.

```
# examples:
1. dbo:isbn format is different '!' from '^([iIsSbBnN 0-9-])*$'
2. dbo:postCode format is different '!' from '^[0-9]{5}$'
3. foaf:phone contains any letters ('[A-Za-z]')
```

39.1 Example

```
# test binding (DQTP):
dbo:isbn format is different '!' from '^([iIsSbBnN 0-9-])*$'

P1 => dbo:isbn
NOP => !
REGEX => '^([iIsSbBnN 0-9-])*$'
```

```
# DQTP:
SELECT DISTINCT ?s WHERE { ?s %%P1%% ?value .
    FILTER ( %%NOP%% regex(str(?value), %%REGEX%%) ) }
```

- MATCH Pattern [3]
- P1 is the property we need to check against REGEX and NOP can be a not operator (!) or empty.

```
# valid data:
FoundationsOfSWTechnologies
    dbo:isbn 'ISBN-13 978-1420090505' .
```

```
# invalid data:
HandbookOfSWTechnologies
    dbo:isbn 'DOI 10.1007/978-3-540-92913-0' .
```

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	dbo:isbn	-	$\neg$ A	domain	-
class	$\neg$ A	-	-	A	$\neg$	-
class	A	-	-	xsd:string	regex	'^([iIsSbBnN 0-9-])*'

40 Literal Value Comparison

`Literal Value Comparison` corresponds to the requirement `R-43-LITERAL-VALUE-COMPARISON`. Examples are:

- dbo:deathDate before '<' dbo:birthDate
- dbo:releaseDate after '>' dbo:latestReleaseDate
- dbo:demolitionDate before '<' dbo:buildingStartDate

40.1 Simple Example

```
# DQTP:
SELECT ?s WHERE {
    ?s %%P1%% ?v1 .
    ?s %%P2%% ?v2 .
    FILTER ( ?v1 %%OP%% ?v2 ) }
```

This constraint corresponds to the COMP Pattern [3]. Depending on the property semantics, there are cases where two different literal values must have a specific ordering with respect to an operator. P1 and P2 are the datatype properties we need to compare and OP is the comparison operator (<, <=, >, >=, =, !=).

```
# test binding (DQTP):
dbo:deathDate before ‘<’ dbo:birthDate

P1 => dbo:deathDate
P2 => dbo:birthDate
OP => <
```

```
# valid data:
:AlbertEinstein
    dbo:birthDate ’1879-03-14’^^xsd:date ;
    dbo:deathDate ’1955-04-18’^^xsd:date .
```

```
# invalid data:
:NeilArmstrong
    dbo:birthDate ’2012-08-25’^^xsd:date ;
    dbo:deathDate ’1930-08-05’^^xsd:date .
```

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	dbo:birthDate	dbo:deathDate	xsd:date	$>$	-

40.2 Simple Example

Duplicate strings are not allowed:

```
dc:subject "foo"@en
dc:subject "foo"@fr
dc:subject "bar"@fr
dc:subject "foo"
```

There are no identical strings.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	dc:subject	dc:subject	xsd:string	$\neq$	-

41 Negative Literal Ranges

`Negative Literal Ranges` corresponds to the requirement `R-142-NEGATIVE-RANGES-OF-RDF-LITERAL-VALUES`. The literal value of a resource (having a certain type) must (not) be within a specific range. P1 is a data property of an instance of class T1 and its literal value must be between the range of [Vmin,Vmax] or outside (‘!’).

41.1 Simple Example

– dbo:height of a dbo:Person is not within [0.4,2.5]

41.2 Simple Example

– geo:lat of a spatial:Feature is not within [-90,90]

41.3 Simple Example

– geo:long of a gml:Feature must be in range [-180,180]

42 Literal Ranges

`Literal Ranges` corresponds to the requirement `R-45-RANGES-OF-RDF-LITERAL-VALUES`. P1 is a data property (of an instance of class C1) and its literal value must be between the range of [V_{min},V_{max}].

42.1 Example

```
# OWL 2 DL (functional-style syntax):
Declaration( Datatype( NumberPlayersPerWorldCupTeam ) )
DatatypeDefinition(
    NumberPlayersPerWorldCupTeam
    DatatypeRestriction(
        xsd:nonNegativeInteger
        xsd:minInclusive "1"^^xsd:nonNegativeInteger
        xsd:maxInclusive "23"^^xsd:nonNegativeInteger ) )
DataPropertyRange( position NumberPlayersPerWorldCupTeam )
```

```
# OWL 2 DL (turtle syntax):
NumberPlayersPerWorldCupTeam
    a rdfs:Datatype ;
    owl:equivalentClass [
        a rdfs:Datatype ;
        owl:onDatatype xsd:nonNegativeInteger ;
        owl:withRestrictions (
            [ xsd:minInclusive "1"^^xsd:nonNegativeInteger ]
            [ xsd:maxInclusive "23"^^xsd:nonNegativeInteger ] ) ] .
position rdfs:range NumberPlayersPerWorldCupTeam .
```

The data range 'NumberPlayersPerWorldCupTeam' contains the non negative integers 1 to 23, as each world cup team can only have 23 football players at most.

```
# valid data:
MarioGoetze
    position "19"^^:NumberPlayersPerWorldCupTeam .
```

```
# invalid data:
MarioGoetze
    position "99"^^:NumberPlayersPerWorldCupTeam .
```

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	NumberPlayersPerWorldCupTeam	-	-	$\geqslant$1, $\leqslant$23	$\sqcap$	-
class	$\geqslant$1	-	-	xsd:nonNegativeInteger	$\geqslant$	1
class	$\leqslant$23	-	-	xsd:nonNegativeInteger	$\leqslant$	23

43 Ordering

`Ordering` corresponds to the requirements `R-121-SPECIFY-ORDER-OF-RDF- RESOURCES` and `R-217-DEFINE-ORDER-FOR-FORMS/DISPLAY`. With this constraint objects of object properties can be ordered as well as literals of data properties and objects of object properties.

43.1 Simple Example

Define the order of the property `listElement` for the class `rdf:List`. List elements are of datatype `xsd:string`.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	rdf:List	listElement	-	xsd:string	order	1

43.2 Simple Example

Property p should be ordered, i.e., point to a list (rdf:List) of values.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	p	-	-	ordered property	-

44 Validation Levels

`Validation Levels` corresponds to the requirements

- `R-205-VARYING-LEVELS-OF-ERROR`,
- `R-135-CONSTRAINT-LEVELS`,
- `R-158-SEVERITY-LEVELS-OF-CONSTRAINT-VIOLATIONS`, and
- `R-193-MULTIPLE-CONSTRAINT-VALIDATION-EXECUTION-LEVELS`.

Different levels of severity, priority should be assigned to constraints. Possible validation levels could be: informational, warning, error, fail, should, recommended, must, may, optional, closed (only this) constraints, open (at least this).

Not expressible in DL!

45 String Operations

`String Operations` corresponds to the requirement `R-194-PROVIDE-STRING-FUNCTIONS-FOR-RDF-LITERALS`. Some constraints require building new strings out of other strings. Calculating the string length would also be another constraint of this type.

45.1 Simple Example

The length of strings of the property `hasISBN` for the context class `Book` must be exactly 3.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Book	hasISBN	-	xsd:string	length	3

46 Context-Specific Valid Classes

`Context-Specific Valid Classes` corresponds to the requirements `R-209-VALID-CLASSES`. What types of resources (rdf:type) are valid in a specific context? Context can be an input stream, a data creation function, or an API.

46.1 Simple Example

Within the context `AP` (application profile) the classes `Book` and `Paper` are valid.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	AP	-	-	Book, Paper	context-specific valid classes	true

47 Context-Specific Valid Properties

`Context-Specific Valid Properties` corresponds to the requirement `R-210-VALID-PROPERTIES`. What properties can be used within this context? Context can be an data receipt function, data creation function, or API.

47.1 Simple Example

Within the context `AP` (application profile) the properties `dcterms:subject` and `dcterms:title` are invalid.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	AP	dcterms:subject, dcterms:title	-	-	context-specific valid properties	false

48 Default Values

`Default Values` corresponds to the requirements `R-31-DEFAULT-VALUES-OF-RDF-OBJECTS` and `R-38-DEFAULT-VALUES-OF-RDF-LITERALS`. Default values for objects and literals are inferred automatically. It should be possible to declare the default value for a given property, e.g. so that input forms can be pre-populated and to insert a required property that is missing in a web service call.

48.1 Simple Example

Jedis have only 1 blue laser sword per default. Siths, in contrast, normally have 2 red laser swords.

```
# rule (SPIN)
# -----------
owl:Thing
    spin:rule [
        a sp:Construct ;
            sp:text """
                CONSTRUCT {
                    ?this laserSwordColor "blue"^^xsd:string ;
                    ?this numberLaserSwords "1"^^xsd:nonNegativeInteger .
                }
                WHERE {
                    ?this a Jedi .
                } """ ; ] .
owl:Thing
    spin:rule [
        a sp:Construct ;
            sp:text """
                CONSTRUCT {
                    ?this laserSwordColor "red"^^xsd:string ;
                    ?this numberLaserSwords "2"^^xsd:nonNegativeInteger .
                }
                WHERE {
                    ?this a Sith .
                } """ ; ] .
```

```
# data:
Joda a Jedi .
DarthSidious a Sith .
```

```
# inferred triples:
Joda
    laserSwordColor "blue"^^xsd:string ;
    numberLaserSwords "1"^^xsd:nonNegativeInteger .
DarthSidious
    laserSwordColor "red"^^xsd:string ;
    numberLaserSwords "2"^^xsd:nonNegativeInteger .
```

Not expresible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Jedi	laserSwordColor	-	-	default values	'blue'
property	Jedi	numberLaserSwords	-	-	default values	'1'
property	Sith	laserSwordColor	-	-	default values	'red'
property	Sith	numberLaserSwords	-	-	default values	'2'

49 Mathematical Operations

`Mathematical Operations` corresponds to the requirements

- `R-42-MATHEMATICAL-OPERATIONS` and
- `R-41-STATISTICAL-COMPUTATIONS`.

Examples are:

- adding 2 dates
- add number of days to start date
- area = width * height
- Statistical Computations: average, mean, sum

49.1 Simple Example

Calculate rectangle areas.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Rectangle	area	width, height	xsd:integer	multiplication	-

50 Language Tag Matching

`Language Tag Matching` corresponds to the requirement `R-47-LANGUAGE-TAG-MATCHING`.

50.1 Simple Example

Only English names are allowed.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	name	-	-	language tag	'en'

51 Language Tag Cardinality

`Language Tag Cardinality` corresponds to the requirements

- `R-49-RDF-LITERALS-HAVING-AT-MOST-ONE-LANGUAGE-TAG` and
- `R-48-MISSING-LANGUAGE-TAGS`.

51.1 Simple Example

Check that no language is used more than once per property

```
# DQTP:
SELECT DISTINCT ?s WHERE { ?s %%P1%% ?c
    BIND ( lang(?c) AS ?l )
    FILTER (isLiteral (?c) && lang(?c) = %%V1%%)}
GROUP BY ?s HAVING COUNT (?l) > 1
```

This corresponds to the test pattern `ONELANGPattern` [3]. A literal value should contain at most 1 literal for a language. `P1` is the property containing the literal and `V1` is the language we want to check.

```
# test binding (DQTP):
P1 => foaf:name
V1 => en
```

A single English ("en") `foaf:name`.

```
# valid data:
:LeiaSkywalker
    foaf:name 'Leia Skywalker'@en .
```

```
# invalid data:
:LeiaSkywalker
    foaf:name 'Leia Skywalker'@en ;
    foaf:name 'Leia'@en .
```

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	foaf:name	-	-	language tag exact cardinality	1

52 Whitespace Handling

`Whitespace Handling` corresponds to the requirement `R-50-WHITESPACE-HANDLING-OF-RDF-LITERALS`. Avoid whitespaces in literals neither leading nor trailing white spaces.

52.1 Simple Example

Check if literals of the property `p` do not include whitespaces.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	p	-	-	whitespace	-

53 HTML Handling

`HTML Handling` corresponds to the requirement `R-51-HTML-HANDLING-OF-RDF-LITERALS`. Check if there are no HTML tags included in literals (of specific data properties within the context of specific classes).

53.1 Simple Example

Check if literals of the property `p` of the class `C` do not include HTML tags.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	c	p	-	-	HTML	-

54 Required Properties

`Required Properties` corresponds to the requirement `R-68-REQUIRED-PROPERTIES`.

54.1 Simple Example

For persons the property `hasAncestor` has to be stated pointing to persons.

$$Person \sqsubseteq \exists hasAncestor.Person$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Person	hasAncestor	-	Person	$\exists$	-

55 Optional Properties

`Optional Properties` corresponds to the requirement `R-69-OPTIONAL-PROPERTIES`.

55.1 Simple Example

For persons the property `hasAncestor` pointing to persons is optional.

$$\exists hasAncestor.Person \sqsubseteq Person$$

Same as the definition of domains.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	∃ hasAncestor.Person	hasAncestor	-	Person	∃	-
class	Person	-	-	∃ hasAncestor.Person	⊑	-

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Person	hasAncestor	-	Person	optional	-

56 Repeatable Properties

`Repeatable Properties` corresponds to the requirement `R-70-REPEATABLE-PROPERTIES`.

56.1 Simple Example

The property `commandsVessel` is repeatable for individuals of the class `Captain`.

$$Captain \sqsubseteq \geqslant 1 commandsVessel.\top$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Captain	commandsVessel	-	⊤	⩾	1

syntactic sugar:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	Captain	commandsVessel	-	⊤	repeatable	-

57 Conditional Properties

`Conditional Properties` corresponds to the requirement `R-71-CONDITIONAL-PROPERTIES`.

Multiple conditions are possible:

- universal quantification on object and data properties,
- existential quantification on object and data properties,
- if specific properties are present, then specific other properties also have to be present

57.1 Simple Example

If an individual has a `parentOf` property relationship, then this individual also has a `ancestorOf` property relationship.

$$\text{parentOf} \sqsubseteq \text{ancestorOf}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	parentOf	ancestorOf	$\top$	$\sqsubseteq$	

58 Recommended Properties

`Recommended Properties` corresponds to the requirement `R-72-RECOMMENDED-PROPERTIES`. Which properties are not required but recommended within a particular context.

58.1 Simple Example

Property `p` is recommended to use within the context of the class `C`.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	C	p	-	-	recommended	-

59 Negative Property Constraints

`Negative Property Constraints` corresponds to the requirements

- `R-52-NEGATIVE-OBJECT-PROPERTY-CONSTRAINTS` and
- `R-53-NEGATIVE-DATA-PROPERTY-CONSTRAINTS`.

Instances of a specific class must not have some object property. In OWL 2 DL, this can be expressed as follows: `ObjectComplementOf ( ObjectSomeValuesFrom ( ObjectPropertyExpression owl:Thing ) )`.

59.1 Example

```
# ShEx:
<FeelingForce> {
    feelingForce (true) ,
    attitute xsd:string }
<JediMentor> {
    feelingForce (true) ,
    attitute ('good') ,
    laserSwordColor xsd:string ,
    numberLaserSwords xsd:nonNegativeInteger ,
    mentorOf @<JediStudent> ,
   !studentOf @<JediMentor> }
<JediStudent> {
    feelingForce (true) ,
    attitute ('good') ,
    laserSwordColor xsd:string ,
    numberLaserSwords xsd:nonNegativeInteger ,
   !mentorOf @<JediStudent> ,
    studentOf @<JediMentor> }
```

A matching triple has any predicate except those excluded by the '!' operator.

```
# individuals matching 'FeelingForce' and 'JediMentor' data shapes:
Obi-Wan
    feelingForce true ;
    attitute 'good' ;
    laserSwordColor 'blue' ;
    numberLaserSwords '1'^^xsd:nonNegativeInteger ;
    mentorOf Anakin .
```

```
# individuals matching 'FeelingForce' and 'JediStudent' data shapes:
Anakin
    feelingForce true ;
    attitute 'good' ;
    laserSwordColor 'blue' ;
    numberLaserSwords '1'^^xsd:nonNegativeInteger ;
    studentOf Obi-Wan .
```

$$JediMentor \sqsubseteq \neg(\exists studentOf.JediMentor)$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	∃ studentOf.JediMentor	studentOf	-	JediMentor	∃	-
class	JediMentor	-	-	∃ studentOf.JediMentor	¬	-

60 Handle RDF Collections

`Handle RDF Collections` corresponds to the requirement `R-120-HANDLE-RDF-COLLECTIONS`. Examples are:

- size of collection
- first / last element of list must be a specific literal
- compare elements of collection

– are collections identical?
– actions on RDF lists[14]
– 2. list element equals 'XXX'
– Does the list have more than 10 elements?

60.1 Example

Get 2. list element:

```
# SPIN:
getListItem
    a spin:Function ; rdfs:subClassOf spin:Functions ;
    spin:constraint [
        rdf:type spl:Argument ;
        spl:predicate sp:arg1 ;
        spl:valueType rdf:List ;
        rdfs:comment "list" ; ] ;
    spin:constraint [
        rdf:type spl:Argument ;
        spl:predicate sp:arg2 ;
        spl:valueType xsd:nonNegativeInteger ;
        rdfs:comment "item position (starting with 0)" ; ] ;
    spin:body [
        a sp:SELECT ;
        sp:text """
            SELECT ?item
            WHERE {
                ?arg1 contents/rdf:rest{?arg2}/rdf:first ?item } """ ; ] ;
    spin:returnType rdfs:Resource .
```

```
# data:
Jinn students
     ( Xanatos Kenobi ) .
```

```
# SPIN:
BIND ( getListItem( ?list, "1"xsd:nonNegativeInteger ) AS ?listItem ) .
```

– SPIN function call
– retrieves the 2. item from the list (2. student of Jedi mentor Jinn)

```
# result:
Kenobi
```

Not expressible in DL!

60.2 Simple Example

Append a string list element 'list element' to a list the property `p` of the class `C` is pointing to.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	C	p	-	xsd:string	append list element	'list element'

[14] See http://www.snee.com/bobdc.blog/2014/04/rdf-lists-and-sparql.html

61 Recursive Queries

`Recursive Queries` corresponds to the requirement `R-222-RECURSIVE-QUERIES`. If we want to define Resource Shapes, remember that it is a recursive language (the valueShape of a Resource Shape is in turn another Resource Shape). There is no way to express that in SPARQL without hand-waving "and then you call the function again here" or "and then you embed this operation here" text. The embedding trick doesn't work in the general case because SPARQL can't express recursive queries, e.g. "test that this Issue is valid and all of the Issues that references, recursively". Most SPARQL engines already have functions that go beyond the official SPARQL 1.1 spec. The cost of that sounds manageable.

61.1 Simple Example

```
# ShEx:
IssueShape {
    related @IssueShape*
}
```

$$IssueShape \sqsubseteq \geqslant 1 related.IssueShape$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	IssueShape	related	-	IssueShape	$\geqslant$	1

62 Value is Valid for Datatype

Make sure that a value is valid for its datatype.

62.1 Simple Example

A date is really a date, as an example. SPARQL regex can be used for this purpose.

62.2 Simple Example

```
# SPIN:
FILTER ( datatype( ?shoeSize ) = xsd:nonNegativeInteger )
isNumeric ( ?shoeSize )
```

The datatype of ?showSize is xsd:nonNegativeInteger. The datatype is really numeric.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	shoeSize	-	xsd:nonNegativeInteger	value valid for datatype	-

63 Individual Equality

`Individual equality` states that two different names are known to refer to the same individual [4].

63.1 Simple Example

$$\{julia\} = \{john\}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	{julia}	-	-	{john}	=	-
class	{john}	-	-	{julia}	=	-

64 Individual Inequality

This is by default because of the UNA.

64.1 Simple Example

$$\{julia\} \neq \{john\}$$

Asserts that Julia and John are actually different individuals.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	{julia}	-	-	{john}	≠	-
class	{john}	-	-	{julia}	≠	-

65 Equivalent Properties

An equivalent object properties axiom EquivalentObjectProperties(OPE1 ... OPEn) states that all of the object property expressions OPEi, 1 ⩽ i ⩽ n, are semantically equivalent to each other. This axiom allows one to use each OPEi as a synonym for each OPEj — that is, in any expression in the ontology containing such an axiom, OPEi can be replaced with OPEj without affecting the meaning of the ontology. The axiom EquivalentObjectProperties(OPE1 OPE2) is equivalent to the following two axioms SubObjectPropertyOf(OPE1 OPE2) and SubObjectPropertyOf(OPE2 OPE1).

An equivalent data properties axiom EquivalentDataProperties(DPE1 ... DPEn) states that all the data property expressions DPEi, 1 ⩽ i ⩽ n, are semantically equivalent to each other. This axiom allows one to use each DPEi as a synonym for each DPEj — that is, in any expression in the ontology containing such an axiom, DPEi can be replaced with DPEj without affecting the meaning of the ontology. The axiom EquivalentDataProperties(DPE1 DPE2) can be seen as a syntactic shortcut for the following axiom SubDataPropertyOf(DPE1 DPE2) and SubDataPropertyOf(DPE2 DPE1).

65.1 Simple Example

```
# OWL 2:
hasBrother owl:equivalentProperty hasMaleSibling .
Chris hasBrother Stewie .
Stewie hasMaleSibling Chris .
```

entailments:

```
Chris hasMaleSibling Stewie .
Stewie hasBrother Chris .
```

$$\text{hasBrother} \sqsubseteq \text{hasMaleSibling} \sqcap \text{hasMaleSibling} \sqsubseteq \text{hasBrother}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	hasBrother	hasMaleSibling	-	$\sqsubseteq$	
property	$\top$	hasMaleSibling	has Brother	-	$\sqsubseteq$	

$$hasBrother \equiv hasMaleSibling$$

66 Property Assertions

`Property Assertions` corresponds to the requirement `R-96-PROPERTY-ASSERTIONS` and includes positive property assertions and negative property assertions. A positive object property assertion ObjectPropertyAssertion(OPE a1 a2) states that the individual a1 is connected by the object property expression OPE to the individual a2. A negative object property assertion NegativeObjectPropertyAssertion(OPE a1 a2) states that the individual a1 is not connected by the object property expression OPE to the individual a2. A positive data property assertion DataPropertyAssertion(DPE a lt) states that the individual a is connected by the data property expression DPE to the literal lt. A negative data property assertion NegativeDataPropertyAssertion(DPE a lt) states that the individual a is not connected by the data property expression DPE to the literal lt.

66.1 Simple Example

```
# OWL 2:
NegativeObjectPropertyAssertion( hasSon Peter Meg )
```

Meg is not a son of Peter.

$$hasSon(Peter, Meg)$$

The negation of such an assertion is not necessary, as it's meaningless!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	{Peter}	hasSon	-	{Meg}	$\neq$	-

66.2 Simple Example

```
# OWL 2:
DataPropertyAssertion( :hasAge :Meg "17"^^xsd:integer )
```

Meg is seventeen years old.

$$hasAge(Meg, "17"^^xsd:integer)$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	{Meg}	hasAge	-	-	=	"17"^^ xsd:integer

67 Functional Properties

An object property functionality axiom FunctionalObjectProperty(OPE) states that the object property expression OPE is functional — that is, for each individual x, there can be at most one distinct individual y such that x is connected by OPE to y. Each such axiom can be seen as a syntactic shortcut for the following axiom: SubClassOf(owl:Thing ObjectMaxCardinality(1 OPE)).

67.1 Simple Example

```
# OWL 2:
hasFather rdf:type owl:FunctionalProperty .
Stewie hasFather Peter .
Stewie hasFather Peter_Griffin .
```

Each object can have at most one father.
entailment:

```
Peter_Griffin owl:sameAs Peter .
```

(funct hasFather)

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	hasFather	-	-	functional	-

68 Inverse-Functional Properties

An object property inverse functionality axiom InverseFunctionalObjectProperty(OPE) states that the object property expression OPE is inverse-functional - that is, for each individual x, there can be at most one individual y such that y is connected by OPE with x. Each such axiom can be seen as a syntactic shortcut for the following axiom: SubClassOf(owl:Thing ObjectMaxCardinality(1 ObjectInverseOf(OPE))).

68.1 Simple Example

```
# OWL 2:
fatherOf rdf:type owl:InverseFunctionalProperty .
Peter fatherOf Stewie .
Peter_Griffin fatherOf Stewie .
```

Each object can have at most one father.
Entailment:

```
Peter owl:sameAs Peter_Griffin .
```

(funct hasFather$^-$)

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	hasFather$^-$	hasFather	-	inverse	-
property	$\top$	hasFather$^-$	-	-	functional	-

or alternatively:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	hasFather$^-$	hasFather	-	inverse	-
property	$\top$	hasFather$^-$	-	-	$\leqslant$	1

69 Value Restrictions

Individual Value Restrictions: A has-value class expression ObjectHasValue(OPE a) consists of an object property expression OPE and an individual a, and it contains all those individuals that are connected by OPE to a. Each such class expression can be seen as a syntactic shortcut for the class expression ObjectSomeValuesFrom(OPE ObjectOneOf(a)). Literal Value Restrictions: A has-value class expression DataHasValue(DPE lt) consists of a data property expression DPE and a literal lt, and it contains all those individuals that are connected by DPE to lt. Each such class expression can be seen as a syntactic shortcut for the class expression DataSomeValuesFrom(DPE DataOneOf(lt)).

69.1 Simple Example

$$FatherOfStewie \sqsubseteq \exists fatherOf.\{Stewie\}$$

```
# OWL 2:
Peter fatherOf Stewie .
ObjectHasValue( fatherOf Stewie )
```

The has-value class expression contains those individuals that are connected through the `fatherOf` property with the individual `Stewie`. `Peter` is classified as its instance.

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	FatherOfStewie	fatherOf	-	$\{Stewie\}$	$\exists$	

70 Self Restrictions

A self-restriction ObjectHasSelf(OPE) consists of an object property expression OPE, and it contains all those individuals that are connected by OPE to themselves.

70.1 Simple Example

```
# OWL 2:
Peter likes Peter .
ObjectHasSelf( likes )
```

$$\text{LikesThemselves} \sqsubseteq \exists \text{ likes.Self}$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\exists$ likes . Self	likes	-	Self	$\exists$	
class	LikesThemselves	-	-	$\top$, $\exists$ likes . Self	$\sqsubseteq$	

The self-restriction contains those individuals that like themselves. `Peter` is classified as its instance.

71 Data Property Facets

`Data Property Facets` corresponds to the requirement `R-46-CONSTRAINING-FACETS`. For datatype properties it should be possible to declare frequently needed "facets" to drive user interfaces and validate input against simple conditions, including min/max value, regular expressions, string length etc. similar to XSD datatypes. Constraining facets to restrict datatypes of RDF literals. Constraining facets may be: xsd:length, xsd:minLength, xsd:maxLength, xsd:pattern, xsd:enumeration, xsd:whiteSpace, xsd:maxInclusive, xsd:maxExclusive, xsd:minExclusive, xsd:minInclusive, xsd:totalDigits, xsd:fractionDigits.

71.1 Simple Example

string matches regular expression

```
# OWL 2 QL (functional-style syntax):
Declaration( Datatype( SSN ) )
DatatypeDefinition(
    SSN
    DatatypeRestriction( xsd:string xsd:pattern "[0-9]{3}-[0-9]{2}-[0-9]{4}" ) )
DataPropertyRange( hasSSN SSN )
```

```
# OWL 2 QL (turtle syntax):
SSN
    a rdfs:Datatype ;
    owl:equivalentClass [
        a rdfs:Datatype ;
        owl:onDatatype xsd:string ;
        owl:withRestrictions (
            [ xsd:pattern "[0-9]{3}-[0-9]{2}-[0-9]{4}" ] ) ] .
hasSSN rdfs:range SSN .
```

A social security number is a string that matches the given regular expression. The second axiom defines SSN as an abbreviation for a datatype restriction on xsd:string. The first axiom explicitly declares SSN to be a datatype. The datatype SSN can be used just like any other datatype; for example, it is used in the third axiom to define the range of the hasSSN property.

```
# valid data:
TimBernersLee
    hasSSN "123-45-6789"^^SSN .
```

```
# invalid data:
TimBernersLee
    hasSSN "123456789"^^SSN .
```

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	SSN	-	-	xsd:string	xsd:pattern	'[0-9]3-[0-9]2-[0-9]4'

72 Primary Key Properties

It is often useful to declare a given (datatype) property as the "primary key" of a class, so that the system can enforce uniqueness and also automatically build URIs from user input and data imported from relational databases or spreadsheets.

72.1 Simple Example

The `Primary Key Properties` constraint is often useful to declare a given (datatype) property as the "primary key" of a class, so that a system can enforce uniqueness. Starfleet officers, e.g., are uniquely identified by their command authorization code (e.g. to activate and cancel auto-destruct sequences). It means that the property `commandAuthorizationCode` is inverse functional - mapped to DL and the RDF-CO as follows:

$$(\texttt{funct}\ commandAuthorizationCode^{-})$$

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	commandAuthorizationCode$^-$	commandAuthorizationCode$^-$	-	inverse	-
property	$\top$	commandAuthorizationCode$^-$	-	-	functional	-

Keys, however, are even more general, i.e., a generalization of inverse functional properties [6]. A key can be a datatype property, an object property, or a chain of properties. For this generalization purposes, as there are different sorts of key, and as keys can lead to undecidability, DL is extended with `key boxes` and a special `keyfor` construct[5]. This leads to the following DL and RDF-CO mappings (only one simple property constraint):

commandAuthorizationCode `keyfor` StarfleetOfficer

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	StarFleetOfficer	commandAuthorizationCode	-	-	keyfor	-

73 Use Sub-Super Relations in Validation

`Exploiting Class/Property Specialization Ontology Axioms` corresponds to the requirement `R-224-instance-level-data-validation-exploitingclass/property-specialization-axioms-in-ontologies`. Validation of instances data (direct or indirect) exploiting the subclass or sub-property link in a given ontology. This validation can indicate when the data is verbose (redundant) or expressed at a too general level, and could be improved. Examples are:

- If dc:date and one of its sub-properties dcterms:created or dcterms:issued are present, check that the value in dc:date is not redundant with dcterms:created or dcterms:issued for ingestion
- Check if dc:rights has the same value than edm:rights either as rdf:resource or literal, if yes dc:rights is redundant
- If one or more dc:coverage are present, suggest the use of one of its sub-properties, dcterms spatial or dcterms:temporal.

73.1 Simple Example

If dc:date and one of its sub-properties dcterms:created or dcterms:issued are present, check that the value in dc:date is not redundant with dcterms:created or dcterms:issued for ingestion

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	dc:date	dcterms:created, dcterms:issued	-	not redundant	-

74 Cardinality Shortcuts

In most library applications, cardinality shortcuts tend to appear in pairs, with repeatable / non-repeatable establishing maximum cardinality and optional / mandatory establishing minimum cardinality.

- Optional & Non-Repeatable = [0,1]
- Optional & Repeatable = [0,*]
- Mandatory & Non-Repeatable = [1,1]
- Mandatory & Repatable = [1,*]

Can be expressed in DL using minimum and maximum cardinality restrictions. We propose to use syntactic sugar instead:

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	C	p	-	-	optional and non-repeatable	-

75 Aggregations

Some constraints require aggregating multiple values, especially via COUNT, MIN and MAX.

75.1 Simple Example

p = COUNT (q)

Context class is `C`.

Not expressible in DL

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
property	C	p	q	-	count	-

76 Provenance

Any provenance information must be available for instances of given classes.

76.1 Simple Example

For *publications*, any provenance information must be available.

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	Publication	-	-	-	provenance	-

77 Data Model Consistency

Is the data consistent with the intended semantics of the data model? Such validation rules ensure the integrity of the data according to the data model.

Not expressible in DL!

78 Structure

SKOS is based on RDF, which is a graph-based data model. Therefore we can concentrate on the vocabulary's graph-based structure for assessing the quality of SKOS vocabularies and apply graph- and network-analysis techniques.

Not expressible in DL!

79 Labeling and Documentation

Not expressible in DL!

80 Vocabulary

Vocabularies should not invent any new terms or use deprecated elements.

80.1 Simple Example

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	$\top$	-	-	-	vocabulary	-

81 HTTP URI Scheme Violation

81.1 Simple Example

Not expressible in DL!

Mapping to RDF-CO:

c. set	context class	left p. list	right p. list	classes	c. element	c. value
class	$\top$	-	-	-	HTTP URI Scheme Violation	-

82 Evaluation

82.1 Evaluation of Constraint Languages

We evaluated to which extend the most promising five constraint languages fulfill each requirement. Tilde means that this constraint may be fulfilled by that particular constraint language - either by limitations, workarounds, or extensions. We also evaluated if a specific constraint is fulfilled by OWL 2 QL or if the more expressive OWL 2 DL is needed. Inferencing may be performed prior to validating constraints. This is marked with an asterisk.

constraint	DSP	OWL2-DL	OWL2-QL	ReSh	ShEx	SPIN
*Subsumption	✗	✓	✓	~	✓	✓
*Class Equivalence	✗	✓	✓	✗	✗	✓
*Sub Properties	✗	✓	✓	✗	✗	✓
*Property Domains	✗	✓	✓	✗	✗	✓
*Property Ranges	✗	✓	✓	✗	✗	✓
*Inverse Object Properties	✗	✓	✓	~	✗	✓
*Symmetric Object Properties	✗	✓	✓	✗	✗	✓
*Asymmetric Object Properties	✗	✓	✓	✗	✗	✓
*Reflexive Object Properties	✗	✓	✓	✗	✗	✓
*Irreflexive Object Properties	✗	✓	✓	✗	✗	✓
Disjoint Properties	✗	✓	✓	✗	✗	✓
Disjoint Classes	✗	✓	✓	✗	✗	✓
Context-Sp. Property Groups	✗	~	~	✓	✓	✓
Context-Sp. Inclusive OR of P.	✗	~	~	✗	✗	✓
Context-Sp. Inclusive OR of P. Groups	✗	~	~	✗	✗	✓
Recursive Queries	✓	✓	✓	✓	✓	~
Individual Inequality	✗	✓	✓	✗	✗	✓
*Equivalent Properties	✗	✓	✓	✗	✗	✓
Property Assertions	✗	✓	~	✗	✗	✓
Data Property Facets	✗	✓	✓	✗	✗	✓
Literal Pattern Matching	✗	✓	✗	✗	✗	✓
Negative Literal Pattern Matching	✗	✓	✗	✗	✗	✓
*Object Property Paths	✗	✓	✗	✗	✗	✓
*Intersection	✗	✓	✗	✓	✓	✓
*Disjunction	✗	✓	✗	✗	✗	✓
*Negation	✗	✓	✗	✗	✗	✓
*Existential Quantifications	✗	✓	✗	~	~	✓
*Universal Quantifications	✗	✓	✗	✗	✗	✓
*Minimum Unqualified Cardinality	✓	✓	✗	~	✓	✓
*Minimum Qualified Cardinality	✓	✓	✗	~	✓	✓
*Maximum Unqualified Cardinality	✓	✓	✗	~	✓	✓
*Maximum Qualified Cardinality	✓	✓	✗	~	✓	✓
*Exact Unqualified Cardinality	✓	✓	✗	~	✓	✓
*Exact Qualified Cardinality	✓	✓	✗	~	✓	✓
*Transitive Object Properties	✗	✓	✗	✗	✗	✓
Context-Sp. Exclusive OR of P.	✗	✓	✗	✗	✓	✓
Context-Sp. Exclusive OR of P. Groups	✗	~	✗	✓	✓	✓
Allowed Values	✓	✓	✗	✓	✓	✓
Not Allowed Values	✗	✓	✗	✗	✓	✓
Literal Ranges	✗	✓	✗	✗	✗	✓
Negative Literal Ranges	✗	✓	✗	✗	✗	✓
Required Properties	✓	✓	✗	✓	✓	✓
Optional Properties	✓	✓	✗	✓	✓	✓
Repeatable Properties	✓	✓	✗	✓	✓	✓
Negative Property Constraints	✗	✓	✗	✗	✓	✓
*Individual Equality	✗	✓	✗	✗	✗	✓
*Functional Properties	✗	✓	✗	✗	✗	✓
*Inverse-Functional Properties	✗	✓	✗	✗	✗	✓
*Value Restrictions	✓	✓	✗	✓	✓	✓
*Self Restrictions	✗	✓	✗	✗	✗	✓
*Primary Key Properties	✗	✓	✗	✗	✗	✓
*Class-Specific Property Range	✓	✓	✗	✓	✓	✓
*Class-Sp. Reflexive Object P.	✗	✓	✗	✗	✗	✓
Membership in Controlled Vocabularies	✓	✗	✗	✗	✗	✓
IRI Pattern Matching	✗	✗	✗	✗	✓	✓
Literal Value Comparison	✗	✗	✗	✗	✓	✓
Ordering	✗	✗	✗	✗	✗	✓
Validation Levels	✗	✗	✗	✗	✗	✓
String Operations	✗	✗	✗	✗	✗	✓

constraint	DSP	OWL2-DL	OWL2-QL	ReSh	ShEx	SPIN
Context-Specific Valid Classes	✗	✗	✗	✗	✗	✓
Context-Specific Valid Properties	✗	✗	✗	✗	✗	✓
*Default Values	✗	✗	✗	✓	✗	✓
Mathematical Operations	✗	✗	✗	✗	✗	✓
Language Tag Matching	✗	✗	✗	✗	✗	✓
Language Tag Cardinality	✗	✗	✗	✗	✗	✓
Whitespace Handling	✗	✗	✗	✗	✗	✓
HTML Handling	✗	✗	✗	✗	✗	✓
Conditional Properties	✗	✗	✗	✗	✗	✓
Recommended Properties	✗	✗	✗	✗	✗	✓
Handle RDF Collections	✗	✗	✗	✗	✗	✓
Value is Valid for Datatype	✗	✗	✗	✗	✗	✓
Use Sub-Super Relations in Validation	✗	✗	✗	✗	✗	✓
*Cardinality Shortcuts	✗	✓	✗	✓	✓	✓
Aggregations	✗	✗	✗	✗	✗	✓
*Class-Specific Irreflexive Object Properties	✗	✓	✗	✗	✗	✓
Provenance	✗	✗	✗	✗	✗	✓
Data Model Consistency	✗	✗	✗	✗	✗	✓
Structure	✗	✗	✗	✗	✗	✓
Labeling and Documentation	✗	✗	✗	✗	✗	✓
Vocabulary	✗	✗	✗	✗	✗	✓
HTTP URI Scheme Violation	✗	✗	✗	✗	✗	✓

82.2 Evaluation of Constraints Classes

We evaluated to which extend the most promising five constraint languages fulfill each of the overall 74 requirements to formulate RDF constraints [2]. If a constraint can be expressed in DL, we added the mapping to DL and to the generic constraint. If a constraint cannot be expressed in DL, we only added the mapping to the generic constraint. Therefore, we show that each constraint can be mapped to a generic constraint. The following table shows the absolute numbers and the relative percentages for each of the three dimensions to classify constraints:

Constraint Classes	#	%
Property Constraints	49	60 (60.49)
Class Constraints	20	25 (24.96)
Property and Class Constraints	12	15 (14.81)
Simple Constraints	49	60 (60.49)
Simple Constraints (Syntactic Sugar)	11	14 (13.58)
Complex Constraints	21	26 (25.93)
DL Expressible	52	64 (64.2)
DL Not Expressible	29	36 (35.80)
Total	81	100

legend of the detailed evaluation:

- `property constraint:` property constraint (✓) vs. class constraint (✗) vs. property and class constraints (∼)
- `simple constraint:` simple constraint (✓) vs. simple constraint [syntactic sugar] (∼) vs. complex constraint (✗)

– `DL:` expressible in DL (✓) vs. not expressible in DL (✗)

constraint	property c.	simple c.	DL
*Subsumption	✗	✓	✓
*Class Equivalence	✗	✗	✓
*Sub Properties	✓	✓	✓
*Property Domains	✓	~	✓
*Property Ranges	✓	~	✓
*Inverse Object Properties	✓	✓	✓
*Symmetric Object Properties	✓	✓	✓
*Asymmetric Object Properties	✓	~	✓
*Reflexive Object Properties	✓	~	✓
*Irreflexive Object Properties	✓	~	✓
Disjoint Properties	✓	✓	✓
Disjoint Classes	✗	✗	✓
Context-Sp. Property Groups	~	✗	✓
Context-Sp. Inclusive OR of P.	~	✗	✓
Context-Sp. Inclusive OR of P. Groups	~	✗	✓
Recursive Queries	✓	✓	✓
Individual Inequality	✗	✗	✓
*Equivalent Properties	✓	✗	✓
Property Assertions	✓	✓	✓
Data Property Facets	✗	✓	✗
Literal Pattern Matching	✗	✓	✗
Negative Literal Pattern Matching	✗	✗	✗
*Object Property Paths	✓	✓	✓
*Intersection	✗	✓	✓
*Disjunction	✗	✓	✓
*Negation	✗	✓	✓
*Existential Quantifications	✓	✓	✓
*Universal Quantifications	✓	✓	✓
*Minimum Unqualified Cardinality	✓	✓	✓
*Minimum Qualified Cardinality	✓	✓	✓
*Maximum Unqualified Cardinality	✓	✓	✓
*Maximum Qualified Cardinality	✓	✓	✓
*Exact Unqualified Cardinality	✓	~	✓
*Exact Qualified Cardinality	✓	~	✓
*Transitive Object Properties	✓	✓	✓
Context-Sp. Exclusive OR of P.	~	✗	✓
Context-Sp. Exclusive OR of P. Groups	~	✗	✓
Allowed Values	✗	✓	✓
Not Allowed Values	✗	✗	✓
Literal Ranges	✗	✗	✗
Negative Literal Ranges	✗	✗	✗
Required Properties	✓	✓	✓
Optional Properties	✓	~	✓
Repeatable Properties	✓	✓	✓
Negative Property Constraints	~	✗	✓
*Individual Equality	✗	✗	✓
*Functional Properties	✓	✓	✓
*Inverse-Functional Properties	✓	✗	✓
*Value Restrictions	✓	✓	✓
*Self Restrictions	~	✗	✓
*Primary Key Properties	✓	~	✓
*Class-Specific Property Range	✓	~	✓
*Class-Sp. Reflexive Object P.	✓	✓	✓
Membership in Controlled Vocabularies	~	✗	✓
IRI Pattern Matching	✗	✓	✗
Literal Value Comparison	✓	✓	✗
Ordering	✓	✓	✗
Validation Levels	~	✓	✗
String Operations	✓	✓	✗

constraint	property c.	simple c.	DL
Context-Specific Valid Classes	✗	✓	✗
Context-Specific Valid Properties	✓	✓	✗
*Default Values	✓	✓	✗
Mathematical Operations	✓	✓	✗
Language Tag Matching	✓	✓	✗
Language Tag Cardinality	✓	✓	✗
Whitespace Handling	✓	✓	✗
HTML Handling	✓	✓	✗
Conditional Properties	✓	✓	✓
Recommended Properties	✓	✓	✗
Handle RDF Collections	✓	✓	✗
Value is Valid for Datatype	✓	✓	✗
Use Sub-Super Relations in Validation	✓	✓	✗
*Cardinality Shortcuts	✓	✓	✓
Aggregations	✓	✓	✗
*Class-Specific Irreflexive Object Properties	✓	~	✓
Provenance	✗	✓	✗
Data Model Consistency	~	✗	✗
Structure	~	✗	✗
Labeling and Documentation	~	✗	✗
Vocabulary	✗	✓	✗
HTTP URI Scheme Violation	✗	✓	✗

Constraints can be classified as `property constraints` and `class constraints`. Two thirds of the total amount of constraints are property constraints, one fifth are class constraints, and approx. 10% are composed of both property and class constraints. Constraints may be either atomic (`simple constraints`) or created out of simple and/or complex constraints (`complex constraints`). Almost two thirds are simple constraints, a quarter are complex constraints. Almost 15 percent are complex constraints which can be formulated as simple constraints when using them in terms of syntactic sugar. Constraints can either be expressible in DL or not. The majority - nearly 70% - of the overall constraints are expressible in DL.

82.3 CWA and UNA Dependency

- Dependent on the *CWA*: do further triples change the validation result?
- Dependent on the *UNA*: Do validation results changes in case two resources are the same?

	Dependency	
Constraint Types	*CWA*	*UNA*
*Subsumption	✓	✓
*Class Equivalence	✓	✓
*Sub Properties	✓	✓
*Property Domains	✓	✓
*Property Ranges	✓	✓
*Inverse Object Properties	✓	✓
*Symmetric Object Properties	✓	✓
*Asymmetric Object Properties	✗	✗
*Reflexive Object Properties	✓	✓
*Irreflexive Object Properties	✗	✗
Disjoint Properties	✗	✗
Disjoint Classes	✗	✓
Context-Sp. Property Groups	✓	✓
Context-Sp. Inclusive OR of P.	✓	✓
Context-Sp. Inclusive OR of P. Groups	✓	✓
Recursive Queries	✗	✗
Individual Inequality	✗	✗
*Equivalent Properties	✓	✓
Property Assertions	✓	✓
Data Property Facets	✗	✗
Literal Pattern Matching	✗	✗
Negative Literal Pattern Matching	✗	✗
*Object Property Paths	✓	✓
*Intersection	✓	✓
*Disjunction	✓	✓
*Negation	✗	✗
*Existential Quantifications	✓	✓
*Universal Quantifications	✗	✓
*Minimum Unqualified Cardinality	✓	✓
*Minimum Qualified Cardinality	✓	✓
*Maximum Unqualified Cardinality	✗	✓
*Maximum Qualified Cardinality	✗	✓
*Exact Unqualified Cardinality	✓	✓
*Exact Qualified Cardinality	✓	✓
*Transitive Object Properties	✓	✓
Context-Sp. Exclusive OR of P.	✗	✓
Context-Sp. Exclusive OR of P. Groups	✗	✓
Allowed Values	✗	✓
Not Allowed Values	✗	✓
Literal Ranges	✗	✗
Negative Literal Ranges	✗	✗
Required Properties	✓	✓
Optional Properties	✗	✗
Repeatable Properties	✓	✗
Negative Property Constraints	✗	✓
*Individual Equality	✓	✗
*Functional Properties	✓	✓
*Inverse-Functional Properties	✓	✓
*Value Restrictions	✓	✓
*Self Restrictions	✓	✓

	Dependency	
Constraint Types	*CWA*	*UNA*
*Primary Key Properties	✓	✓
*Class-Specific Property Range	✓	✓
*Class-Sp. Reflexive Object P.	✓	✓
Membership in Controlled Vocabularies	✓	✓
IRI Pattern Matching	✗	✓
Literal Value Comparison	✗	✗
Ordering	✓	✗
Validation Levels	✓	✗
String Operations	✗	✗
Context-Specific Valid Classes	✗	✗
Context-Specific Valid Properties	✗	✗
*Default Values	✓	✓
Mathematical Operations	✗	✗
Language Tag Matching	✓	✗
Language Tag Cardinality	✓	✓
Whitespace Handling	✗	✗
HTML Handling	✗	✗
Conditional Properties	✓	✓
Recommended Properties	✓	✓
Handle RDF Collections	✗	✗
Value is Valid for Datatype	✗	✗
Use Sub-Super Relations in Validation	✗	✓
*Cardinality Shortcuts	✓	✓
Aggregations	✗	✗
*Class-Specific Irreflexive Object Properties	✗	✓
Provenance	✓	✓
Data Model Consistency	✓	✓
Structure	✓	✓
Labeling and Documentation	✓	✓
Vocabulary	✓	✓
HTTP URI Scheme Violation	✗	✓

82.4 Constraining Elements of Constraint Types

- Constraints of 64% of all constraint type can be expressed using logical constructs from description logics.
- For a constraint types, there can be multiple constraining elements
- The validation of each constraining element has to be implemented
- We use SPARQL as extension mechanism to express constraints of 3 constraint types which cannot be expressed otherwise as they are too complex to represent them using a high-level constraint language

Constraint Type	**DL**	**Constraining Elements**
*Subsumption	$\sqsubseteq$	sub-class
*Class Equivalence	$\equiv$	equivalent class
*Sub Properties	$\sqsubseteq$	sub-property
*Property Domains	$\exists$, $\sqsubseteq$	property domain
*Property Ranges	$\sqsubseteq$, $\forall$	property range
*Inverse Object Properties	inverse	inverse property
*Symmetric Object Properties	symmetric	symmetric property
*Asymmetric Object Properties	asymmetric	asymmetric property
*Reflexive Object Properties	reflexive	reflexive property
*Irreflexive Object Properties	reflexive, $\neg$	irreflexive property
Disjoint Properties	$\sqsubseteq$, $\neg$	disjoint property
Disjoint Classes	$\sqcap$, $\sqsubseteq$	disjoint class
Context-Sp. Property Groups	$\sqcap$	property group
Context-Sp. Inclusive OR of P.	$\sqcup$	inclusive or
Context-Sp. Inclusive OR of P. Groups	$\sqcup$, $\sqcap$	inclusive or
Recursive Queries	-	recursive
Individual Inequality	$\neq$	different individuals
*Equivalent Properties	$\equiv$	equivalent properties
Property Assertions	$=$ or $\neq$	$=$, $\neq$
Data Property Facets	-	SPARQL functions: REGEX, STRLEN XML Schema constraining facets: xsd:length, xsd:minLength, xsd:maxLength, xsd:pattern, xsd:enumeration, xsd:whiteSpace, xsd:maxInclusive, xsd:maxExclusive,xsd:minExclusive, xsd:minInclusive, xsd:totalDigits, xsd:fractionDigits
Literal Pattern Matching	-	REGEX, xsd:pattern
Negative Literal Pattern Matching	-	REGEX, xsd:pattern and $\neg$
*Object Property Paths	$\sqsubseteq$	property path
*Intersection	$\sqcap$	intersection
*Disjunction	$\sqcup$	disjunction
*Negation	$\neg$	negation
*Existential Quantification	$\exists$	existential quantification
*Universal Quantification	$\forall$	universal quantification
*Minimum Unqualified Cardinality	$\geqslant$	minimum unqualified cardinality restriction
*Minimum Qualified Cardinality	$\geqslant$	minimum qualified cardinality restriction
*Maximum Unqualified Cardinality	$\leqslant$	maximum unqualified cardinality restriction
*Maximum Qualified Cardinality	$\leqslant$	maximum qualified cardinality restriction
*Exact Unqualified Cardinality	($\geqslant$, $\leqslant$, $\sqcap$) or $=$	exact unqualified cardinality restriction
*Exact Qualified Cardinality	($\geqslant$, $\leqslant$, $\sqcap$) or $=$	exact qualified cardinality restriction
*Transitive Object Properties	$\sqsubseteq$	transitive property
Context-Sp. Exclusive OR of P.	$\neg$, $\sqcap$, $\sqcup$	exclusive or
Context-Sp. Exclusive OR of P. Groups	$\neg$, $\sqcap$, $\sqcup$	exclusive or
Allowed Values	$\sqcup$	allowed values
Not Allowed Values	$\sqcup$, $\neg$	not allowed values
Literal Ranges	-	xsd:minInclusive, xsd:maxExclusive xsd:maxInclusive, xsd:minExclusive
Negative Literal Ranges	-	$\neg$ and xsd:minInclusive, xsd:maxExclusive xsd:maxInclusive, xsd:minExclusive
Required Properties	$\exists$	required property

Constraint Type	**DL**	**Constraining Elements**
Optional Properties	$\exists$, $\sqsubseteq$	optional property
Repeatable Properties	$\geqslant$	repeatable property
Negative Property Constraints	$\neg$, $\exists$	negative properties
*Individual Equality	$=$	equal individuals
*Functional Properties	functional	functional property
*Inverse-Functional Properties	inverse, functional	inverse functional property
*Value Restrictions	$\exists$	value restriction
*Self Restrictions	$\exists$	self restriction
Primary Key Properties	inverse, functional	primary key
*Class-Specific Property Range	$\sqsubseteq$, $\exists$, $\neg$	property range
*Class-Sp. Reflexive Object P.	reflexive	reflexive property
Membership in Controlled Vocabularies	$\forall$, $\sqcap$, $\sqcup$	membership in controlled vocabularies
IRI Pattern Matching	-	IRI pattern matching
Literal Value Comparison	$>$ or $\geqslant$ or $<$ or $\leqslant$ or $=$ or $\neq$	$>$, $\geqslant$, $<$, $\leqslant$, $=$, $\neq$
Ordering	-	ordered properties, ordered values
Validation Levels	-	validation level
String Operations	-	SPARQL string functions: STRLEN, SUBSTR, UCASE, LCASE, STRSTARTS, STRENDS, CONTAINS, STRBEFORE, STRAFTER, ENCODE_FOR_URI, CONCAT, langMatches, REGEX, REPLACE
Context-Specific Valid Classes	-	context-specific valid classes
Context-Specific Valid Properties	-	context-specific valid properties
Default Values	-	default values
Mathematical Operations	-	addition, subtraction, multiplication, division
Language Tag Matching	-	language tag matching
Language Tag Cardinality	-	language tag minimum cardinality, language tag maximum cardinality, language tag exact cardinality
Whitespace Handling	-	no whitespaces
HTML Handling	-	no html
Conditional Properties	$\sqsubseteq$	conditional property
Recommended Properties	-	recommended property
Handle RDF Collections	-	actions on RDF collections: append list element
Value is Valid for Datatype	-	value is valid for datatype
Use Sub-Super Relations in Validation	-	non redundant properties
*Cardinality Shortcuts		optional & non-repeatable property, optional & repeatable property, mandatory & non-repeatable property, mandatory & repeatable property
Aggregations	-	aggregations: count
*Class-Specific Irreflexive Object Properties	reflexive, $\neg$	irreflexive property
Provenance	-	provenance information required
Data Model Consistency	-	<SPARQL query>
Structure	-	<SPARQL query>
Labeling and Documentation	-	<SPARQL query>
Vocabulary	-	vocabulary
HTTP URI Scheme Violation	-	HTTP URI scheme violation

83 Conclusion and Future Work

There is no standard way to validate RDF data conforming to RDF constraints like XML Schemas serve to validate XML documents. Two working groups currently try to achieve a solution for RDF validation - the W3C RDF Data Shapes working group and the DCMI RDF Application Profiles working group. We initiated a comprehensive database on RDF validation requirements: http://purl.org/net/rdf-validation. The intention of this database is to collaboratively work on case studies, use cases, requirements, and solutions on RDF validation. In this paper, we evaluated to which extend the five most promising constraint languages on being the standard (DSP, OWL2, ReSh, ShEx, and SPIN) fulfill each of the requirements to formulate RDF constraints. Each of these requirements corresponds to a type of constraint. The majority of the constraints can be expressed in DL, which serves as a logical underpinning of related requirements. We developed an ontology to express any constraint generically, so that constrains expressed by a constraint language α can be transformed into constraints expressed by a constraint language β without any information loss. By expressing any constraint generically, we can provide a validation of the generically expressed constraint. When specific constraints are then transformed into generic constraints, we can provide the validation of the semantically equivalent specific constraints (expressed by multiple constraint languages) out-of-the-box without any additional effort and without any difference in validation results. As not every constrains can be represented in DL, we need to represent *constraints expressible by DL* as well as *constraints not expressible by DL* by means of this ontology. We shown in terms of an evaluation that any constraint can be expressed using the developed ontology. As part of future work, we will continuously add, modify, and maintain case studies, use cases, requirements, and solutions within the RDF validation requirements database.

References

1. Thomas Bosch and Kai Eckert. Requirements on RDF Constraint Formulation and Validation. In *Proceedings of the DCMI International Conference on Dublin Core and Metadata Applications (DC 2014)*, Austin, Texas, USA, 2014. http://dcevents.dublincore.org/IntConf/dc-2014/paper/view/257.
2. Thomas Bosch, Andreas Nolle, Erman Acar, and Kai Eckert. RDF Validation Requirements - Evaluation and Logical Underpinning. *Computing Research Repository (CoRR)*, abs/1501.03933, 2015. http://arxiv.org/abs/1501.03933.
3. Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen, and Amrapali Zaveri. Test-driven Evaluation of Linked Data Quality. In *Proceedings of the 23rd International World Wide Web Conference (WWW 2014)*, WWW '14, pages 747–758, Republic and Canton of Geneva, Switzerland, 2014. International World Wide Web Conferences Steering Committee.
4. Markus Krötzsch, František Simančík, and Ian Horrocks. A Description Logic Primer. In Jens Lehmann and Johanna Völker, editors, *Perspectives on Ontology Learning*. IOS Press, 2012.

5. Carsten Lutz, Carlos Areces, Ian Horrocks, and Ulrike Sattler. Keys, Nominals, and Concrete Domains. *Journal of Artificial Intelligence Research*, 23(1):667–726, June 2005. http://dl.acm.org/citation.cfm?id=1622503.1622518.
6. Michael Schneider. OWL 2 Web Ontology Language RDF-Based Semantics. W3C recommendation, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-rdf-based-semantics-20091027/.