Formulating RDF Constraints and Validating RDF Data Generically

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

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

Abstract. More and more domains adopt Linked Data principles and seek for ways to ensure data quality by validating RDF data according to predefined constraints, particularly as many communities are used to have this functionality in the XML world. XML Schema is the standard constraint language to formulate constraints on and to validate XML documents. For RDF, however, there are several candidates for languages to express constraints on RDF data - but no standard one meeting the majority of RDF validation use cases. We identified requirements to formulate constraints and to validate RDF data and mapped each requirement to an RDF constraint type. The majority of these constraint types can be represented in description logics providing their logical underpinning. The **contributions** of this paper are: (1) we provide a basic terminology and classification system for RDF constraints, (2) we developed a vocabulary to describe constraints of any constraint type generically, (3) we show how to transform constraints, expressed by any constraint language α, into generically expressed constraints and into constraints represented by any other constraint language β and discuss why these transformations are useful, and (4) we explain how to overcome the necessity to implement the validation of each constraint type for multiple constraint languages.

Keywords: RDF Validation, RDF Constraints, RDF Validation Requirements, Linked Data, Semantic Web

1 Introduction

In 2013, the W3C organized the RDF Validation Workshop[3], where experts from industry, government, and academia discussed first use cases for RDF constraint[4] formulation and RDF data validation. In 2014, two working groups on

Thomas: feedback from Erman: the major problem is semantics. OWL basically work in different semantics than others. And their semantics are formal. Closed world vs. Open World is also a part of this.

[3] http://www.w3.org/2012/12/rdf-val/

[4] For simplicity reasons, we use the terms *constraint types/constraints* instead of *RDF constraint types/RDF constraints* in the rest of the paper

RDF validation have been established to develop a language to express constraints on RDF data: the W3C RDF Data Shapes working group[5] and the DCMI RDF Application Profiles task group[6]. Bosch and Eckert [1] collected the findings of these working groups and initiated a database of RDF validation requirements which is available for contribution at http://purl.org/net/rdf-validation. The intention is to collaboratively collect case studies, use cases, requirements, and solutions regarding RDF validation in a comprehensive and structured way. Bosch et al. identified 74 requirements to formulate constraints; each of them corresponding to a constraint type which may be expressed by at least one existing constraint language [3].

Recently, RDF validation as a research field gained speed due to common needs of data practitioners. A typical example is the library domain that co-developed and adopted Linked Data principles very early. For libraries, the common description of resources are key business and they have a long tradition in developing and using interoperable data formats. While they embrace the openness of Linked Data and the data modeling principles provided by RDF, the data is still mostly represented in XML and this is unlikely to change soon. Among the reasons for the success of XML is the possibility to formulate fine-grained constraints to be met by the data and to validate the data according to these constraints using powerful systems like DTDs, XML Schemas, RELAX NG, or Schematron. A typical example is the definition of a library record describing a book. There are clear rules which information has to be available to describe a book properly (required fields, like a title), but also how information like an ISBN number is properly represented. Libraries seek to make their own data reusable for general purposes, but also to enrich and interlink their own data. Checking if third-party data meets own requirements or validating existing data according to new needs for a Linked Data application are among common use cases for RDF validation.

2 Motivation

There is not the one language meeting the majority of requirements to formulate constraints. As there are more than 10 candidate languages with different syntaxes and semantics which can be used to express constraints, the language selection heavily depends on the individual use case. The five most promising constraint languages on being the standard are *Description Set Profiles (DSP)*, *Resource Shapes (ReSh)*, *Shape Expressions (ShEx)*, the *SPARQL Inferencing Notation (SPIN)*, and the *Web Ontology Language (OWL 2)*. The idea behind this paper is to achieve complete RDF validation requirements coverage by representing any constraint type (corresponding to a requirement to formulate constraints) in a generic way using a lightweight vocabulary consisting of only a few terms (see section 3).

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

Cardinality restrictions, e.g., can be expressed either generically (***generic constraints***) by *description logics (DL)* or specifically (***specific constraints***) by domain-specific constraint languages such as *OWL 2*, *DSP*, *ShEx*, *ReSh*, and *SPIN*. The knowledge representation formalism *DL*, with its well-studied theoretical properties, provides the foundational basis to express constraints in a generic way. For that reason, we map constraint types to *DL* statements. With *minimum qualified cardinality restrictions* (*R-75, R-81*)[7], researchers from the library domain may restrict that *publications* must have at least one *author* which must be a *person*. This constraint can be represented generically (*generic constraint*) using *DL* (`Publication` $\sqsubseteq \geqslant 1$ `author.Person`), but also specifically (*specific constraint*) by multiple constraint languages:

```
# OWL 2:
Publication
    a owl:Restriction ;
    owl:minQualifiedCardinality 1 ;
    owl:onProperty author ;
    owl:onClass Person .

# ShEx:
Publication { author @Person{1, } }

# ReSh:
Publication a rs:ResourceShape ; rs:property [
    rs:name "author" ; rs:propertyDefinition author ;
    rs:valueShape Person ;
    rs:occurs rs:One-or-many ; ] .

# DSP:
[   a dsp:DescriptionTemplate ;
    dsp:resourceClass Publication ;
    dsp:statementTemplate [ a dsp:NonLiteralStatementTemplate ;
        dsp:minOccur 1 ; dsp:maxOccur "infinity" ;
        dsp:property author ;
        dsp:nonLiteralConstraint [ a dsp:NonLiteralConstraint ;
            dsp:valueClass Person ] ] ] .
```

As each *publication* must have at least one *author* which must be a *person* and as the book *The-Lord-Of-The-Rings* is a *publication* (`rdf:type(The-Lord-Of-The-Rings, Publication)`), *The-Lord-Of-The-Rings* must have at least one *author* relationship to a *person*. Constraint violations are raised either if *The-Lord-Of-The-Rings* does not have any *author* relationship, or if *The-Lord-Of-The-Rings* has an *author* which is not a *person* (`author(The-Lord-Of-The-Rings, Tolkien)`, `rdf:type(Tolkien, Hobbit)`), or if *The-Lord-Of-The-Rings* has an *author* for which no class is assigned (`author(The-Lord-Of-The-Rings, Tolkien)`). The data is valid, however, if *The-Lord-Of-The-Rings* is connected to a *person* via the property *author* (`author(The-Lord-Of-The-Rings, Tolkien)`, `rdf:type(Tolkien, Person)`).

As there is no standard way to formulate constraints, semantically equivalent *cardinality restrictions* may be represented by multiple constraint languages with different syntax and semantics. This causes confusion and weakens the common

[7] Requirements/constraint types are uniquely identified by alphanumeric technical identifiers like *R-75-MINIMUM-QUALIFIED-CARDINALITY-ON-PROPERTIES*

understanding between several parties about the semantics of particular constraints and therefore how to ensure high data quality. Thus, when choosing constraint language α to express a constraint of an arbitrary constraint type, it should be possible to transform this constraint into a semantically equivalent constraint formulated by any other constraint language β (see section 4). This is important in order to enhance the interoperability of constraint languages and to resolve ambiguities in the communication of RDF data producers and consumers, as constraint transformations avoid the necessity to understand several constraint languages.

Thomas: Erman thinks formal generic semantics should be defined

We provide a validation environment (available at http://purl.org/net/rdfval-demo) which enables to validate RDF data according to constraints expressed by diverse constraint languages. The next step, however, is to extend the ***RDF Validator*** by validating constraints of any constraint type represented by any constraint language and to validate semantically equivalent *specific constraints*, expressed by different constraint languages, in exactly the same way, i.e., validation is independent from the used constraint language (see section 5). We are able to offer an already implemented validation mechanism for any *specific constraint* out-of-the-box, (1) if we implement the validation of the corresponding *generic constraint type* once and (2) if we transform semantically equivalent *specific constraints* into generically expressed constraints.

This paper aims to address two main **audiences**: (1) RDF data providers and consumers seeking how to ensure high quality (meta)data and thus for ways to enhance the interoperability and common understanding of constraints expressible by multiple constraint languages and (2) RDF practitioners thinking of how to provide aligned implementations for the validation of both existing an newly developed constraint languages. The main **contributions** of this paper are: (1) we provide a basic terminology and classification system for RDF constraints (section 3), (2) we developed a vocabulary to describe constraints of any RDF constraint type generically (section 3), (3) we show how to transform constraints, expressed by any constraint language α, into generically expressed constraints and into constraints represented by any other constraint language β and discuss why these transformations are useful (section), and (4) we explain how to overcome the necessity to implement the validation of each constraint type for multiple constraint languages (section 5).

3 A Vocabulary to Describe RDF Constraints Generically

In order to develop a vocabulary to describe constraints of any constraint type generically, it is needed to define the basic terminology for the formulation of constraints and to classify them. A ***constraint language*** is a language which is used to formulate constraints. The W3C Data Shapes working group defines ***RDF constraint*** as a component of a schema what needs to be satisfied[8]. We identified four dimensions to classify constraint types:

[8] https://www.w3.org/2014/data-shapes/wiki/Glossary

- ***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 lightweight vocabulary (only three classes, three object properties, and three data properties) which is universal enough to describe constraints of any constraint type expressible by any constraint language (see the conceptual model in figure 1). We call this vocabulary the ***RDF Constraints Vocabulary (RDF-CV)***[9].

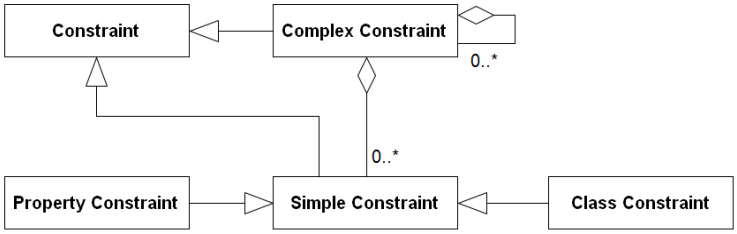

Fig. 1. *RDF Constraints Vocabulary (RDF-CV)* - conceptual model

Constraints can be expressed either generically (***generic constraints***) by the *RDF-CV* or specifically (***specific constraints***) by domain-specific constraint languages. As the *RDF-CV* describes constraints generically, the vocabulary does not distinguish constraints according to the dimension *universality*. The majority of the constraint types can be expressed in *DL*. In contrast, there are constraint types which cannot be expressed in *DL*, but are also representable by the *RDF-CV* (see section 6). ***Complex constraints*** encompass ***simple constraints*** (atomic constraints) and/or further *complex constraints*, i.e. *DL* statements representing *complex constraints* are created out of *DL* statements standing for atomic and composed constraints (if expressible in *DL*). Simple constraints may be applied to either properties (***property constraints***) or classes (***class constraints***). The *RDF-CV* does not contain any terms standing for *simple* and *complex constraints*, since *context classes* of *simple constraints* are just reused within *complex constraints* (*simple constraints* associated with *context classes* must hold for individuals of these *context classes*). As a consequence, the distinction of *property* and *class constraints* is sufficient to describe constraints of all possible constraint types.

3.1 *Simple Constraints* (Expressible in *DL*)

Sub-classes of ***simple constraints*** are ***property constraints*** and ***class constraints*** (see the *RDF-CV* implementation model in figure 2). For both *property*

[9] Available at: https://github.com/boschthomas/RDF-Constraints-Vocabulary

and *class constraints* a *context class*, a list of *classes*, the *constraining element*, and the *constraining value* can be stated. Lists of *left* and *right properties* can only be specified for *property constraints*.

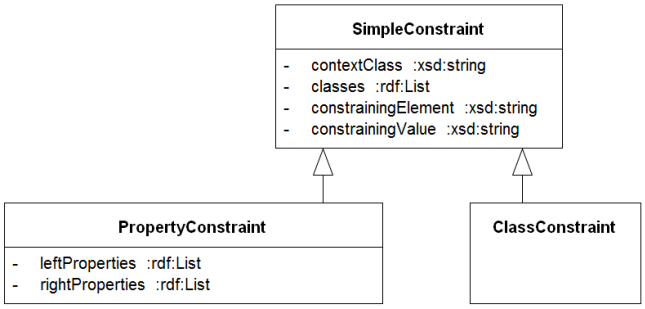

Fig. 2. *RDF Constraints Vocabulary (RDF-CV)* - implementation model

A *simple constraint* holds for all individuals of a ***context class***. Consider, e.g., the following *minimum qualified cardinality restriction* on the object property *author*, which restricts *publications* to have at least one *author* which is a *person*:

$$Publication \sqsubseteq \geqslant 1 author.Person$$

The cardinality restriction is a constraint on the property *author* and is therefore classified as a *property constraint* which holds for all individuals of the class *Publication* (see table 1 for the mapping to *RDF-CV*):

c. type	context class	left p. list	right p. list	classes	c. element	c. value
property	Publication	author	-	Person	$\geqslant$	1

Table 1. Property Constraint - Cardinality Restriction

The ***constraining element*** indicates the actual type of constraint like *DL* concept and role constructors, (in)equality, and further keywords for constraint types which cannot be expressed in *DL* (e.g. regular expressions and constraining facets). In some cases, a constraint is only complete when in addition to the *constraining element* a ***constraining value*** is stated. The cardinality restriction `⩾1 author.Person`, e.g., constructs an anonymous class of all individuals having at least one *author* relationship to *persons*. The *constraining element* of this *property constraint* is the *DL* at-least restriction $\geqslant$ and the *constraining value* is *1*. *Simple constraints* may refer to a list of ***classes***. The qualified cardinality restriction above, e.g., refers to the class *Person*, i.e. restricts the objects of the property *author* to be of the class *Person*.

For *property constraints*, ***left*** and ***right property lists*** are specified. The assignment of properties to these lists happens relative to the *constraining element* - which may be an operator (e.g., $\sqsubseteq$ in case of *object property paths*). *Object Property Paths* (*R-55*; also called *complex role inclusion axioms* in *DL*) express that, if an individual x is connected by a sequence of object properties

with an individual y, then x is also related to y by a particular object property. As *Stephen-Hawking* is the *author* of the book *A-Brief-History-Of-Time* (`authorOf(Stephen-Hawking, A-Brief-History-Of-Time)`) whose *genre* is *Popular-Science* (`genre(A-Brief-History-Of-Time, Popular-Science)`), the following *object property path* infers that *Stephen-Hawking* is an *author* of the *genre Popular-Science* (`authorOfGenre(Stephen-Hawking, Popular-Science)`):

$$\text{authorOf} \circ \text{genre} \sqsubseteq \text{authorOfGenre}$$

Thus, when mapped to the *RDF-CV* (see table 2), the properties *authorOf* and *genre* are on the left side of the *constraining element* $\sqsubseteq$, and the property *authorOfGenre* is on the right side:

c. type	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	authorOf, genre	authorOfGenre	$\top$	$\sqsubseteq$	-

Table 2. Property Constraint - Object Property Paths

3.2 *Simple Constraints* (Not Expressible in *DL*)

There are *simple constraints* which cannot be expressed in *DL* such as *literal pattern matching*, *literal value comparison*, *literal ranges*, *default values*, and *language tag cardinality* [3].

Literal pattern matching (*R-44*) restricts literals to match given patterns. The following *universal restriction*, e.g., ensures that *books* can only have valid *ISBN* identifiers, i.e., strings that match a given regular expression:

$$\texttt{Book} \sqsubseteq \forall\ \texttt{identifier.ISBN}$$

Even though, the restriction of the datatype *ISBN* cannot be expressed in *DL*, *OWL 2 DL* can be used to express the *literal pattern matching* constraint:

```
1 ISBN a RDFS:Datatype ; owl:equivalentClass [ a RDFS:Datatype ;
2     owl:onDatatype xsd:string ;
3     owl:withRestrictions ([ xsd:pattern "^\d{9}[\d|X]$" ])] .
```

The first OWL 2 axiom explicitly declares *ISBN* to be a datatype. The second OWL 2 axiom defines *ISBN* as an abbreviation for a datatype restriction on *xsd:string*. The datatype *ISBN* can be used just like any other datatype like in the *universal restriction* above. The *literal pattern matching* constraint validates *ISBN* literals according to the regular expression causing constraint violations for triples which do not not match. In table 3, the *simple constraint* (*literal pattern matching*) and the *complex constraint* (*universal restriction*) are mapped to the *RDF-CV*:

The *literal pattern matching* constraint type introduces the new *constraining element regex*, for which a validation mechanism has to be implemented. Validation has to be implemented once for each *generic constraint type* which is not expressible in *DL*.

c. type	context class	left p. list	right p. list	classes	c. element	c. value
property	Book	identifier	-	ISBN	∀	-
class	ISBN	-	-	xsd:string	regex	'^\d{9}[\d|X]$'

Table 3. Simple Constraint (Not Expressible in DL)

3.3 *Complex Constraints*

Complex constraints are composed out of *simple constraints* and/or *complex constraints*. *Exclusive or* is a logical operation that outputs true whenever both inputs differ (one is true, the other is false). The *complex constraint context-specific exclusive or of property groups* (*R-13*) restricts individuals of given classes to have properties of only one of multiple mutually exclusive property groups. *Publications*, e.g., are either identified by an *ISBN* and a *title* (for books) or by an *ISSN* and a *title* (for periodical *publications*), but it should not be possible to assign both identifiers to a given *publication*, which can be expressed in *ShEx* as follows:

```
1 Publication {
2     ( isbn string , title string ) |
3     ( issn string , title string ) }
```

As the *The-Great-Gatsby* is a *publication* (`rdf:type(The-Great-Gatsby, Publication)`) having an *ISBN* number (`isbn(The-Great-Gatsby, '978-0241965672')`) and a *title* (`title(The-Great-Gatsby, 'The Great Gatsby')`) and not an additional *ISSN* identifier, the *The-Great-Gatsby* is a valid *publication*. The *complex constraint* is mapped to the *RDF-CV* (see table 4) and expressed in *DL* as follows:

$$\begin{aligned} \text{Publication} \sqsubseteq (\neg \text{E} \sqcap \text{F}) \sqcup (\text{E} \sqcap \neg \text{F}) \\ \text{E} \equiv \text{A} \sqcap \text{B} \\ \text{F} \equiv \text{C} \sqcap \text{D} \\ \text{A} \sqsubseteq \geqslant 1 \text{ isbn.string} \sqcap \leqslant 1 \text{ isbn.string} \\ \text{B} \sqsubseteq \geqslant 1 \text{ title.string} \sqcap \leqslant 1 \text{ title.string} \\ \text{C} \sqsubseteq \geqslant 1 \text{ issn.string} \sqcap \leqslant 1 \text{ issn.string} \\ \text{D} \sqsubseteq \geqslant 1 \text{ title.string} \sqcap \leqslant 1 \text{ title.string} \end{aligned}$$

Complex constraints are composed of many other *complex* (e.g. *minimum and maximum qualified cardinality restrictions*) and *simple constraints* (e.g. constraints on sets). As *exact (un)qualified cardinality restrictions (=n)* and *exclusive or (XOR)* are frequently used *complex constraints*, we propose using them as *simple constraints* - in terms of syntactic sugar. As a consequence, the *context-specific exclusive or of property groups* constraint above is represented as a *generic constraint* more intuitively and concisely (see table 5).

3.4 *Complex Constraints* as *Simple Constraints*

Almost 15 percent of all constraint types are *complex constraints* which can be simplified and therefore formulated as *simple constraints* when using them in terms of syntactic sugar (see section 6).

c. type	context class	left p. list	right p. list	classes	c. element	c. value
class	Publication	-	-	¬E ⊓ F, E ⊓ ¬F	⊔	-
class	¬E ⊓ F	-	-	¬E, F	⊓	-
class	E ⊓ ¬F	-	-	E, ¬F	⊓	-
class	¬E	-	-	E	⊓	-
class	E	-	-	A, B	⊓	-
class	¬F	-	-	F	⊓	-
class	F	-	-	C, D	⊓	-
class	A	-	-	A1, A2	⊓	-
property	A1	isbn	-	string	⩾ ⩽	1
property	A2	isbn	-	string	⩾ ⩽	1
class	B	-	-	B1, B2	⊓	-
property	B1	title	-	string	⩾ ⩽	1
property	B2	title	-	string	⩾ ⩽	1
class	C	-	-	C1, C2	⊓	-
property	C1	issn	-	string	⩾ ⩽	1
property	C2	issn	-	string	⩾ ⩽	1
class	D	-	-	D1, D2	⊓	-
property	D1	title	-	string	⩾ ⩽	1
property	D2	title	-	string	⩾ ⩽	1

Table 4. Complex Constraints

c. type	context class	left p. list	right p. list	classes	c. element	c. value
class	Publication	-	-	E, F	XOR	
class	E	-	-	A, B	⊓	
class	F	-	-	C, D	⊓	
property	A	isbn	-	string	=	1
property	B	title	-	string	=	1
property	C	issn	-	string	=	1
property	D	title	-	string	=	1

Table 5. Simplified Complex Constraints

There are three forms of *OWL RBox axioms*: *role inclusions*, *equivalence* and *disjointness*. *OWL* provides a variety of other axiom types: *role transitivity*, *symmetry*, *asymmetry*, *reflexivity* and *irreflexivity*. These axiom types are sometimes considered as basic axiom types in *DL* - using some suggestive notation such as `Trans(ancestorOf)` to express that the role *ancestorOf* is transitive. Such axioms, however, are just syntactic sugar - all role characteristics can be expressed using the basic features of *DL*. The *irreflexive object properties* constraint type (*R-60*) restricts that no individual is connected by a given object property to itself [4]. With the following *irreflexive object property* constraint, e.g., one can state that individuals cannot be authors of themselves:

$$\top \sqsubseteq \neg \exists authorOf.Self$$

When mapped to the *RDF-CV* (see table 6), the *complex constraint* aggregates three *simple constraints* (one *property* and two *class constraints*):

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

Table 6. Irreflexive Object Properties as Complex Constraints

When using the *OWL RBox axiom role irreflexivity* in terms of syntactic sugar, the *complex constraint* can be expressed more concisely in form of a *simple property constraint* with exactly the same semantics (see table 7):

c. type	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	authorOf	-	-	irreflexive	-

Table 7. Irreflexive Object Properties as Simple Constraints

The *primary key properties* constraint type (*R-226*) is often useful to declare a given (datatype) property as the "primary key" of a class, so that a system can enforce uniqueness. *Books*, e.g., are uniquely identified by their *ISBN* number, i.e., the property *isbn* is inverse functional (`funct` $isbn^-$), which can be represented by the *RDF-CV* in form of a *complex constraint* (see table 8):

c. type	context class	left p. list	right p. list	classes	c. element	c. value
property	$\top$	$isbn^-$	$isbn^-$	-	inverse	-
property	Book	$isbn^-$	-	-	functional	-

Table 8. Primary Key Properties as Complex Constraints

Keys, however, are even more general, i.e., a generalization of *inverse functional properties* [6]. A key can be a datatype, an object property, or a chain of properties. For these generalization purposes, as there are different sorts of keys, and as keys can lead to undecidability, *DL* is extended with *key boxes* and a special construct *keyfor* [5]. When using the *keyfor DL* construct (`isbn keyfor Book`), the *complex constraint* can be expressed by only one *simple property constraint* (see table 9 for the mapping to the *RDF-CV*).

c. type	context class	left p. list	right p. list	classes	c. element	c. value
property	Book	isbn	-	-	keyfor	-

Table 9. Primary Key Properties as Simple Constraints

4 How to Improve the Understandability of Constraints

RDF data providers have to ensure that their data conforms to constraints which are explicitly defined within contracts between data providers and consumers. As there is no standard way to express constraints, they may be represented by a variety of constraint languages[10] - each of them different in syntax and semantics. To overcome the necessity to understand diverse constraint languages, we propose to transform any *specific constraint* (sc_α) (expressed by any constraint language α) into any other *specific constraint* (sc_β) (expressed by any other constraint language β) - in case they have the same semantics. We use the *RDF-CV* to intermediately represent constraints in a generic way for these transformations. By defining mappings $(m(sc_\alpha), m(sc_\beta))$ between equivalent *specific*

[10] Candidates for constraint languages are, i.a., SPIN, OWL 2, DSP, ShEx, ReSh, Bibframe, DQTP, Pellet ICV, RDFUnit, SPARQL, and Stardog ICV

constraints and the corresponding *generic constraint* (gc) we are able to convert them automatically:

$$gc = m(sc_\alpha) = m(sc_\beta)$$

We do not need to define mappings between each pair of semantically equivalent *specific constraints*. Let's assume we are able to express a particular constraint type by 5/6/7/8/9/10 constraint languages. Without the intermediate *generic constraint*, we would have to define for each constraint type *1* mapping for each pair of *specific constraints* (expressed by different constraint languages) - that are $\binom{n}{2}$ mappings (10/15/21/28/36/45 mappings). With an intermediate *generic constraint*, however, we only need to define for each constraint type n mappings (1 mapping for each *specific constraint*) from n *specific constraints* to the corresponding *generic constraint* (5/6/7/8/9/10 mappings).

$$\text{Publication} \equiv \exists \text{ publisher.Publisher}$$

The *existential quantification* (*R-86*) above restricts *publications* to have at least one *publisher* and can be expressed by at least 10 different constraint languages[11]:

```
# OWL 2:
[ a owl:Restriction ;
  owl:onProperty publisher ;
  owl:someValuesFrom Publisher ;
  rdfs:subClassOf Publication ] .

# SPARQL:
ASK { ?x a Publication . FILTER NOT EXISTS { ?x publisher [ a Publisher ] } . }

# ShEx:
Publication {
    publisher @Publisher{1,} }

# DSP:
descriptionTemplate a dsp:DescriptionTemplate ;
    dsp:resourceClass Publication ;
    dsp:statementTemplate [ a dsp:NonLiteralStatementTemplate ;
        dsp:minOccur 1 ; dsp:maxOccur "infinity" ;
        dsp:property publisher ;
        dsp:nonLiteralConstraint [ a dsp:NonLiteralConstraint ;
            dsp:valueClass Publisher ] ] .

# ReSh:
Publication a rs:ResourceShape ; rs:property [
    rs:propertyDefinition :publisher ;
    rs:valueShape :Publisher ;
    se:min 1 ; se:maxundefined true ; ] .

# SPIN:
CONSTRUCT {
    _:cv a spin:ConstraintViolation ; spin:violationRoot ?subject ;
         rdfs:label ?violationMessage ; spin:violationPath ?property . } WHERE
{ ?subject a ?classSubject . FILTER NOT EXISTS { ?subject ?property [ a ?classObject ] } . }
```

[11] OWL 2, SPARQL, ShEx, Bibframe, DSP, ReSh, SPIN, DQTP, Pellet IVC, and Stardog ICV

The *DL* statement `Publication` ≡ ∃ `publisher.Publisher` is mapped to a *(simple) generic constraint* conforming to the *RDF-CV* (see table 10). Then, one mapping ($m(sc_{OWL2})$, $m(sc_{ReSh})$, $m(sc_{ShEx})$, ...) is defined for each *specific constraint* (sc_{OWL2}, sc_{ReSh}, sc_{ShEx}, ...) to the corresponding *generic constraint* (gc).

c. type	context class	left p. list	right p. list	classes	c. element	c. value
property	Publication	publisher	-	Publisher	∃	

Table 10. Existential Quantification as Simple Constraints

5 How to Provide Constraint Validation Out-Of-The-Box

SPARQL is generally seen as the method of choice to validate RDF data according to certain constraints, although, it is not ideal for their formulation. In contrast, OWL 2, DSP, ReSh, and ShEx constraints are comparatively easy to understand. We use SPIN as basis to define a validation environment in which the validation of any constraint language[12] can be implemented by representing them in SPARQL (the implementation can be tested at http://purl.org/net/rdfval-demo). The SPIN engine checks for each resource if it satisfies all constraints, which are associated with the classes assigned to the resource, and generates a result RDF graph containing information about all constraint violations [2]. This way, we are able to offer RDF data validation according to all *specific constraints* (expressed by any constraint language) by defining mappings from each *specific constraint* to *SPIN*. We already specified mappings to *SPIN* for all *OWL 2* and *DSP*[13] constructs and for some *ReSh* and *ShEx* constructs. As constraint languages differ in syntax and semantics it is rather difficult to ensure that semantically equivalent *specific constraints* always lead to identical validation results.

We do not have to map each *specific constraint* to SPIN and we get identical validation results for semantically equivalent *specific constraints* (expressed by different constraint languages), (1) if we map semantically equivalent *specific constraints* to the corresponding *generic constraint* and (2) if we map the *generic constraint* to SPIN. As a consequence, we only have to define one SPIN mapping for each constraint type corresponding to an RDF validation requirement. 19 of the overall 74 identified constraint types can be expressed by at least four constraint languages [3]. This means that there must be 76 (four for each constraint type) implementations to validate RDF data according to 76 *specific constraints* - instead of only 19 implementations (one for each *generic constraint*) leading to identical validation results.

Within the *RDF Validator*[14], we define one SPIN construct template for each *generic constraint* and therefore constraint type. A SPIN construct template

[12] The only limitation is that constraint languages must be represented in RDF

[13] SPIN mappings: https://github.com/boschthomas/rdf-validation/tree/master/SPIN

[14] For details about the validation environment see [2]

contains a SPARQL CONSTRUCT query which generates constraint violation triples (*spin:ConstraintViolation*) indicating the subject (*spin:violationRoot*) and the properties (*spin:violationPath*) causing constraint violations, and the reason why constraint violations have been raised (*rdfs:label*). A SPIN construct template creates constraint violation triples if all triple patterns within the SPARQL WHERE clause match.

$$\text{Publication} \equiv \forall \text{ author.Person}$$

According to the *universal quantification* (*R-91*) above, *authors* of *publications* must be *persons*. The SPARQL CONSTRUCT query for the constraint type *universal quantification* is shown below:

```
CONSTRUCT {
    [   a spin:ConstraintViolation ; spin:violationRoot ?subject ;
        rdfs:label ?violationMessage ; spin:violationPath ?lp1 . ] }
WHERE {
    [   a gclo:PropertyConstraint ;
        gclo:contextClass ?cc ;
        gclo:leftProperties ( ?lp1 ) ;
        gclo:classes ( ?c1 ) ;
        gclo:constrainingElement "universal quantification" ] .
    ?subject a ?cc .
    ?subject ?lp1 ?o .
    FILTER NOT EXISTS { ?o a ?c1 } . }
```

We provide a mapping from the *RDF-CV* to SPIN[15] in order to automatically validate RDF data complying with each type of *generic constraint* (*simple*, *complex*, *property*, and *class constraints*).

6 Evaluation

We evaluated to which extend the five most promising constraint languages on being the standard fulfill each of the overall 74 requirements to formulate RDF constraints; each of them corresponding to a constraint type. We recently published a technical report[16] (serving as appendix of this paper) in which we explain each constraint type in detail and give examples for each (represented by different constraint languages). The technical report also contains mappings to *DL* to logically underpin each constraint type and to determine which *DL* constructs are needed to express each constraint type [3]. If a constraint type can be expressed in *DL*, we added the mapping to *DL* and to the *generic constraint* conforming to the *RDF-CV*. In contrast, if a constraint type cannot be expressed in *DL*, we only added the mapping to the *generic constraint*. This way, we prove that each constraint type can be mapped to a *generic constraint* and therefore be represented generically by means of the *RDF-CV*. Table 11 displays the classification of constraint types according to the dimensions *context*, *complexity*, and *DL expressivity*:

[15] RDF-CV to SPIN: https://github.com/boschthomas/RDF-CV-2-SPIN
[16] Available at: http://arxiv.org/abs/1501.03933

Classes of Constraint Types	#	%
Property Constraints	48	64.86
Class Constraints	17	22.96
Property and Class Constraints	9	12.16
Simple Constraints	46	62.16
Simple Constraints (Syntactic Sugar)	10	13.51
Complex Constraints	18	24.32
DL Expressible	51	68.92
DL Not Expressible	23	31.08
Total	74	100

Table 11. Evaluation

Constraint types can be classified as *property constraints* and *class constraints* (*context*). Two thirds of the total amount of constraint types are *property constraints* (e.g., *R-86: existential quantification* and *R-91: universal quantification*), one fifth are *class constraints* (e.g., *R-30/37: allowed values* for objects/literals), and approx. 10% are composed of both *property and class constraints* (e.g., *R-13: context-specific exclusive or of property groups*). According to the dimesion *complexity*, constraint types can be either atomic (*simple constraints*) or complex (*complex constraints*), i.e., created out of *simple* and/or *complex constraints*. Almost two thirds of the constraint types are *simple constraints* (e.g., *R-55: object property paths*), a quarter are *complex constraints* (e.g., *R-45: ranges of literal values*), and nearly 15 percent are *complex constraints* which can be formulated as *simple constraints* when using them in terms of syntactic sugar (e.g., *R-81/75/82/76/80/74: (un)qualified cardinality restrictions*). Constraint types can either be *expressible in DL* or *not* (*DL expressivity*). The majority - nearly 70% - of the overall constraint types are expressible in *DL*. *Literal pattern matching* (*R-44*) is an example of a constraint type which cannot be represented in *DL*, but can also be represented generically by means of the *RDF-CV*. Altogether, most of the constraint types are *simple constraints on properties* which are *expressible in DL*. Thus, the majority of constraint types are directly and relatively easy formulated in form of *simple constraints* which can be mapped to equivalent *DL* constructs.

7 Related Work

Thomas: 1 page / explain how typical constraints are expressed by different constraint languages

8 Conclusion and Future Work

It is part of future work (1) to extend the *RDF Validator* by creating *generic constraints* automatically according to inputs of domain experts who may not be familiar with the formulation of constraints, (2) to offer bidirectional transformations between all *specific constraints* (expressed by all existing constraint

languages) and the corresponding *generic constraints*, and (3) to provide translations between semantically equivalent *specific constraints* (expressed by any constraint language) by using *generic constraints* as an intermediate transformation step.

References

1. Thomas Bosch and Kai Eckert. Requirements on rdf constraint formulation and validation. *Proceedings of the DCMI International Conference on Dublin Core and Metadata Applications (DC 2014)*, 2014.
2. Thomas Bosch and Kai Eckert. Towards description set profiles for rdf using sparql as intermediate language. *Proceedings of the DCMI International Conference on Dublin Core and Metadata Applications (DC 2014)*, 2014.
3. Thomas Bosch, Andreas Nolle, Erman Acar, and Kai Eckert. Rdf validation requirements - evaluation and logical underpinning. 2015.
4. Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A description logic primer. *CoRR*, abs/1201.4089, 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.
6. Michael Schneider. OWL 2 Web Ontology Language RDF-Based Semantics. W3C recommendation, W3C, October 2009.