Evaluating Ontology Completeness via SPARQL and Relations-between-classes based Constraints

Philippe A. Martin
EA2525 LIM, Uni. of La Réunion
(and adjunct researcher of the School
of ICT, Griffith University, Australia)
F-97490 Sainte Clotilde, France

***Abstract*—This article first distinguishes constraints from rules, and descriptive constraints from prescriptive ones. Both kinds can be used to calculate a constraint-based completenesses (as opposed to a real-world-based completeness), i.e. evaluating how much of a knowledge base is complete with respect to some constraints, e.g. for evaluating how well this base follows given ontology design patterns or best practices. Such evaluations may also guide knowledge elicitation and modelisation. This article explores the ways constraints can be represented via relations between classes, hence via any knowledge representation language (KRL) that has an expressiveness at least equal to RDF or RDFS. Compared to the popular practice of both representing and checking constraints via queries, this approach is as simple, offers more possibilities for exploiting both knowledge and constraints, and permits the selection and use of inference engines adapted to the expressiveness of the exploited knowledge instead of the use of restricted or *ad hoc* constraint-validation tools. This approach is also modular in the sense it separates content from usage: the represented "content focused constraints" can then be exploited via few "content independent" queries, one for each usage and kind of constraint. This approach provides more possibilities.**

Keywords—constraints, ontology completeness, OWL, SPARQL

I. Introduction

Knowledge representations (KRs) are formal descriptions enabling automatic logical inferencing, and thus automatic KR comparison, search, merge, etc. KRs are logic formulas, e.g. the *binary predicates* of 1st-order logic; these predicates are called *triples* or *property instances* in RDF and *binary relations* in Conceptual Graphs (CGs) [1]. For the purpose of clarity, this article uses the intuitive terminology of CGs: (information) *objects* are either *types* or *individuals*, and types are either *relation types* or *concept types* (*classes* and *datatypes* in RDF). A formal knowledge base (KB) is a collection of such objects written using a KR language (KRL). An ontology is a KB that is essentially about types, rather than about individuals.

Creating a KB or evaluating its quality – for knowledge sharing or exploitation purposes, or for designing or generating software, or evaluating their qualities – are difficult. Models and constraints (e.g. design patterns) help these tasks and can be stored into an ontology. E.g., the author of this article is building an ontology representing and organizing ontology design patterns as well as software design patterns. Reference [2], a survey on quality assessment for Linked Data, provides many dimensions and metrics for evaluating the quality of KBs and hence helping the selection or design of KBs. One of the quality dimensions is the *(degree of) completeness of a KB with respect to some criteria or constraints*: concisely, "*its completeness*". Evaluating this degree is common in various tasks or fields but is performed differently by different tools and sometimes in implicit or *ad hoc* ways. Examples of such tasks or fields are: i) the automatic/manual extraction of knowledge or the creation of a KB, ii) the exploitation of ontology design patterns, KB design libraries (e.g., the KADS library) or top-level ontologies (e.g., DOLCE), and iii) the evaluation of ontologies or, more generally, datasets. In this last field, as noted in [2], *completeness* commonly refers to a *degree* to which the "information *required* to satisfy some given criteria or a given query" are present in the considered dataset. To complement this very general definition, this article distinguishes two kinds of completeness:

- *Constraint-based completeness* measures the percentage of elements in a dataset that satisfy *explicit* representations of what *must* or *must not* be represented in the dataset. These representations are constraints such as integrity constraints or, more generally, those expressed by ontology design patterns and schemas of databases or of structured documents. E.g.: the constraint that, in a particular dataset, at least one movie must be associated to each movie actor.

- *Real-world-based completeness* measures the degree to which certain real-world information are represented in the dataset. E.g., regarding movies associated to an actor, calculating the completeness may consist in dividing "the number of movies associated to this actor in the dataset" by "the number of movies he actually played in". Either the missing information are found in a *gold standard dataset* or the degree is estimated via *completeness oracles* [3], i.e. rules or queries estimating what is missing in the dataset to answer a given query correctly. The four kinds of completeness collected by [2] – *schema/property/population/interlinking* completeness – assume a closed-world-assumption and a gold standard dataset. Thus, they are *real-world based completenesses*.

One way to define or calculate the *constraint-based completeness of a KB* is to divide "the number of statements satisfying the constraints in that KB" by "the total number of statements in the KB". As a variant, instead of statements only, one may want to consider objects, i.e. measure the percentage of objects for which all relations from/to them satisfy the constraints. Other variants may be defined by considering only certain kinds of objects or statements. Defining constraints via KRs, instead of via queries, permits the definition of "content-independent (alias, domain-independent) queries" to exploit these constraints. Otherwise, a different (content-dependent) query has to be created for each variant of constraint based checking or completeness. Because of this lack of modularity,

when stored in an ontology, content-dependent queries are also less easily organized than content-independent ones.

This article *does not address real-world based completeness* but the techniques this article proposes may also be used for representing certain domain-specific parts of the rules used for calculating real-world based completeness. From now on, "completeness" refers to *constraint-based completeness*.

Section II explores the first research question of this article: what does the expression "must and must not be represented in the dataset" entail or, more precisely, given the "descriptive vs. prescriptive" distinction, what kinds of constraints need to be considered for evaluating *constraint-based completeness* via content-independent queries?

Section III proposes an *approach* to answer a second research question: how to represent constraints in a KRL independent way – or, more precisely, in any KRL that has an expressiveness at least equal to RDF or RDFS – even though actually defining the semantics of some of these constraints would require much more expressive logics? The proposed solution relies i) on the representation of constraints via restricted constructs based on *relations between classes (or to classes)*, e.g. rules using the rdfs:subClassOf relation for representing a restricted implication between the condition and conclusion of the rule, ii) on the use of certain special types for specifying that a statement is a constraint of a certain kind, and iii) on the exploitation of these types in content-independent queries. Such constraint representations can then be exploited via most inference engines and KR query languages instead of tools tied to a particular KRL or goal (knowledge acquisition, ontology design pattern application, ontology evaluation, etc.).

Section IV shows commands (queries or update requests) answering a third research question: how to implement the above cited *approach* in SPARQL or slight extensions of it.

Section V.A illustrates applications or use cases for the given content-independent queries. It is actually a summary of Section IV in [4], an unpublished extension of this article and a companion Web site for this article. Section V.B evaluates the proposed approach and compares it to other ones.

II. Definitions For The Considered Constraints

A. Considered Constraints: Just For Checks, Not Inferences

In this article, as in KIF (Knowledge Interchange Format) [5], a rule is a statement that can be represented in the form "X =>> Y" where "=>>" is a restricted version of the logical implication ("=>"): it supports modus ponens, not modus tollens.

A *rule allowing the derivation of a non-modal statement* is a rule that *can* be represented in the form "X =>> Y" where Y does *not* include a modality (e.g., *must*). An example is "if x is a Person then x has a parent". If this statement and "Tom is a Person" are in a KB, an inference engine can derive the non-modal statement "Tom has a parent".

Reference [6] defines constraints as *positive* or *negative*, respectively expressing statements of the form "if A, B *must* be true" and "if A, B *must be false*". Thus, [6] defines constraints as rules where the conclusion has a "must" modality. These are the kinds of constraints considered in this article, *with the interpretation that* in such constraints the "must" entails that the constraints can only be used for checking statements, i.e. that they are *not* rules allowing the derivation of non-modal statements. More formally, this means that such positive and negative constraints can respectively be *translated* into the forms "A ∧ ¬B =>> false" and "A ∧ B =>> false" where A and B do *not* contain a "must" modality and A may be empty. As an example, consider the positive constraint "if x is a Person, x *must* have a parent". From this *constraint* and the fact "Tom is a Person", an inference engine must *not* derive "Tom has a parent". It may derive "Tom *must* have a parent" but, in practice, such derivation is not made. As a somewhat opposite example, RDFS-aware engines do not exploit relations of type rdfs:domain or rdfs:range as relation signature *constraints* but as *inference supporting statements*: when a relation *r* has a type partially defined by an rdfs:domain (vs. rdfs:range) relation, RDFS-aware engines may infer a type for the source (vs. destination) of *r*.

In this article, constraints that are directly represented in a form ending by "=>> false" – or, equivalently, "=>> ⊥" – are called *constraints in inconsistency-implying form*. Not all KRLs allow to represent rules (instead of – or in addition to – implications); in those that do, representing negative constraints using the inconsistency-implying form is easy but using this form for representing positive constraints may not possible: the KRL may not permit the representation of the negation in the "¬B" part. This in why in this article i) negative constraints are represented in inconsistency-implying form, and ii) positive constraints are in the form "A =>> B" *but with a representation supporting the distinction between a constraint and an actual rule*. Furthermore, as in most rule-based systems, in the rest of this article the A and B parts share variables. More precisely, these parts are representations of relations from a same object (i.e. from a type or an individual, including a relation or a more complex statement since they are particular kinds of individuals).

In the research literature on constraints, these ones are *generally* not represented – or checked – via modal logic based KRLs but rather using *queries*, e.g. via SPARQL or the nonmonotonic-epistemic-logic query language EQL-Lite [7]. In (unidirectional) rule based systems, *rules* with empty conclusions (or "false" as conclusions) are handled like constraints. However, this is a particularity of these systems. It should not be relied upon for general knowledge representation purpose. For such a purpose, the special semantics of constraints has to be made explicit via special syntactic sugar or special types. Since KRLs rarely propose syntactic sugar for expressing constraints, a more generic approach for expressing that a statement is a constraint, as opposed to an inference supporting statement, is to state that this statement is an instance of a type expressing a particular kind of constraint (as explained in Section III.A). Then, these constraints can be retrieved and exploited by content-independent queries such as those provided below. These constraints can also be directly interpreted and exploited by inference engines designed to take into account the used constraint types. In any case, either i) constraints are not represented in a way they can be exploited as inference supporting statements, or ii) the results of these inferences must not be detrimental, i.e., must not influence the checking of constraints. Both techniques will be illustrated below.

B. Prescriptive (i.e. Not Using All Inferences) vs. Descriptive

As noted in [8], a common distinction between engineering models is whether they are i) *descriptive* of some reality, e.g. like most ontologies (e.g., by default, ontologies written in RDF

or OWL), or ii) *prescriptive* of what must be in the considered dataset, as with system specifications, meta-models, XML schemas, database schemas, SHACL statements, etc. Similarly, this article distinguishes two kinds of constraints. First, like definitions or axioms, *descriptive constraints* enable inference engines to check *the use* of certain formal terms, if and only if these terms are used. On the other hand, *prescriptive constraints* enable inference engines to check that certain formal terms are *actually* used (not just inferred) or not used, under certain conditions. E.g., prescriptive constraints can be used for checking that if the instances of a type are defined as (necessarily) having certain relations, these relations are *explicitly* given by users whenever they create an instance of such a type. Here, "explicitly" emphasizes that these relations must not exist just because they were automatically deduced, e.g. by inheritance, but only because they were set by a user (manually or automatically). As an example, assume that a KB includes the rule "if x is a Person, x has a parent" and that a user enters that "John is a Person" in the *base of facts* of this KB (this base is the set of relations from/to individuals; for a description-logic based KB, this is its A-box). Even if this KB also includes the descriptive constraint "if x is a Person, x *must* have a parent *in the represented world (descriptive-must)*", an error message should not be given by a KB checking mechanism since this constraint is satisfied (by inferencing) without the user having to represent a parent for John. On the other hand, if the KB includes the prescriptive constraint "if x is a Person, x *must* have a parent *in the base of facts (prescriptive-must)*", the adding of a new person without a relation to a parent must be rejected. For constraints in inconsistency-implying form, there is no distinction between descriptive and prescriptive: they enable the detection of an incorrect KR whether it has been inferred or not.

To check a positive *prescriptive* constraint, any inference may be useful for *testing the condition* of this constraint, i.e. for *matching* objects in the KB against this condition. As illustrated in the previous paragraph, this is *not* the case when testing the main (alias, first) object of the *conclusion* of the constraint, i.e. the object whose relations are mandatory for all objects matching the condition of the constraint. When testing this first object, if some mechanism automatically associates relations to some of the checked objects – e.g., by dynamic lookup for inherited relations during each object matching, or by forward chaining saturation – this mechanism must be temporarily *disabled or bypassed*. However, disabling or changing these mechanisms (or, in other words, the used entailment regime [9]) generally cannot be done in the middle of a query. E.g., SPARQL does not permit such a change. Hence, instead, bypassing methods are needed. Section III.B proposes one.

To sum up, such prescriptive constraints are original, enable checks that descriptive constraints cannot, and are not *equivalent to the use of the closed world assumption*. The techniques presented by this article for defining and checking prescriptive constraints can be performed with open-world assumption. Non-modal logical expressions are only descriptive. E.g., simply stating that "any Person (necessarily) has a parent" is only descriptive. Content-independent queries, special procedures or inference engines exploiting prescriptive constraints need to distinguish them from descriptive ones via their types.

C. Descriptive Constraints Restricted To Named Individuals

When using a constraint to check if certain objects in a KB satisfy a methodology or an ontology design pattern, one might want to take into account automatically deduced relations but only if they are *from or to named individuals* ("IRIs" in RDF terminology), not if they are *between anonymous individuals* ("blank nodes" in RDF terminology). Such a "partly descriptive - partly prescriptive" constraint may be termed "descriptive but restricted to named individuals". It requires the author of the constraint to represent which individuals must be named.

Although RDFS provides the type rdfs:label for relations between an individual and its names, it does not provide a type for relating an individual to its identifiers (even though they are unique names) since IRIs (International Resource Identifiers) are directly interpreted as named individuals in RDF. OWL also does not provide such a relation type which could be used for distinguishing which individuals *are (or must be) named* from those who are not (or need not be). In OWL2-DL, this distinction can be made if each named individual is declared as instance of the class owl:NamedIndividual. However, doing so in OWL2 Full still does not permit to make the distinction. SPARQL supports the distinction via the operators isIRI and isBlank. Hence, the solution proposed in this article is to provide i) the relation type cstr:id for enabling the authors of a constraint to specify which individual must have an identifier (as illustrated in the second to last paragraph of Section III), and ii) a SPARQL update request permitting the adding of cstr:id relations to each named individual type in a KB (cf. Section IV.B). Thus, if some descriptive constraints use cstr:id relations and if this last SPARQL update request is run, these constraints will correctly be checked by content-independent queries for descriptive constraints (e.g., see Section IV.D.1).

III. Approach For Representing And Exploiting The Considered Constraints

A. Using Constraint Types

Reference [10] shows that SPARQL queries can represent and check certain kinds of *integrity constraints* that exploit some forms of the Unique Name Assumption or Closed World Assumption. Instead, as explained in the introduction, the goal is here to enable the representation of constraints that i) can be exploited via *content-independent* queries, ii) can be represented via any KRL that has an expressiveness at least equal to RDF or RDFS, and iii) can be marked as descriptive or prescriptive (this distinction is not made in [10]).

To that end, the proposed approach is to introduce a few types for constraints. By setting instanceOf or subtypeOf relations from certain KRs to some of those types, KB authors can state that these KRs are constraints and can indicate which kind of constraints. Thus, these constraints can be exploited by content-independent queries or inference engines that understand the used constraint types. For these engines, the types change the way the statements must be interpreted. This approach is similar to the use of OWL2 types in RDF statements and their exploitation by OWL2-aware inference engines. The name of the proposed ontology of constraint types is CSTR. In this ontology, cstr:Constraint is the supertype of all types of constraints. Similarly, the type cstr:Prescriptive_constraint, a subtype of cstr:Constraint, enables one to state that some rules are actually prescriptive constraints or to retrieve all and only such constraints. The prefix "cstr:" in these identifiers is an abbreviation for the namespace http://www.webkb.org/kb/it/CSTR. Other types are later referred to, when needed. Presenting the rest of CSTR is not needed in this article.

B. *Using "Clones Without Types" For Bypassing Certain Inferences When Checking The Conclusions Of Positive Prescriptive Constraints*

For adequately checking positive prescriptive constraints Section II.B introduced the need for temporarily disabling or bypassing "inference mechanisms that automatically associate relations to objects" when testing the first object of the conclusion of the constraint. This subsection proposes a method to so. Statically (i.e. via a pre-treatment of the KB like the one given in Section IV.C) or dynamically (i.e. during the checking of such constraints), this method creates a "clone without type" of each object matching the condition of such a constraint and then, when checking its conclusion, does so on the clone instead of the original object. The clone has the same relations as the original object except for *instanceOf* relations (it has none; furthermore, if it is a named individual, it has an identifier different from the original object). Thus, by so using *clones without types*, "inferences exploiting *types* to associate relations to an object" are avoided. As an abbreviation, from now on, this is referred to as avoiding *inheritance*. In the case of RDFS or OWL entailments, "avoiding inheritance" means that, when searching relations associated to an object, the types of this object and their superclasses are not exploited. Creating *clones without types* is not necessarily easy since there may be information in the KB that lead certain inference engines to regenerate types for some clones. E.g., assuming there is an rdfs:domain relation from the relation type parent to Person, if an object of type Person is source of a parent relation and this object has its type removed, an inference engine may set it again. To avoid such a case, instead of using rdfs:domain or rdfs:range relations, one may write inconsistency-implying constraints that are equivalent to these relations except that they are usable only for checking purposes. When SPARQL is used for creating a *clone without type*, as illustrated in Section IV.C, another potential problem is that the whole KB is duplicated, not just one object. Finally, this method based on clones without types does not work if there are inferences that do not exploit types (e.g. via duck typing instead of inheritance) or if a forward chaining saturation on the KB is automatically run before the above cited pre-treatment. However, these last two cases are rare.

This method relies on a temporary update of KRs before their checking by an inference engine. Thus, this method does *not* rely on a particular KRL, inference engine or tool feature. I.e., this solution is *KRL independent* and *tool independent*: it can be used with any KRL and any tool. Hence, depending on the domain and application, different inference engines can be reused to check or evaluate ontology completeness. However, with some query languages such as current standard versions of SPARQL, the temporary update cannot be done dynamically: a KB pre-treatment is necessary. This is a limitation since KB servers, e.g. SPARQL endpoints, rarely allow their users to modify a KB for checking it. With LDScript [11], an extension of SPARQL, the temporary update can be done dynamically. As with SPARQL, the whole KB is duplicated but now it is temporary and done every time an object is matched with the conclusion of a prescriptive constraint (cf. Section IV.D.2).

C. *Representing Constraints Via Relations Between Classes*

1) Approaches:

One way to represent and exploit (simple) *rules* in a KRL that has an expressiveness at least equal to RDF or RDFS is to use an rdfs:subClassOf relation for representing the implication between the condition and conclusion of a rule (as in OWL-ER [12], an intersection between OWL2, Datalog+ and RuleML). However then, either this implication must not be used for modus tollens or the results must not be detrimental. The situation is not much more complex when subclassOf rules are used as a way to represent *constraints*. There are three cases.

- If the conclusion is (equivalent to) owl:Nothing, i.e. if the *inconsistency-implying form* is used, the rule is semantically a constraint and, depending on the inference engines, modus tollens may or may not be a danger,
- Otherwise, if a *prescriptive* constraint is represented, the "Clones Without Types" based method prevents the results of modus ponens or modus tollens to influence the checking of constraints (this is where these results could have been detrimental).
- Otherwise, i.e. if a *descriptive* constraint is represented, one must use an inference engine that does not exploit rules for modus ponens nor modus tollens when the condition of the rule is subtype of cstr:SubclassOf-based_constraint_condition or instance of cstr:Type_of_subclassOf-based_constraint_condition.

In other words, using subclassOf-based constraints when inferences based on subclassOf relations then have to be ignored is generally not relevant. However, the idea of using classes for representing the conditions and conclusion of a constraint without using variables is interesting. Here are two simple ways.

- The "*subclassOf-analogous" way*: it consists in relating the condition class and the conclusion class by a relation that is not a subclassOf one. To do so, CSTR proposes the relation types cstr:descriptive_constraint_conclusion and cstr:prescriptive_constraint_conclusion.
- The "*individual-based constraint" way*: it consists in creating a constraint *individual* and, from it, relations to express its type (i.e., descriptive vs. prescriptive) and the classes for its condition and its conclusion. To support this, CSTR proposes the relation types cstr:condition_class and cstr:conclusion_class. Here, the classes are not *directly* connected by a relation but *indirectly* connected. A type could be used instead of an individual but, in the general case, this would not bring more advantages.

A disadvantage of any solution using relations from/to classes when these relations are not subclassOf ones is that the result requires a KRL with an expressivity at least equal to RDF. For OWL-based representations, this means interpreting them with the RDF-Based Semantics, not the OWL2 Direct Semantics. Reference [4], the companion Web site for this article, proposes and categorizes types and requests to represent and exploit constraints in the three above ways. Because of their similarities, in this article *mainly only the requests for the last way are given* and the CSTR ontology needs not be further detailed.

2) Examples of Individual-based Constraints:

In this article, the Turtle notation is used when SPARQL is not used since SPARQL is an extension of this notation. For clarity purposes, the names of relation types have a lowercase initial while other names have an uppercase initial. In SPARQL, Turtle or other graph-based notations, a statement of the form "SourceConcept *rel1* DestConcept1a, DestConcept1b; *rel2* DestConcept2a, DestConcept2b" can for example be read

"SourceConcept has for *rel1* DestConcept1a and DestConcept1b, and has for *rel2* DestConcept2a, and has for *rel2* DestConcept2b".

Here is the positive prescriptive constraint "if x is a Person, x *must* have a parent *in the base of facts (prescriptive-must)*" represented as an individual-based constraint (using CSTR):

```
[] rdf:type cstr:Prescriptive_constraint;   cstr:condition_class :Person;
   cstr:conclusion_class  [rdf:type owl:Class;  owl:equivalentClass
                              [rdf:type owl:Restriction;  owl:onProperty :parent;
                               owl:someValuesFrom :Person] ].
```

For the general *descriptive* version of this constraint, it is sufficient to replace "prescriptive" by "descriptive" in the previous representation. Here is a version where parents are restricted to be *named individuals* (i.e. to have a cstr:id relation):

```
[] rdf:type cstr:Descriptive_constraint;   cstr:condition_class :Person;
   cstr:conclusion_class  [rdf:type owl:Class;   owl:equivalentClass
                              [rdf:type owl:Restriction;  owl:onProperty :parent;
                               owl:someValuesFrom :Named_person] ].
```

The *negative* constraint "if x is a Person_without_parent, x must not have a parent" may be translated into the inconsistency-implying form "x is a Person_without_parent =>> false". Here is its representation as an individual-based constraint:

```
[] rdf:type  cstr:Descriptive_constraint;  #optional rdf:type relation
   cstr:condition_class [rdf:type owl:Class;     owl:equivalentClass
                            [rdf:type owl:Restriction;    owl:onProperty :parent;
                             owl:maxCardinality "0"^^xsd:nonNegativeInteger] ];
   cstr:conclusion_class  owl:Nothing.
```

IV. Implementation In SPARQL or Slight Extensions Of It

In some extensions of SPARQL, e.g. LDScript [11], the next commands (queries or update requests) can be grouped into scripts or functions. Variable names begin by "?".

A. First Example of KB Pre-treatment: Creating "Identifier Relations" for Named Individuals

From each selected named individual, the next command adds a cstr:id relation with destination the identifier of that individual. Thus, as explained in Section II.C, queries checking descriptive constraints also work on those that include cstr:id relations, i.e. that have restrictions to named individuals. Here, only an individual that has a type with a superclass and that has other relations is selected since in practice only such an individual might violate a constraint. To search for individuals, just looking for each object that is not a relation and that does not have rdfs:Class as type would be an incomplete strategy and many conditions would have to be added for filtering out objects such as i) classes defined via an equivalence to a restriction, and ii) owl:Thing and some other types from OWL, RDFS or XSD.

```
INSERT { ?o cstr:id ?id } WHERE                          #?o  is an individual that
{ ?o rdf:type ?t . FILTER NOT EXISTS { ?o rdf:type rdfs:Class }    # is typed,
  ?t rdfs:subClassOf ?superClass . FILTER isIRI(?o)       # is named by an IRI,
  FILTER NOT EXISTS { ?o cstr:id ?id }                    # had no cstr:id relation,
  {?o ?r ?o2  FILTER(?r!=rdf:type)} UNION {?o1 ?r ?o}  # has other relations.
  BIND( str(?o) as ?id )  #?id is now the IRI identifying ?o
}
```

B. Second Example of KB Pre-treatment: Creating "Clones Without Types" of Objects For Exploiting These Objects Without Inheritance Mechanism

Section III.B introduced a method for handling positive prescriptive constraints, i.e. for bypassing or avoiding the relation lookup mechanism above abbreviated as "inheritance". The next command implements the KB pre-treatment supporting the "clones without types" based method when, as is the case with SPARQL, i) a particular entailment regime cannot be changed within a query, and ii) clones cannot be temporarily created within a query. For the sake of clarity, this command assumes that the KB does not include user-defined 2nd-order types. For every object ?o in the KB, if this object is an individual, this command creates ?o2, a partial copy of ?o that has the same relations except for rdf:type relations. This partial copy has for identifier the one of ?o but with the suffix "_cloneWithoutType". This command also relates ?o to ?o2 by a relation of type cstr:cloneWithoutType.

```
INSERT {?o cstr:cloneWithoutType ?o2 .   ?o2 ?r ?dest .
          ?o2 cstr:cloneWithoutTypeOf ?o} WHERE   #?o  is an individual that
{ ?o  rdf:type ?t . FILTER NOT EXISTS { ?o rdf:type rdfs:Class }    # has a type
             ?t rdfs:subClassOf ?superClass .        # (which has a superclass),
  FILTER NOT EXISTS { ?o  cstr:cloneWithoutType ?c1 }     # is not a clone,
  FILTER NOT EXISTS { ?c2 cstr:cloneWithoutType ?o }    # has not a clone

  { #Case 1: cloning each individual having at least 1 relation different from
    #          rdf:type and owl:sameAs;  "?o2 ?r ?dest" is inserted
    ?o ?r ?dest.  FILTER(?r!=rdf:type)  FILTER(?r!=owl:sameAs)
  }
  UNION #Case 2: cloning each individual not having a relation different from
          #          rdf:type and owl:sameAs;  "?o2 ?r ?dest" is not inserted
  { ?o ?r1 ?dest   #?o has at least one relation from it
    FILTER NOT EXISTS
    {?o ?r ?dest2.  FILTER(?r!=rdf:type)  FILTER(?r!=owl:sameAs) }
  }

  BIND( uri(concat(str(?o),"_cloneWithoutType")) as ?o2 )
}
```

C. Checking Individual-based Positive Descriptive Constraints

The next query lists *every object violating* at least one of the individual-based positive descriptive constraints – including those restricted to named individuals if cstr:id relations have been added to named individuals. As shown by the code, *such an object* satisfies two conditions. First, this object matches – hence, has for type – the condition of a constraint ?posConstr that is of type cstr:Descriptive_constraint. Second, this object does not match the conclusion of the constraint. This query requires a SPARQL engine with an entailment regime enabling the matching (alias, categorization) of an individual with respect to a class expression and thence the deduction of an rdf:type relation between them. *In the code of the commands in this Section IV, such deduced rdf:type relations are highlighted in bold*. For inferencing completeness purposes, such deductions require an entailment regime able to handle the expressiveness used in the constraints and the rest of the KB. The code for checking subclassOf-based or subclassOf-analogous positive descriptive constraints is similar.

```
SELECT  ?objectNotMatchingPosConstr   ?posConstr_condition
          ?posConstr_conclusion  WHERE
{ ?posConstr  rdf:type  cstr:Descriptive_constraint;
                cstr:condition_class   ?posConstr_condition;
                cstr:conclusion_class ?posConstr_conclusion.
  FILTER (?posConstr_conclusion != owl:Nothing)
  ?objectNotMatchingPosConstr  rdf:type  ?posConstr_condition.
  FILTER NOT EXISTS  #no objects satisfying the conclusion
  { ?objectNotMatchingPosConstr  rdf:type ?posConstr_conclusion }
}
```

D. Checking Individual-based Positive Prescriptive Constraints

The next query assumes that the "clones without types" have been statically created as seen in Subsection B. In the rest of this

Section IV, when a command has some code that has *not* been used in a previous command, this code is in italics.

```
SELECT ?objectNotMatchingPosConstr ?posConstr_condition
          ?posConstr_conclusion WHERE
{ ?posConstr rdf:type cstr:Prescriptive_constraint;
               cstr:condition_class ?posConstr_condition;
               cstr:conclusion_class ?posConstr_conclusion.
  FILTER (?posConstr_conclusion != owl:Nothing)
  ?objectNotMatchingPosConstr rdf:type ?posConstr_condition.
  ?objectNotMatchingPosConstr cstr:cloneWithoutType ?cloneWithoutType
  FILTER NOT EXISTS #no objects with clones satisfying the conclusion
  { ?cloneWithoutType rdf:type ?posConstr_conclusion }
}
```

Here is the same query in LDScript, with an embedded query that temporarily creates the above cited partial copies "*on the fly*", thus removing the necessity to modify the KB.

```
SELECT ?objectNotMatchingPosConstr ?posConstr_condition
          ?posConstr_conclusion WHERE
{ ?posConstr rdf:type cstr:Prescriptive_constraint;
               cstr:condition_class ?posConstr_condition;
               cstr:conclusion_class ?posConstr_conclusion.
  FILTER (?posConstr_conclusion != owl:Nothing)
  ?objectNotMatchingPosConstr rdf:type ?posConstr_condition.
  BIND( cstr:getCloneWithoutType(?objectNotMatchingPosConstr)
        as ?cloneWithoutType ) #the called functions are defined below
  BIND( cstr:copyOfKbIntoTemporaryGraphPlusTheCloneWithoutType
          (?objectNotMatchingPosConstr,?cloneWithoutType) as ?g )
  FILTER NOT EXISTS { GRAPH ?g { ?cloneWithoutType
                                 rdf:type ?posConstr_conclusion } }
}

FUNCTION cstr:getCloneWithoutType (?object)
{ uri( concat( str(?object),"_cloneWithoutType") ) }

FUNCTION cstr:copyOfKbIntoTemporaryGraphPlusTheCloneWithoutType
                (?objectNotMatchingPosConstr, ?cloneWithoutType)
{ LET (?g = CONSTRUCT { ?cloneWithoutType ?r ?dest . ?x ?r2 ?y } WHERE
            { VALUES ?cloneWithoutType { UNDEF }
              ?objectNotMatchingPosConstr ?r ?dest . FILTER (?r != rdf:type)
              ?x ?r2 ?y . FILTER (?x != ?objectNotMatchingPosConstr)
            } )
  { xt:entailment(?g) }    #triggers inferences on ?g
}
```

The same "*on the fly*" cloning technique can be used for adding cstr:id relations to named individuals. Thus, this technique also permits the checking of constraints restricted to named individuals without having to modify the KB.

E. Checking Individual-based Inconsistency-implying Constraints

The next query lists every object violating an individual-based inconsistency-implying constraint.

```
SELECT ?objectMatchingNegConstr ?negConstr_condition WHERE
{ ?negConstr cstr:condition_class ?negConstr_condition;
               cstr:conclusion_class owl:Nothing.
  ?objectMatchingNegConstr rdf:type ?negConstr_condition.
}
```

F. Checking SubclassOf-analogous Constraints

The usable content-independent queries here are identical to their counterparts in the last three subsections except for the initialization of ?posConstr_condition and ?posConstr_conclusion since now they are related by a cstr:descriptive_constraint_conclusion relation or a cstr:prescriptive_constraint_conclusion relation. E.g., here is a query for checking subclassof-analogous positive descriptive constraints. See the line in italics for the new initialization.

```
SELECT ?objectNotMatchingPosConstr ?posConstr_condition
          ?posConstr_conclusion WHERE
{ ?posConstr_condition
       cstr:descriptive_constraint_conclusion ?posConstr_conclusion.
  FILTER (?posConstr_conclusion != owl:Nothing)
  ?objectNotMatchingPosConstr rdf:type ?posConstr_condition.
  FILTER NOT EXISTS #no listing of objects satisfying the conclusion
  { ?objectNotMatchingPosConstr rdf:type ?posConstr_conclusion }
}
```

There are other ways to write the queries. For example:

- Instead of "FILTER (?posConstr_conclusion != owl:Nothing)", one may use "FILTER NOT EXISTS { ?posConstr_conclusion cstr:prescriptiveConclusion owl:Nothing }". The first way has the advantage of not being dependent on the chosen representation for constraint and hence this way minimizes the difference between the queries. On the other hand, with this way, owl:Nothing cannot be replaced by equivalent class expressions (in SPARQL).
- In Subsection D, the line "?objectNotMatchingPosConstr cstr:cloneWithoutType ?cloneWithoutType" before "FILTER NOT EXISTS" could be replaced by the line "BIND(uri(concat(str(?objectNotMatchingPosConstr), "_cloneWithoutType")) as ?cloneWithoutType)" within the "FILTER NOT EXISTS" block.

G. Checking SubclassOf-based Constraints

The previous queries do not rely on inference engines to take into account the special meaning of CSTR classes. Hence, as explained in Section III.C.1, these queries *cannot* be adapted for checking *subclassOf-based* constraints representing positive *descriptive* constraints. For *prescriptive* constraints, the queries are the same as their counterparts in the last four subsections except for the initialization of ?posConstr_condition and ?posConstr_conclusion. E.g., for a positive rescriptive constraint, this initialization now is:

```
?posConstr_condition
   rdfs:subClassOf cstr:SubclassOf-based_prescriptive_constraint_condition,
                   ?posConstr_conclusion.
```

Except as a module for calculating the completeness degree of a KB, individual-based inconsistency-implying constraints are useless if, when building the KB, its consistency is already checked by an inference engine that delivers an error message when detecting that an object is instance of a subclass of owl:Nothing. By default, some Description Logic inference engines such as Corese [11] do not deliver error messages or warning messages when detecting such objects. Having to make inferences on instances of a subclass of owl:Nothing also makes Corese behaves abnormally, e.g., not listing such instances as results of the previous described queries when these instances violate positive constraints.

H. Checking Binary Relations Instead of Individuals

To list *binary relations* violating constraints – instead of *individuals* that have some relations violating constraints – it is sufficient to replace rdf:type by the "logical implication relation between statements" in the previous content-independent queries that check positive constraints. For referring to such relations, Tim Berners-Lee uses the type name log:implies [13] in his Notation3 KRL. However, for this replacement to work, the used SPARQL engine must exploit an inference engine that can deduce the existence of such a relation when it exists

between the matched statements. Description Logic inference engines generally do not do so.

Like queries on individuals, queries on relations can use additional filters. E.g., for the *last query* of Subsection G to operate *only* on negative facts, one may add at the end of its *body*:

```
?objectMatchingNegConstr  rdf:type owl:NegativePropertyAssertion.
```

I. Evaluating the Completeness of a KB

A simple way to define or calculate a completeness degree for a KB is to divide "the number of relations (in the KB) that do not violate prescriptive constraints" by "the total number of relations". Another completeness degree may be obtained by dividing "the number of individuals that do not violate prescriptive constraints" by "the total number of individuals". The next query implements a variant of this last definition: instead of individuals, it exploits "objects that are source of at least one relation to another object". Furthermore, this query assumes that the constraints are represented as individual-based constraints. This query can be adapted to implement the above first definition via the method given in Subsection H. Similarly, descriptive constraints could also be taken into account.

```
SELECT ( ((?nbObjs - ?nbAgainstPosCs - ?nbMatchingNegCs) / ?nbObjs)
            AS ?completeness )
{ { SELECT ( COUNT(DISTINCT ?o) AS ?nbObjs )
    WHERE { ?o ?r ?o2 } } #any object source of a relation to another object
     # For considering only objects that have a type:
     #         { {?o rdf:type ?t1} UNION {?o cstr:type ?t2}
  }
  { SELECT ( COUNT(DISTINCT ?objectNotMatchingPosConstr)
                AS ?nbAgainstPosCs ) WHERE
    {  ... #the body of a query checking an individual-based positive
         #  prescriptive constraint (see Section IV.D) must be copied here

       #if ?objectNotMatchingPosConstr also violates a negative constraint
       #  it must not be counted here (otherwise it would be counted twice),
       FILTER NOT EXISTS       #  hence this code here
       { ?negConstr  cstr:condition_class  ?negConstr_condition;
                     cstr:conclusion_class  owl:Nothing.
         ?objectNotMatchingPosConstr rdf:type ?negConstr_condition
       }
    }
  }
  { SELECT ( COUNT(DISTINCT ?objectMatchingNegConstr)
                AS ?nbMatchingNegCs) WHERE
    { ?negConstr  cstr:condition_class ?negConstr_condition;
                  cstr:conclusion_class owl:Nothing.
      ?objectMatchingNegConstr rdf:type ?negConstr_condition
    }
  }
}
```

V. Applications, Evaluation and Comparisons

A. Examples of Applications or Use Cases

For designing subtype hierarchies, various research works such as [14] advise the use of tree structures. However, [4] shows that "systematically using subtype partitions (except for non-natural types)" instead of tree structures has the same benefits with less disadvantages. It also shows that following this ontology design pattern (ODP) means using representations equivalent to relations of type sub:nonNaturalOrPartitionSubclass (or a subtype of it; the prefix "sub:" is an abbreviation for the namespace http://www.webkb.org/kb/it/SUB). Using *only* OWL2, [4] fully defines this type as well as the other types necessary for stating a descriptive constraint for checking that the above cited ODP is systematically followed. Here is this constraint; it states that "if a class C1 has a subclass relation, all subclass relations from C1 must be of type sub:nonNaturalOrPartitionSubclass":

```
[] rdf:type  cstr:Descriptive_constraint;
   cstr:condition_class          # if C is a class that has a subclass ...
       [rdf:type owl:Class;   owl:equivalentClass
           [rdf:type owl:Restriction;   #"any class that has a subclass"
            owl:onProperty sub:subclass;  owl:someValuesFrom rdfs:Class] ];
   cstr:conclusion_class     # ... then C has  no subclass relation that is
                             #       not of type nonNaturalOrPartitionSubclass
       sub:ClassWithNoRel_subclassButNot-nonNaturalOrPartitionSubclass.
```

This constraint can also be represented in inconsistency-implying form, again only using types defined in OWL2:

```
sub:ClassWithSomeRel_subclassButNot-nonNaturalOrPartitionSubclass
    rdfs:subClassOf  owl:Nothing.
```

Reference [4] then generalizes this constraint (and the types it exploits) for checking all types of transitive relationships. More precisely, it proposes a SPARQL INSERT request that generates a *descriptive* constraint for each transitive relation type having a sub:nonNaturalOrPartitionTrRelType relation indicating which relation types *must* actually be used. E.g., this may be used to express that, instead of relations of type sub:subclass or sub:part, relations of type sub:nonNaturalOrPartitionSubclass or sub:partitionPart must respectively be used.

Similarly, [4] also proposes a SPARQL command which, for each instance of the type sub:MandatoryOutRelationType, generates *prescriptive* constraints for checking the systematic use of certain relation types. E.g., based on the following specification in the KB, the command generates a constraint indicating that every dividable object – i.e. every instance of sub:DividableThing – must be the source of a sub:part relation *except* for each object marked as an instance of sub:PartDestLeaf.

```
sub:part  rdf:type  sub:MandatoryOutRelationType;
          sub:leafObjectType  sub:PartDestLeaf;
          rdfs:domain  sub:DividableThing.
```

B. Evaluation and Comparisons

The originality of the approach proposed in this article is that it enables i) the representation of constraints independently of their exploitation (this one is represented within content-independent queries), ii) the representation of both descriptive and prescriptive constraints with any KRL the expressiveness of which is at least equal to RDFS, and hence iii) the exploitation of most inference engines, especially via SPARQL queries.

Since the proposed approach relies on other methods and tools chosen by each user of the approach, it inherits from their theoretical or practical improvements. It would thus not be relevant to focus on theoretical aspects of a particular method or tool in this article. For a general comparison, [12] and [15] list theoretical points relevant to the proposed approach. Regarding the use of SPARQL to check constraints, [10] shows that SPARQL can be used for both expressing and validating integrity constraints based on some *partial forms* of the Unique Name Assumption and Closed World Assumption. It also shows that this validation is sound and complete when the expressiveness used for the constraints and the rest of the KB are respectively "SROIQ and SRI" or "SROI and SROIQ". In the proposed approach, queries are used only for validating constraints, not expressing them, but this is only a generalization of the approach of [10] which does not change the associated

theoretical results. In [10], the used partial forms of the Unique Name Assumption and Closed World Assumption are specified in SPARQL via its operators EXISTS and NOT EXISTS plus the use of relations of type owl:sameAs or owl:differentFrom. These forms can similarly be expressed via the commands seen in Section IV and the use of relations of type owl:sameAs or owl:differentFrom in the constraints.

The proposed approach was validated experimentally by testing the degree to which a few constraints – including all those introduced in this article – were followed in i) the "family relationship" focused sample ontology given by the OWL2 Primer W3C document [16] and ii) a few ontologies from the Linked Data repository LOV. The validation came from finding the right constraint violations and completeness degrees via the proposed queries and, when necessary, KB pre-treatments.

Besides testing these constraints, queries and requests, another goal of this validation phase was to represent ontology design patterns or best practices (ODPs) as constraints. ODPs, e.g. those recommended by the W3C [17] or those of the "ODP catalog" [18], are i) informal descriptions about how certain things should be represented, and/or ii) collections of types that *should be reused whenever possible*, or iii) lexical or syntactic rules to follow when importing or exporting formal or informal knowledge. Descriptive or prescriptive constraints are ways to represent "*must be reused whenever possible*" and hence ways to formalize and implement ODPs related to the second point. However, during the validation phase, no ODP satisfying the two following criteria was found: i) the ODP could be implemented via a constraint, *and* ii) the ODP was likely not to lead to a completeness degree close to 0% for a randomly chosen ontology. More generally, no widely followed ODP was found.

Querying a KB for detecting *anti-patterns* in it is analogous to querying it for detecting violations of ODPs in it. However, like the SPARQL based works of [19], many works on anti-pattern detection use queries essentially as a way not to use an expressive inference engine for detecting certain problems. Instead, the proposed approach exploits inference engines. With a sufficiently powerful KRL, *any* anti-pattern can be expressed as a negative constraint in inconsistency-implying form.

The introduction of this article summarized the strong distinction that exists between *constraint-based completeness* and *represented-world-based completeness*, and hence the reason why it would not be relevant to further compare the proposed approach with those of tools such as SWIQA and Sieve.

Since the proposed approach is based on a particular use of RDFS it should be compared to SHACL and SPIN.

- SHACL (SHApes Constraint Language) is a language ontology (such as OWL2) proposed by the W3C to enable the definition of constraints in RDF. SHACL does not reuse OWL2 to define constraints: it introduces new terms. It therefore does not support the reuse – for checking constraints – of inference engines that take into account the special meaning of OWL2 terms. Thus, inference engines dedicated to SHACL have to be used and a new KRL (SHACL) has to be learned. In addition, SHACL does not distinguish between descriptive constraints and prescriptive ones, and thus handles prescriptive constraints only very partially. E.g., handling the condition and conclusion of a prescriptive constraint generally require different *entailment regimes* (as explained in Section II.B) but with SHACL, only one regime can be specified for both the condition and conclusion. Furthermore, neither LDScript-like extensions nor SPARQL update requests can be used in SHACL. Thus, pre-treatments of the KB – including the one proposed in Section IV.B for prescriptive constraints – have to be specified via a KRL other than SHACL.

- SPIN (SParql Inferencing Notation) is a W3C language ontology that enables the storage of SPARQL queries in RDF and, via special relations such as spin:rule and spin:constraint, the (possibly recursive) calls of SPARQL queries or Javascript functions for adding nodes or values to the KB. Thus, SPIN enables procedural attachments in a KB and thereby also supports the extension of SPARQL. However, the use of SPIN requires a SPIN aware engine. The approach proposed in this article is KRL independent (hence not based on procedural attachments). SPIN could be used for storing the SPARQL commands proposed in Section IV and Section V.A, thus not only procedurally defining the types proposed for constraints but also providing a way to trigger such commands automatically. SPIN can also be used for checking constraints in other ways that are less modular (i.e., not using content-independent queries) or less logic-based (i.e., more procedural), hence in ways that offer less possibilities for knowledge comparison, translation, inferencing, reuse or exploitation. The widespread use of such other ways may be a reason why SHACL has been designed. This article provides a less restricted alternative. The author also works on a knowledge translation tool exploiting ontology based specifications of conversions, including for constraints.

Some *transformation languages or systems exploit KRs*. Such systems are presented in [20] and [21]. Although few of them *explicitly* have a function that detects KR patterns without also transforming the matched KRs (PatOMat [20] is an exception), these languages or systems could easily be adapted to have such a function and hence be used for handling prescriptive constraints. However, all such systems appear to use rule-based languages with more expressiveness than what relations-between-classes based constraints allow. Typically, these languages allow the direct and explicit use of variables for relating objects shared by both the condition and conclusion of a rule. Indeed, using such languages can simplify the writing of prescriptive constraints. However, regarding *what* can be expressed and checked via constraints, this article shows that i) much can be achieved *simply using relations-between-classes based constraints and SPARQL1.1*, and ii) the power of the proposed approach then relies on the power of the inference engine used for object matching, rather than on the used language.

Some transformation systems, like PatOMat [20], issue SPARQL queries for detecting patterns, based on non-SPARQL specifications for *patterns and their transformations*. Some other transformation systems directly propose an extension of SPARQL such as STTL [21] to write specifications for patterns and their transformations. For instance, as shown in [22], STTL can be combined with LDScript [11] to specify STTL queries (compiled into SPARQL queries) for detecting patterns and then transforming the results. However, [22] does not discuss the exploitation of object matching capabilities of inference engines and it does not distinguish between prescriptive constraints and non-prescriptive ones. To sum up, the SPARQL commands

introduced in this article could also be reused in these transformation systems, although in an adapted form.

VI. CONCLUSION

This goal of this article – supporting constraint checking via few predefined *content-independent* queries, in a KRL independent and tool independent way – is original, useful for modularity as well as knowledge and tool reuse purposes, and applicable to various research fields. E.g., this support can help checking the following of ontology design patterns (ODPs), KB design libraries (e.g., the KADS library) or top-level ontologies (e.g., DOLCE) in order to validate the quality of a KB or, during its design, help elicit knowledge from experts.

The sections II and III answer the first two research questions: what kinds of constraints need to be considered for evaluating *constraint-based completeness*, and how to represent constraints in any KRL that has an expressiveness at least equal to RDF or RDFS? Section II and III do so via complementary means.

- First, by defining the original notion of "prescriptive constraint" for checking that some objects are *explicitly* given instead of possibly inferred as in descriptive constraints (the two constraint kinds thus form a partition).
- Second, by providing i) a general method to check prescriptive constraints, ii) types for distinguishing different kinds of constraints, and iii) three alternative structures for representing them via class expressions. The use of such expressions is both a way to permit the reuse of most KRLs and a way to reuse inference engines by exploiting calculated instanceOf relations
- Third, by showing that both descriptive constraints and prescriptive constraints are i) necessary for evaluating constraint-based completeness via content-independent queries, and ii) in a sense, sufficient too for two reasons. First, descriptive constraints and prescriptive constraints form a (complete) partition. Second, more specialized distinctions, if needed, can *still* be expressed by specializing the given types and using further methods to take into account these more specialized types.

Section IV answers the third research question: how to implement the general approach with query languages such as SPARQL or slight extensions of it? Section V.A shows how some ODPs can be represented as descriptive constraints exploitable by content-independent queries. Both sections highlight the use of KB pre-treatments to counter-balance certain lack of expressiveness of some languages, e.g. for implementing inference bypassing methods or generating constraints.

Section V.B evaluates the proposed techniques and compares the approach to other ones. *A complement to this work will be to* i) represent ODPs in several research areas (knowledge sharing, cooperation, security, etc.), using only relations between classes whenever possible, ii) organize them by relations of specialization or other kinds, and iii) test these ODPs via LDScript or more expressive languages.

ACKNOWLEDGMENT

Many thanks to Dr Olivier Corby (member of the Wimmics and SPARKS teams of, respectively, the INRIA and I3S CNRS laboratories at the University Côte d'Azur, France) for his questions and remarks on the approaches presented in this article and his help during the implementation of these approaches with SPARQL and LDScript via the Corese tool.

REFERENCES

[1] J. sowa, "conceptual graphs summary. conceptual structures: current research and practice," Ellis Horwood, pp. 3–51, 1992.

[2] A. Zaveri, A. A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, and S. Auer, "Quality assessment for linked data: a survey," Semantic Web, vol. 7(1), pp. 63–93, 2016.

[3] L. Galárraga, K. Hose, and S. Razniewski, "Enabling completeness-aware querying in SPARQL," WebDB 2017, pp. 19–22, USA, 2017.

[4] P. Martin, "Relations-between-classes based constraints and constraint-based ontology-completeness," Companion Web site for this article, http://www.webkb.org/kb/it/o_knowledge/d_constraints.html, 2018.

[5] M. Genesereth, and R. Fikes , "Knowledge interchange format, version 3.0, reference manual," Report Logic 92-1, Logic Group, Stanford Uni. http://www.ksl.stanford.edu/pub/KSL_Reports/KSL-92-86.ps.gz, 1992.

[6] M. Chein, and M. Mugnier, "The BG family: facts, rules and constraints," Graph-based Knowledge Representation - Computational Foundations of Conceptual Graphs. Chapter 11 (pp. 311–334), Springer-Verlag London, 428p., 2008.

[7] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "EQL-lite: effective first-order query processing in description logics," IJCAI 2007, pp. 274–279, India.

[8] U. Assmann, and G. Wagner, "Ontologies, metamodels and model-driven paradigm," Ontologies for Software Engineering and Technology, Springer-Verlag, Berlin, chapter 9, 2006.

[9] W3C Recommendation 2013, "SPARQL 1.1 entailment regimes," http://www.w3.org/TR/sparql11-entailment/

[10] J. Tao, E. Sirin, J. Bao, and D. McGuinness, "Integrity constraints in OWL," AAAI 2010, pp. 1443–1448, USA.

[11] O Corby, C. Faron-Zucker, and F. Gandon, "LDScript: a linked data script language," ISWC 2017, Austria.

[12] J. Baget, A. Gutierrez, M. Leclère, M. Mugnier, S. Rocher, and C. Sipieter, "Datalog+, RuleML and OWL 2: formats and translations for existential rules," Challenge+DC@RuleML 2015, 9th International Web Rule Symposium (RuleML), Germany.

[13] T. Berners-lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler, "N3logic: a logical framework for the world wide web," Journal Theory and Practice of Logic Programming, vol. 8(3), pp. 249–269, Cambridge University Press New York, USA, 2008.

[14] A. Rector, S. Brandt, N. Drummond, M. Horridge, C. Pulestin, and R. Stevens, "Engineering use cases for modular development of ontologies in OWL," Applied Ontology, vol. 7(2), pp. 113–132, IOS Press, 2012.

[15] R. Swan, "Querying existential rule knowledge bases: decidability and complexity," PhD thesis (159p), University of Montpellier, 2016.

[16] W3C Recommendation 2012, "OWL 2 web ontology language primer (second edition)," http://www.w3.org/TR/owl2-primer/

[17] W3C Recommendation 2006, "Semantic web best practices and deployment working group," https://www.w3.org/2001/sw/BestPractices/

[18] ODP 2018, "Ontology design patterns . org (ODP)," http://ontologydesignpatterns.org/wiki/Main_Page

[19] C. Roussey, and A. Zamazal, "Antipattern detection: how to debug an ontology without a reasoner," WoDOOM 2013, International Workshop on Debugging Ontologies and Ontology Mappings, pp. 45–56, France.

[20] O. Zamazal, and V. Svátek, "PatOMat – versatile framework for pattern-based ontology transformation," Computing and Informatics, vol. 34(2), pp. 305–336, 2015.

[21] O. Corby, and C. Faron-Zucker, "STTL: a SPARQL-based transformation language for RDF," WEBIST 2015, 11th International Conference on Web Information Systems and Technologies, Portugal.

[22] O. Corby, C. Faron-Zucker, and R. Gazzotti, "Validating ontologies against OWL 2 profiles with the SPARQL template transformation language," RR 2016, 10th International Conference on Web Reasoning and Rule Systems, LNCS, vol. 9898, pp. 39–45, Springer, UK.