Ontology Design Rules Based On Comparability Via Particular Relations

Philippe A. Martin [1][0000-0002-6793-8760], Olivier Corby [2][0000-0001-6610-0969] and Catherine Faron Zucker [2][0000-0001-5959-5561]

[1] EA2525 LIM, ESIROI I.T., University of La Réunion, F-97490 Sainte Clotilde, France
+ adjunct researcher of the School of I.C.T. at Griffith University, Australia
Philippe.Martin@univ-reunion.fr

[2] Wimmics team, INRIA / Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France
olivier.corby@inria.fr, faron@i3s.unice.fr

Abstract. The difficulty of representing and organizing knowledge in reasonably complete ways raises at least two research questions: "how to check that particular relations are systematically used not just whenever possible but whenever relevant for knowledge providers?" and "how to extend best practices, ontology patterns or methodologies advocating the systematic use of particular relations and, at the same time, automatize the checking of compliance with these methods?". As an answer, this article proposes a generic "ontology design rule" (ODR). A general formulation of this generic ODR is: in a given KB, for each pair of knowledge base objects (types or individuals) of a given set chosen by the user of this ODR, there should be *either* statements connecting these objects by relations of particular given types *or* statements negating such relations. This article further specifies this ODR and shows its interests for subtype relations and other transitive relations, e.g. part relations and specialization relations with genus & differentia. This article shows how this ODR can be implemented via OWL and SPARQL, at least for common simple cases (and, generically, via an higher-order logic based language).

Keywords: Ontology Design Patterns, Ontology Completeness, OWL, SPARQL.

1 Introduction

Representing and organizing knowledge within or across knowledge bases (KBs) is a fundamental and difficult task for knowledge sharing and inferencing, and thereby for knowledge retrieval and exploitation. At least three kinds of research avenues (relevant to refer to in this article) guide this task. The first are ontologies made for reuse purposes (with methodologies implicitly or explicitly based on these ontologies, e.g. the Ontoclean methodology): foundational ontologies such as DOLCE and BFO; task-oriented ones such as OWL-S; general ones such as DBpedia and Schema.org; domain-oriented ones such as those from BioPortal. The second are catalogs of best practices [1, 2] and ontology patterns [3, 4] or anti-patterns [5, 6]. The third are ontology/KB evaluation criteria and measures [7], e.g. for knowledge connectedness, precision, consistency, conciseness and completeness. The results of these three kinds of research avenues are especially helpful for building reusable ontologies.

These three kinds of research avenues advocate the use of relations of particular types between *objects* of particular types. In the RDF terminology, one would say that these three kinds of research avenues advocate the use of properties to connect *resources* of particular classes – e.g. the use of subClassOf or equivalentClass relations between classes or other objects, whenever this is relevant. *However*, often, only a knowledge provider knows when it is relevant to use a particular property. This limits the possibilities of checking or guiding the use of the advocated properties. *Furthermore*, it may also be useful that the knowledge provider represents when the advocated properties do not or cannot occur. E.g., representing disjointWith or complementOf relations between classes to express that subClassOf or equivalentClass relations cannot occur between these classes has many advantages that Section 2 illustrates. Using all these relations is especially useful between top-level classes since many inference engines can exploit the combination of these relations, e.g. via inheritance mechanisms. *Finally*, checking that particular relations are represented as either existing or forbidden can be done automatically. *Thus*, as an answer to the research questions "how to check that particular relations are systematically used not simply whenever this is possible but whenever this is relevant for the knowledge providers?" and "how to extend best practices, ontology patterns or methodologies that advocate the systematic use of particular relations, and make the compliance with these methods easier to check?", this article proposes the following *generic "ontology design rule" (ODR)*. A first general formulation of this generic ODR is: *in a given KB, for each pair of objects of a given set chosen by the user of this ODR, there should be either statements connecting these objects by relations of particular given types or statements negating such relations, i.e. expressing that these relations do not or cannot occur in the given KB*. A negated relation can be expressed directly via a negated statement or indirectly, e.g. via a disjointWith relation that forbids the existence of such a relation.

In its more precise version given in the next page, we call this ODR the "comparability via particular relation types" ODR, or simply the "comparability ODR". We call it an ODR, not a pattern nor a KB evaluation criteria/measure because this is something in between. As above explained, it is always automatically checkable. It is also reusable for evaluating a KB for example by applying it to all its objects and dividing the number of successful cases by the number of objects. An example of KB evaluation criteria that can be generalized by a reuse of this ODR is the "schema-based coverage" criteria of [1] which measures the percentage of objects using the relations that they should or could use according to schemas or relation signatures. Examples of methodologies, best practices or ontology patterns that can be generalized via the use of this ODR are those advocating the use of tree structures or of genus & differentia when organizing or defining types. (Section 2.3 and Section 3.3 detail this last point.)

Before formulating this ODR more precisely, it seems interesting to further detail its application to the OWL properties subClassOf or equivalentClass – along with those that negate or exclude them, e.g. disjointWith and complementOf. Using all these properties whenever relevant, as this applied ODR encourages, will for example lead the authors of a KB to organize the direct subtypes of each class – or at least each top-level class – into "complete sets of exclusive subtypes" (each of such sets being a subclass partition, or in other words, a disjoint union of subtypes equivalent to the subtyped class), and/or "incomplete sets of exclusive subtypes", and/or "(in-)complete sets of subtypes that are not exclusive but still different and not relatable by subClassOf

relations", etc. The more systematic the organization, the more a test of whether a class is *subClassOf_or_equivalent* (i.e. is subClassOf, equivalentClass or sameAs) another class will lead to a *true/false result, not an "unknown" result*. In other words, the more such a test will lead to a true/false result without the use of "negation as failure" (e.g. via the "closed-world assumption": any statement not represented in the KB is considered to be false) or the use of the "unique name assumption" (with which different identifiers are supposed to refer to different things). Since most inferences are based on such subClassOf_or_equivalent tests, the more systematic the organization, the more inferences will be possible without having to use negation as failure. This is interesting since using negation as failure implies making an assumption about the content of a KB whereas adding subClassOf or disjointWith relations means adding information to a KB.

The next two sections illustrate some of the many advantages of the more systematic organization resulting from the application of this ODR: for inferencing or querying, for avoiding what could have otherwise been implicit redundancies or inconsistencies and, more generally, for improving the completeness, consistency and precision of a KB. These advantages are not restricted to subClassOf_or_equivalent relations. They apply to all *specializationOf_or_equivalent* relations, i.e. specializationOf relations (they generalize subClassOf relations), equivalence relations or sameAs relations. As we shall see, most of these advantages also apply to other transitive relations such as *partOf_or_equivalent* (i.e. isSubPartOf, equivalentClass or sameAs). We call "specialization of an object" any other object that represents or refers to more information on the same referred object. This covers all subtype relations but also specialization relations between individuals, e.g. between the statements "some cars are red" and "John's car is dark red". We call "statement" a relation or a set of connected relations.

We adopt the following "comparability" related definitions. Two objects are "*comparable via a relation of a particular type*" (or, more concisely, "comparable via a particular relation type") if they are *either identical (sameAs), equivalent (by intension, not extension) or connected by a relation of this type*. Two objects are "uncomparable via a relation of a particular relation type" (or, more concisely, "uncomparable via a particular property") if they are different and if some statement in the KB forbids a relation of this type between these two objects. Given these definitions, the comparability ODR can be defined as testing whether "*each object (in the KB or a part of the KB selected by the user of this ODR) is defined as either comparable or uncomparable to each other object* (or *at least some object*, if the user prefers) *via each of the tested relation types*". In a nutshell, the comparability ODR checks that between particular selected objects there is "either a comparability or an uncomparability via particular relations". This ODR does not rely on particular kinds of KBs or inference engines but powerful engines may be relevant for checks if they infer more relations.

Stronger versions of this ODR can be used. E.g., for a more organized KB, some users may wish to have "either comparability or *strong* uncomparability" via relations of particular types between any two objects. Two objects are "strongly uncomparable via a relation of a particular type" if they are different and if some statement in the KB forbids the existence of a relation of this type between the two objects *as well as* between their specializations. E.g., disjoint classes are strongly uncomparable since they cannot have shared instances or shared subclasses (except for `owl:Nothing`).

A more general version of this ODR could also be defined by using "equivalence by intension or extension" instead of simply "equivalence by intension". In this article,

"equivalence" means "equivalence by intension" and "specialization" is also "specialization by intension". This article also assumes that the equivalence or specialization relations (or their negations) which are automatically detectable by the used inference engine are made explicit by KB authors and thus can be exploited via SPARQL queries. In a description logics based KB, this can be achieved by performing type classification and individual categorization before checking the ODR.

Figure 1 shows a simple graphic user interface for selecting various *options or variants* for this ODR. With the shown selected items (cf. the blue items in Figure 1 and the words in italics in the rest of this sentence), this interface generates a function call or query to check that each object (in the *default KB*) which is instance of *owl:Thing* is either *comparable or uncomparable* via *specialization* relations and *part* relations to *each* other object in the *default KB*. Figure 1 shows a function call. After the conversion of its last three parameters into more formal types, a similar call can be made to a generic function. [8] is an extended version and on-line companion article for this one. In its appendix, [8] defines this generic function and the types it exploits. To achieve this, these definitions are written in a higher-order logic based language.

With the *comparability_or_uncomparability* option (hence with the *comparability* ODR), equivalence or sameAs relations are always exploited in addition to the specified relations. When it is not relevant to also exploit equivalence or sameAs relations, the *connectability_or_un-connectability* option shown in Figure 1 should be selected.

The next two sections show the interests of this ODR for, respectively, i) subtype relations, and ii) other relations, e.g. part relations and specialization relations with genus & differentia. When relevant, these sections present type definitions in OWL, as well as SPARQL queries or update operations, to illustrate how this ODR can be implemented. Figure 1 shows that SHACL (a constraint language proposed by the W3C) may also be exploited when its expressiveness is sufficient to express the constraint that needs to be represented. However, this exploitation is not described in this article. Section 4 provides more comparisons with other works and concludes.

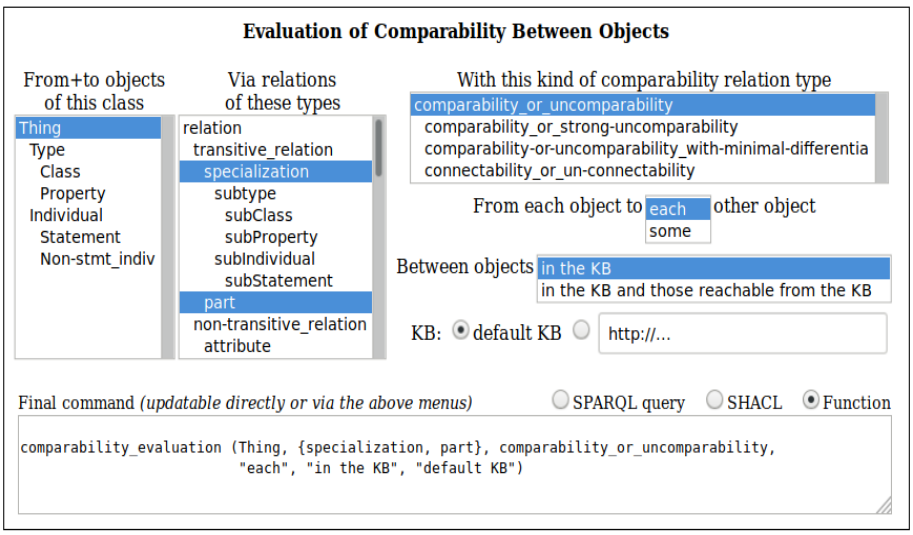

Fig. 1. A simple interface for object comparability/connectability evaluation

2 Comparability of Types Via Subtype Relations

2.1 Representation Via OWL

In this document, OWL refers to OWL-2 (OWL-2 DL or OWL-2 Full) [9] and OWL entities are prefixed by "`owl:`". All the types that we propose in this article are in `http://www.webkb.org/kb/it/o_knowledge/d_odr_content/sub/` and the "`sub`" namespace is here used to abbreviate this URL. Unless otherwise specified, the syntax used for defining these types is Turtle, and the syntax used for defining queries or update operations is SPARQL. SPARQL uses Turtle for representing relations. For clarity purposes, identifiers for relation types have a lowercase initial while other identifiers have an uppercase initial.

To illustrate the interest of representing exclusion relations between classes – and, more generally, of the interest of making types "uncomparable via subClassOf relations" whenever possible – here is an example in two parts. The first part is composed of the following RDF+OWL/Turtle statements. They do represent any exclusion relation. They represent a few relations from WordNet 1.3 (not the current one, WordNet 3.1). According to these relations, Waterloo is both a battle and a town, any battle is a (military) action, any town is a district, and any district is a location.

```
wn:Waterloo rdf:type wn:Battle, wn:Town.
wn:Battle rdfs:subClassOf wn:Military_action.
                          wn:Military_action rdfs:subClassOf wn:Action.
wn:Town rdfs:subClassOf wn:District.
                        wn:District rdfs:subClassOf wn:Location.
```

Now, as a second part of the example, a disjointWith relation is added between two top-level classes: the one for actions and the one for locations. This exclusion relation between actions and locations has not been made explicit in WordNet but is at least compatible with the informal definitions associated to categories in WordNet. *Given all these relations*, an OWL inference engine (that handles disjointWith relations) *detects* that the categorization of Waterloo as both a battle and a town is *inconsistent*. As illustrated in Section 2.3, many other possible problems in WordNet 1.3 were similarly detected. Most of them do not exist anymore in the current WordNet.

```
wn:Action owl:disjointWith wn:Location.
```

OWL DL is sufficient for representing statements implying that particular *classes* are "comparable via subClassOf (relations)" or "strongly uncomparable via subClassOf". For this second case, which amounts to state that two classes are disjoint, the properties `owl:AllDisjointClasses`, `owl:complementOf`, `owl:disjointWith` and `owl:disjointUnionOf` can be used. OWL Full [9] is necessary for setting `owl:differentFrom` relations between *classes*, and hence, as shown in the next page, for defining the property `sub:different_and_not_subClassOf` as a sub-property of `owl:differentFrom`. In turn, this property is necessary for representing statements implying that particular classes are *weakly uncomparable*, i.e. uncomparable but not strongly uncomparable (hence not disjointWith nor complementOf). OWL Full is also necessary for defining the properties `sub:different_and_not_equivalentClass` and `sub:proper-subClassOf` (alias, `sub:subClassOf_and_not-equivalentClass`). With all the above cited types, it is possible for KB authors to express any relationship of "comparability or uncomparability via subClassOf".

OWL inference engines generally cannot exploit OWL Full and hence do not enforce nor exploit the semantics of definitions requiring OWL Full. When inference engines do not accept OWL Full definitions, the above cited "sub:" properties have to be solely *declared* (as being properties) instead of being *defined* via relations (hence by a logic formula). However, when inference engines do not accept or do not exploit OWL Full definitions, the loss of inferencing possibilities due to the non-exploitation of the above cited "sub:" properties is often small. When the goal is simply to detect whether the comparability ODR is followed, if the SPARQL query proposed in the next subsection is used to achieve that goal, it does not matter whether the above cited "sub:" properties are declared or defined.

Making *every* pair of classes in a KB comparable or uncomparable via subClassOf is cumbersome without the use of properties that create (in-)complete sets of (exclusive) subclasses. We propose such properties, e.g. `sub:complete_set_of_uncomparable-subClasses`, `sub:incomplete_set_of_uncomparable-subClasses` and `sub:proper-superClassOf_uncomparable_with_its_siblings`. Such complex properties cannot be defined in OWL. However, as illustrated below, SPARQL update operations can be written to replace the use of these complex properties by the use of simpler properties that OWL inference engines can exploit.

```
sub:proper-subClassOf  rdfs:subPropertyOf  rdfs:subClassOf;
  owl:propertyDisjointWith owl:equivalentClass .
#a "proper subClass" is a "strict subClass" (a direct or indirect one)

sub:proper-subPropertyOf  rdfs:subPropertyOf  rdfs:subPropertyOf;
  owl:propertyDisjointWith owl:equivalentProperty .

sub:different_and_not_subClassOf  rdfs:subPropertyOf  owl:differentFrom;
  owl:propertyDisjointWith  rdfs:subClassOf .

sub:different_and_not_equivalentClass rdfs:subPropertyOf owl:differentFrom;
  owl:propertyDisjointWith  owl:equivalentClass .

sub:proper-superClassOf  owl:inverseOf  sub:proper-subClassOf .
sub:proper-superClassOf_uncomparable_with_its_siblings
  rdfs:subPropertyOf  sub:proper-superClassOf .  #partial definition only


#Example of a SPARQL update operation to replace the use of
# sub:proper-superClassOf_uncomparable_with_its_siblings relations
# by simpler relations:
DELETE
{ ?c sub:proper-superClassOf_uncomparable_with_its_siblings ?sc1, ?sc2 }
INSERT { ?c sub:proper-superClassOf ?sc1, ?sc2 .
         ?sc1 sub:different_and_not_subClassOf ?sc2 .
         ?sc2 sub:different_and_not_subClassOf ?sc1 }
WHERE{?c sub:proper-superClassOf_uncomparable_with_its_siblings ?sc1, ?sc2
      FILTER (?sc1 != ?sc2) }
```

Similarly, to state that particular properties are (strongly or at least weakly) "uncomparable via `rdfs:subPropertyOf` relations", OWL DL is sufficient. For strong uncomparability, `owl:propertyDisjointWith` relations can be used. Defining that particular properties are only *weakly uncomparable*, i.e. uncomparable but not strongly uncomparable, is possible in OWL Full, exactly as for subClassOf relations: to define these properties, it is sufficient to replace every occurence of "class" by "property" in the above code. As for classes too, if these "sub:" properties are only declared instead of being defined, the loss of inferencing possibilities is small.

2.2 Checking Via SPARQL

Using SPARQL (1.1) [10] to check the "comparability of classes via subClassOf relations" means finding each class that does not follow this ODR, i.e. that each class that is *neither* comparable *nor* uncomparable via subClassOf relations to *each/some other* class in selected KBs ("each/some" depending on what the user wishes to test).

The next page shows a SPARQL query for the "*each* other class" choice, followed by a SPARQL query for the "*some* other class" choice. In any case, if instead of the "comparability_or_uncomparability" option (the default option selected in Figure 1), the user prefers the "comparability_or_strong-uncomparability" option, the two lines about `sub:different_and_not_subClassOf` relations should be removed. For the "connectability_or_un-connectability" option, the line about `owl:equivalentClass` and `owl:sameAs` relations should instead be removed.

```
SELECT distinct ?c1 ?c2 WHERE   #query for the "each other class" choice
{ ?c1 a owl:Class.  ?c2 a owl:Class. FILTER(?c1 != ?c2)

  #skip comparable objects (here, classes comparable to ?c1):
  FILTER NOT EXISTS{ ?c1 rdfs:subClassOf|^rdfs:subClassOf ?c2 }
  FILTER NOT EXISTS{ ?c1 owl:equivalentClass|owl:sameAs ?c2 }

  #skip strongly uncomparable objects:
  FILTER NOT EXISTS{ ?c1 owl:complementOf|owl:disjointWith ?c2 }
  FILTER NOT EXISTS{ [] rdf:type owl:AllDisjointClasses;
                        owl:members/rdf:rest*/rdf:first ?c1,?c2 }
  FILTER NOT EXISTS{ [] owl:disjointUnionOf/rdf:rest*/rdf:first ?c1,?c2 }

  #skip remaining uncomparable objects that are only weakly uncomparable:
  FILTER NOT EXISTS { ?c1 owl:differentFrom ?c2 }
} #no need to use sub:different_and_not_subClassOf here since, at this
  #  point, subClassOf relations have already been filtered out


SELECT distinct ?c1 WHERE    #query for the "some other class" choice
{ ?c1 a owl:Class.  #for each class ?c1

  #skip comparable objects (here, classes comparable to ?c1):
  FILTER NOT EXISTS{?c1 rdfs:subClassOf|owl:equivalentClass|owl:sameAs ?c2
                    FILTER ((?c1!=?c2) && (?c2!=owl:Nothing)) }

  #skip strongly uncomparable objects:
  FILTER NOT EXISTS{ ?c1 owl:complementOf|owl:disjointWith ?c2
                         FILTER ((?c1!=?c2) && (?c2!=owl:Nothing)) }
  FILTER NOT EXISTS{ [] rdf:type owl:AllDisjointClasses;
                        owl:members/rdf:rest*/rdf:first ?c1,?c2 }
  FILTER NOT EXISTS{ [] owl:disjointUnionOf/rdf:rest*/rdf:first ?c1,?c2 }

  #skip remaining uncomparable objects that are only weakly uncomparable:
  FILTER NOT EXISTS { ?c1 owl:differentFrom ?c2 }
}
```

Checking the "comparability of properties via subPropertyOf relations" is similar to checking the "comparability of classes via subClassOf relations". The above SPARQL query can easily be adapted. The first adaptation to make is to replace every occurence of "class" by "property", to replace "disjointWith" by "propertyDisjointWith" and to replace "complementOf" by "inverseOf". The second adaptation to make is to remove the lines about "AllDisjointClasses" and "disjointUnionOf" since in OWL these types do not apply to properties and have no counterpart for properties.

Dealing with several datasets. A KB may reuse objects defined in other KBs; object identifiers *may* be URIs which refer to KBs where more definitions on these objects can be found. We abbreviate this by saying that these other KBs or definitions are reachable from the original KB. Similarly, from this other KB, yet other KBs can be reached. One feature proposed in Figure 1 is to check all objects "in the KB and those *reachable* from the KB". Since comparability checking supports the detection of particular inconsistencies and redundancies (cf. next subsection and next section), the above cited feature leads to the checking that a KB does not have particular inconsistencies or redundancies with the KBs reachable from it. This feature does not imply *fully* checking these other KBs. The above presented SPARQL query does not support this feature since it checks classes in the dataset of a *single* SPARQL endpoint. Implementing this feature via SPARQL while still benefiting from OWL inferences unfortunately requires the SPARQL engine and the exploited OWL inference engine to work on a merge of all datasets reachable from the originally queried dataset. For small datasets, one way to achieve this could be to perform such a merge beforehand via SPARQL insert operations. However, when it is not problematic to give up OWL inferences based on knowledge from other datasets, an alternative is to use a SPARQL query where i) "SPARQL services" are used for accessing objects in other datasets, and ii) transitive properties such as rdfs:subClassOf are replaced by property path expressions such as "`rdfs:subClassOf+`".

2.3 Advantages For Reducing Implicit Redundancies, Detecting Inconsistencies and Increasing Knowledge Querying Possibilities

Within or across KBs, hierarchies of types (classes or properties) may be *at least partially redundant*, i.e. they could be at least partially derived from one another if particular type definitions or transformation rules were given. Implicitly redundant type hierarchies, i.e. non-automatically detectable redundancies between type hierarchies, are reduced and easier to merge (manually or automatically) when types are related by *subtypeOf_or_equivalent* relations, e.g. subClassOf, subPropertyOf, equivalentClass or equivalentProperty relations. Using such relations is also a cheap and efficient way of specifying the semantics of types.

Relating types by *not_subtypeOf-or-equivalent* relations – e.g. disjointWith or complementOf relations – permits the detection or prevention of incorrect uses of such relations and of instanceOf relations. These incorrect uses are generally due to someone not knowing some particular semantics of a type, because this someone forgot this semantics or because this semantics was never made explicit. The two-point list below gives some examples extracted from [11]. In this article, the author – who is also the first author of the present article – reports on the way he converted the noun related part of WordNet 1.3 into an ontology. Unlike for other such conversions, the goal was to avoid modifying the meanings the conceptual categories of WordNet as specified by their associated informal definitions and informal terms. The author reports that, after adding disjointWith relations between top-level *conceptual categories* which according to their informal definitions seemed exclusive, his tool automatically detected 230 violations of these exclusions by lower-level categories. In the case of WordNet, what these violations mean is debatable since it is not an ontology. However, like all such violations, they can at least be seen as heuristics for bringing more precision and structure when building a KB. The authors of WordNet 1.3 were sent the list of the 230 detected possible problems. Most of these possible problems do not occur anymore in the current WordNet (3.1).

- Many of the 230 possible problems were detected via the added exclusion relations between *the top-level category for actions* and other top-level categories which seemed exclusive with it, based on their names, their informal definitions and those of their specializations. Via the expression "informal definition" we refer to the description in natural language that each WordNet category has. Via the expression "categorized as" we refer to the generalization relations that an object has in WordNet. The above mentioned added exclusion relations led to the discovery of categories – e.g. those for some of the meanings of the words "epilogue" and "interpretation" – which were i) categorized and informally defined as action results/attributes/descriptions, ii) seemingly exclusive with actions (given how they were informally defined and given they were not also informally defined as actions), and iii) (rather surprisingly) *also* categorized as actions. Given these last three points, [11] removed the "categorization as action" of these action result/attribute/description categories. Based on the content of WordNet 3.1, it appears that the authors of WordNet then also made this removal.
- Other causes for the 230 violations detected via the added exclusion relations between top-level categories came from the fact that WordNet uses *generalization relations* between categories instead of other relations. E.g., instead of location/place relations: in WordNet 1.3, many categories informally defined as battles were classified as both battles and cities/regions (this is no more the case in WordNet 3.1). E.g., instead of member relations: in WordNet, the classification of species is often intertwined with the classification of genus of species.

Several research works in knowledge acquisition, model-driven engineering or ontology engineering, e.g. [12–15], have advocated the use of *tree* structures when designing a subtype hierarchy, hence the use of i) single inheritance only, and ii) multiple tree structures, e.g. one per *view* or *viewpoint*. They argue that each object of the KB has a *unique place* in such trees and thus that such trees can be used as decision trees or ways to avoid redundancies, normalize KBs and ease KB searching/handling. This is true but the same advantages can be obtained by creating subtypes *solely via sets of disjoint (direct) subtypes*. Indeed, to keep these advantages, it is sufficient (and necessary) that *whenever* two types are disjoint, this disjointness is specified. With tree structures, there are no explicit disjointWith relations but the disjointness is still (implicitly) specified. Compared to the use of multiple tree structures, the use of disjoint subtypes and multiple inheritance has the advantages of i) not requiring a special inference engine to handle "tree structures with bridges between them" (e.g. those of [12, 16]) instead of a classic ontology, and ii) generally requiring less work for knowledge providers than creating and managing many tree structures with bridges between them. Furthermore, when subtype partitions can be used, the completeness of these sets supports additional inferences for checking or reasoning purposes. The above rationale do not imply that views or tree structures are not interesting, they only imply that sets of disjoint subtypes are good alternatives when they can be used instead.

Methods or patterns to fix (particular kinds of) detected conflicts are not within the scope of this article. Such methods are for example studied in the belief set/base revision/contraction as well as in KB debugging. [17] proposes an adaptation of base revision/debugging for OWL-like KBs. The authors of [18] have created ontology design patterns that propose systematic ways to resolve fix some particular kinds of inconsistencies, especially the violation of exclusion relations.

As illustrated in Section 2.1, the OWL properties usable to express that some types are “comparable or uncomparable via subtypeOf” – e.g. subClassOf, subPropertyOf, equivalentClass, equivalentProperty, disjointWith and complementOf relations – can be combined to define or declare properties for creating (un-)complete sets of (non-)disjoint subtypes or, more generally, for creating more precise relations which better support the detection of inconsistencies or redundancies. E.g., `sub:properSubClassOf` can be defined and used to prevent unintended subClassOf cycles.

Advantages For Knowledge Querying. Alone, subtypeOf_or_equivalent relations only support the search for specializations (or generalizations) of a query statement, i.e. the search for objects comparable (via subtype relations) to the query parameter. The search for objects “not uncomparable via specialization” to the query parameter – i.e. objects that are or could be specializations or generalizations of this parameter – is more general and sometimes useful.

- Assume that a KB user is searching for lodging descriptions in a KB where sports halls are not categorized as lodgings but are not exclusive with them either, based on the fact that they are not regular lodgings but that they can be used as such when natural disasters occurs. Also assume that the user intuitively shares such views on lodgings and sports halls. Then, querying the KB for (specializations of) “lodgings” will not retrieve sports halls. On the other hand, querying for objects not uncomparable to “lodgings” will return sports halls; furthermore, if lodgings have been defined as covered areas, such a query will not return uncovered areas such as open stadiums. Thus, assuming that the term "lodging" in this previous querying has been used because the author of the query was looking for covered areas only, this person will only get potentially relevant results.
- More generally, when a person does not know which exact type to use in a query or does not know what kind of query to use – e.g. a query for the specializations or the generalizations of the query parameter – a query for objects “not uncomparable” to the query parameter may well collect all and only the objects the person is interested in, if in the KB all or most types are either comparable or uncomparable via subtype relations.

The more systematically the types of a KB are comparable via subtype relations, the more the statements of the KB – as well as other if they have a definition – will be retrievable via comparability or uncomparability based queries.

3 Other Interesting Cases of Comparability

The previous section was about the comparability of types via subtype relations. This subsection generalizes the approach to other types of relations.

3.1 Comparability Via “Definition Element” Relations

In this article, an object definition is a logic formula that all specializations of the object must satisfy. A full definition specifies necessary and sufficient conditions that the specializations must satisfy. In OWL, a full definition of a class is made by relating this class to a class expression via an `owl:equivalentClass` relation. Specifying only necessary conditions – e.g. using `rdfs:subClassOf` instead of `owl:equivalentClass` – means making only a partial definition. An “element of a definition” is any target

domain object which is member of that definition, except for objects of the used language (e.g. quantifiers and logical operators). A "definition element" relation is one that connects the defined object to an element of the definition. E.g., if a *Triangle* is defined as a "Polygon that has as part 3 Edges and 3 Vertices", *Triangle* has as *definition elements* the types *Polygon*, *Edge*, *Vertex* and *part* as well as the value *3*. The property `sub:definition_element` – one of the types that we propose – is the type of all "definition element" relations that can occur with OWL-based definitions. We have fully defined `sub:definition_element` in [8] based on the various ways definitions can be made in OWL; one of its subtypes is `rdfs:subClassOf`. This subsection generalizes Section 2 since a definition may specify other relations than subClassOf relations, as illustrated by the above definition of Triangle. A "definition-element exclusion" relation is one that connects an object O to another one that could not be used for defining O. This property can be defined based on the "definition element" relation type. E.g.:

```
sub:definition-element_exclusion   #reminder: "has_" is implicit
  rdfs:subPropertyOf  owl:differentFrom ;
  owl:propertyDisjointWith  sub:definition_element ;
  owl:propertyDisjointWith [owl:inverseOf sub:definition_element].
```

As explained in Section 2.3, checking that types in a KB are either comparable or uncomparable via subtype relations *reduce* implicit redundancies between type hierarchies. As illustrated by the later paragraph titled "Example of implicit potential redundancies", this checking is not sufficient for finding *every* implicit potential redundancy resulting from a lack of definition, hence for finding every specialization hierarchy that could be derived from another one in the KB if particular definitions were given. However, this new goal can be achieved by generalizing the previous approach since this goal implies that for every pair of objects (in the KB or a selected KB subset), *either* one of these objects is defined using the other *or* none can be defined using the other. In other words, this goal means checking that for every pair of objects in the selected set, these two objects are either comparable or uncomparable via "definition element" relations. To express that objects are strongly uncomparable in this way – and hence *not* potentially redundant – "definition-element exclusion" relations can be used.

The above cited new goal implies that, from every object, *every other object* in the KB is made comparable or uncomparable via "definition element" relations. This is an enormous job for a KB author and very few current KBs would satisfy this ODR. However, given particular reasons and techniques described in [8], a KB contributor/evaluator *may* choose to assume that for avoiding *a good enough amount of* implicit potential redundancies between type hierarchies, it is sufficient to check that from every object, *at least one other object* in the KB is made comparable or uncomparable via "definition element" relations (thus, using the "some other object" option given in Figure 1, instead of the "every other object" option). As explained in [8], this saves a lot of work to the KB contributors and may avoid generating a large number of "definition-element exclusion" relations.

Example of implicit potential redundancies. It is often tempting to specialize particular types of processes or types of physical entities according to particular types of attributes, without explicitly declaring these types of attributes and organizing them by specialization relations. E.g., at first thought, it may sound reasonable to declare a process type Fair_process without relating it to an attribute type Fairness (or Fair) via a definition such as "any Fair_process has as attribute a Fairness". However, Fair_process

may then be specialized by types such as Fair_process_for_utilitarianism, Fair_process_wrt_Pareto-efficiency, Fair_bargaining, Fair_distribution, Fair_distribution_wrt_utilitarianism, Fair_distribution_for_prioritarianism, Fair_distribution_wrt_Pareto-efficiency, etc. It soon becomes apparent that this approach is not relevant since i) every process type can be specialized wrt. a particular attribute type or any combination of particular attribute types, and ii) similar specializations can *also* be made for function types (e.g. starting from Fair_function) and attribute types (starting from Fairness). Even if the KB is not a large KB shared by many persons, many beginnings of such parallel categorizations may happen, without them being related via definitions. Indeed, the above example with process types and attribute relations to attributes types can be replicated with any type and any relation type, e.g. with process types and agent/object/instrument/time relation types or with physical entity types and mass/color/age/place relation types.

Ensuring that objects are either comparable or uncomparable via "definition element" relations is a way to prevent such (beginnings of) implicitly potentially redundant type hierarchies: all/most/many of them depending on the chosen option and assumption. As with disjointWith relations, the most useful "definition-element exclusion" relations are those between some top-level types. To normalize definitions in the KB, e.g. to ease logical inferencing, a KB owner may also use "definition-element exclusion" relations to forbid particular kinds of definitions, e.g. forbid processes to be defined wrt. attributes or physical entities. Each definition for a type T sets "definition element" relations to *other types*, and these relations also apply to the subtypes of T. A special "definition element" relation type may also be used to reach not just the above cited *other types* but their subtypes too. Otherwise, most types would need to be defined if few "definition-element exclusion" relations are set between top-level types.

3.2 Comparability Via Other Transitive Relations, Especially Part Relations

Ensuring that objects are either comparable or uncomparable via specialization relations via specialization relations has many advantages which were illustrated in Section 2.3 and Section 3.1. Similar advantages exist with all transitive relations, not just specialization relations, although to a lesser extent since less inferences – and hence less error detection – can be made with other transitive relations.

Part properties – e.g. for spatial parts, temporal parts or sub-processes – are *partial-order* properties that are often exploited. Unlike subtype relations, they connect individuals. Nevertheless, for checking the "comparability of individuals via part relations (let us assume `sub:part` relations)", the SPARQL query given in Section 2.2 can be adapted. Below is this adapted query for the "*each* other object" choice. The adaptation to make for the "*some* other object" choice is similar to the one in Section 2.2. Two objects that are "comparable via part relations" if one is fully part of the other (or if they are identical). They are "strongly uncomparable via part relations" if they do not share any part (and hence the respective parts of these two objects do not have shared parts either). Two objects that are "weakly uncomparable via part relations" share some parts but none is fully part of the other.

```
SELECT distinct ?i1 ?i2 WHERE #individuals (as checked by the next 2 lines)
{ ?i1 rdf:type ?c1.  FILTER NOT EXISTS { ?i1 rdf:type owl:Class }
  ?i2 rdf:type ?c2.  FILTER NOT EXISTS { ?i2 rdf:type owl:Class }

  #skip comparable objects:
  FILTER NOT EXISTS { ?i1 owl:sameAs|sub:part+|(^sub:part)+ ?i2 }
```

```
  #skip strongly uncomparable objects:
  FILTER NOT EXISTS { ?i1 sub:part_exclusion ?i2 }

  #skip remaining uncomparable objects that are only weakly uncomparable:
  FILTER NOT EXISTS { ?i1 owl:differentFrom ?i2 } #as in Section 2.2
} #with:    sub:part rdfs:subPropertyOf owl:differentFrom ;
  #                  rdf:type owl:TransitiveProperty .
  #         sub:part_exclusion  rdfs:subPropertyOf  owl:differentFrom ;
  #                             owl:propertyDisjointWith  sub:part .
```

3.3 Comparability Via Transitive Relations Plus Minimal Differentia

When defining a type, a good practice is to specify i) its similarities and differences with each of its direct supertypes (e.g., as in the genus & differentia design pattern), and ii) its similarities and differences with each of its siblings for these supertypes. This is an often advocated best practice to improve the understandability of a type, as well as enabling more inferences. E.g., this is the "Differential Semantics" methodology of [13]. Several ODRs can be derived from this best practice, depending on how "difference" is defined. In this article, the term "minimal-differentia" refers to a difference of at least one (inferred or not) relation in the compared type definitions: one more relation, one less or one with a type or destination that is different (semantically, not just syntactically). Furthermore, to check that a class is different from each of its superclasses (i.e. to extend the genus & differentia method), an `rdfs:subClassOf` relation between the two classes does not count as "differing relation". When relevant, this ODR can be generalized to use other transitive relations between objects, e.g. partOf relations.

For the "comparability relation type", Figure 1 proposes the option "comparability-or-uncomparability_with-minimal-differentia". For supporting this option when checking "comparability via subClassOf relations" between any pair of classes in a KB, the code of the SPARQL query of Section 2.2 can be adapted by adding some lines before the filters testing whether the classes are comparable or uncomparable: below, see the FILTER block from the 3rd line to the "...". This block checks that there is a "minimal-differentia" between the tested classes. The retrieval of automatically inferred relations relies on the use of a relevant entailment regime.

```
SELECT distinct ?c1 ?c2 WHERE
{ ?c1 rdf:type owl:Class.   ?c2 rdf:type owl:Class.  FILTER (?c1 != ?c2)

  FILTER (!            #skip classes satisfying the following conditions:
   ( (EXISTS
      { ?c1 ?p1 ?v1 .                #?c1 has at least one property
        FILTER(?p1!=rdfs:subClassOf) #        that is not rdfs:subClassOf
        FILTER                       #and
        ( NOT EXISTS { ?c2 ?p1 ?v2 } #   ?p1 is not in ?c2
          ||  EXISTS { ?c2 ?p1 ?v2   #or ?p1 ?v1 is not in ?c2
                       FILTER (?v1 != ?v2) }
          ||  EXISTS { ?c2 ?p2 ?v2   #or ?p2 ?v2 is not in ?c1
                       FILTER NOT EXISTS { ?c1 ?p2 ?v2 }  } )
      })
    || ((NOT EXISTS                  #or
         { ?c1 ?p1 ?v1               #  ?c1 has no property, except may be
           FILTER((?p1!=rdfs:subClassOf) #  an rdfs:subClassOf property
                  && (?v1 != ?c2))}      #  to ?c2
       ) && EXISTS{?c2 ?p2 ?v2})   #  and ?c2 has (other) properties
   ))
  ... #same filtering for (un-)comparable objects as in Section 2.2
}
```

4 Other Comparisons With Other Works and Conclusion

As previously illustrated, the "comparability ODR" generalizes – or permits one to generalize – some best practices, ontology patterns or methodologies that advocate the use of particular relations between particular objects, and supports an automated checking of the compliance with these practices, patterns or methodologies. This leads to the representation of knowledge that is more connected and precise, or with less redundancies. Since the comparability ODR can be used for evaluating a KB – e.g. by applying it to all its objects and dividing the number of successful cases by the number of objects – it can also be used to create KB evaluation criteria/measures, typically for measuring the *(degree of) completeness of a KB, with respect to some criteria*.

As noted in [7], a *survey* on quality assessment for Linked Data, *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, we distinguish two kinds of completeness.

- *Constraint-based completeness* measures the percentage of elements in a dataset that satisfy *explicit* representations of what – or how – information *must* be represented in the dataset. These representations are constraints such as integrity constraints or, more generally, constraints expressed by database schemas, structured document schemas, or schemas enforcing ontology design patterns. E.g., in a particular dataset, the constraint that at least one movie must be associated to each movie actor, or the constraint that all relations must be binary.
- *Real-world-based completeness* measures the degree to which particular kinds of real-world information are represented in the dataset. E.g., regarding movies associated to an actor, calculating this 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, *i.e. in the real world*". Either the missing information are found in a *gold standard dataset* or the degree is estimated via *completeness oracles* [17], i.e. rules or queries estimating what is missing in the dataset to answer a given query correctly. Tools such as SWIQA and Sieve help perform measures for this kind of completeness.

All the completeness criteria/measures collected by [7] – *schema/property/population/interlinking* completeness – "assume that a gold standard dataset is available". Hence, they are all subkinds of *real-world based completeness*. However, constraint-based completeness is equally interesting and, for its subkinds, categories named *schema/property/population/interlinking completeness* could also be used or have been used [1, 4]. What the comparability ODR can be reused for to ease the measure of completeness is about *constraint-based completeness*. As illustrated in this article, checking such a completeness may lead the KB authors to represent information that increase the KB precision and then enable the finding of yet-undetected problems. Increasing such a completeness does not mean increasing inferencing speed.

This article showed how SPARQL queries could be used for implementing comparability ODRs. More generally, most *transformation languages or systems* that exploit KRs could be similarly reused. [18] and [19] present such systems. The proposed SPARQL queries have been validated experimentally (using Corese [19], a tool which includes an OWL-2 inference engine and a SPARQL engine). Unsurprisingly, in the tested existing ontologies, many objects were not compliant with the ODRs.

References

1. Mendes, P.N., Bizer, C., Miklos, Z., Calbimonte, J.P., Moraru, A., Flouri G.: D2.1 Conceptual model and best practices for high-quality metadata publishing. Delivery 2.1 of PlanetData, FP7 project 257641 (2012).
2. Farias Lóscio, B., Burle, C., Calegari, N.: Data on the Web Best Practices. W3C Recommendation 31 January 2017. https://www.w3.org/TR/dwbp/ (2017).
3. Presutti, V., Gangemi, A.: Content Ontology Design Patterns as practical building blocks for web ontologies. In: ER 2008, Spain (2008).
4. Dodds, L., Davis, I.: Linked Data Patterns – A pattern catalogue for modelling, publishing, and consuming Linked Data. http://patterns.dataincubator.org/book/ 56 pages (2012).
5. Ruy, F.B., Guizzardi, G., Falbo, R.A., Reginato C.C., Santos, V.A.: From reference ontologies to ontology patterns and back. Data & Knowledge Engineering, vol. 109 (C), 41–69 (2017).
6. Roussey C., Corcho Ó, Vilches Blázquez L.M.: A catalogue of OWL ontology antipatterns. In: K-CAP 2009, Redondo Beach, CA, USA, 205–206 (2009).
7. Zaveri, A., Rula, A., Maurino, A., Pietrobon, R., Lehmann, J., Auer, S.: Quality assessment for linked data: A survey. Semantic Web, vol. 7(1), 63–93 (2016).
8. Martin, Ph., Corby, O.: Ontology Design Rules Based On Comparability Via Particular Relations. Extended version and on-line companion article for this present article. http://www.webkb.org/kb/it/o_knowledge/d_odr_content_article.html (2019).
9. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S.: OWL 2 Web Ontology Language Primer (Second Edition). W3C Recommendation 11 December 2012. https://www.w3.org/TR/owl2-primer/ (2012).
10. Harris, S., Seaborne, A.: SPARQL 1.1 Overview. W3C Recommendation 21 March 2013. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/ (2013).
11. Martin, Ph.: Correction and Extension of WordNet 1.7. In: ICCS 2003 (Springer, LNAI 2746,160-173), Dresden, Germany (2003).
12. Marino, O., Rechenmann, F., Uvietta, P.: Multiple Perspectives and Classification Mechanism in Object-Oriented Representation. In: ECAI 1990, 425–430, Pitman Publishing London, Stockholm, Sweden (1990).
13. Bachimont, B., Isaac, A., Troncy, R.: Semantic Commitment for Designing Ontologies: A Proposal. In: EKAW 2002, LNCS, vol 2473, 114–121, Springer Berlin, Spain (2002).
14. Dromey, R.G.: Scaleable Formalization of Imperfect Knowledge. In: AWCVS 2006, 1st Asian Working Conference on Verified Software, 29–31, Macao SAR, China (2006).
15. Rector, A., Brandt, S., Drummond, N., Horridge, M., Pulestin, C., Stevens, R.: Engineering use cases for modular development of ontologies in OWL. Applied Ontology, 7(2), 113–132, IOS Press (2012).
16. Djakhdjakha, L., Mounir, H., Boufaïda, Z.: Towards a representation for multi-viewpoints ontology alignments. IJMSO, International Journal of Metadata, Semantics and Ontologies, 9(2), 91–102, Inderscience Publishers, Geneva (2014).
17. Corman, J., Aussenac-Gilles, N., Vieu, L.: Prioritized Base Debugging in Description Logics. In: JOWO@IJCAI (2015).
18. Djedidi, R., Aufaure, M.: Ontology Change Management. In: I-SEMANTICS 2009, 611–621 (2009).
19. Galárraga, L., Hose, K., Razniewski, S.: Enabling completeness-aware querying in SPARQL. In: WebDB 2017, 19–22, Chicago, IL, USA (2017).
20. Zamazal, O., Svátek, V.: PatOMat – Versatile Framework for Pattern-Based Ontology Transformation. Computing and Informatics, vol. 34 (2), 305–336 (2015).
21. Corby, O., Faron-Zucker, C.: STTL: A SPARQL-based Transformation Language for RDF. In: WEBIST 2015, 11th International Conference on Web Information Systems and Technologies, Lisbon, Portugal (2015).