Mapping Guidelines for CB-NL

Rinke Hoekstra
January 2015
STATUS: Working Draft, comments to rinke.hoekstra@vu.nl

Introduction

Defining mappings between Linked Data datasets can be tricky because of three reasons:

1. Linked Data data can be expressed at varying levels of **expressiveness** (plain RDF, SKOS, RDFS, OWL2)
2. Mappings can be expressed at varying levels of **expressiveness** (using SKOS, or RDF/RDFS/OWL2)
3. A distinction needs to be made between **instance** mapping and **schema** mapping

This document briefly discusses the caveats and tradeoffs when dealing with these characteristics of Linked Data.[1]

Languages vs. vocabularies and ontologies, and vocabularies

Languages

The languages underlying the Linked Data cloud, RDF, RDFS and OWL2, together form a family of so-called **knowledge representation** languages. Like any language, knowledge representation languages have:

1. a **vocabulary**, the `words' of the language,
2. a **grammar**, the allowed combinations of the words when forming sentences, and
3. a **semantics**, the meaning of these combinations of words in sentences.
4. a **syntax**, the rules for writing the sentences down (e.g. interpunction)

Here are some examples:

Language	Vocabulary	Grammar	Semantics	Syntax
RDF	`rdf:type`, `rdf:Property`	directed labeled graph	if `<s,p,o>` then `<p,rdf:type,rdf:Property>`	Turtle, RDF/XML, JSON-LD, N-Triples, etc. but also URI's/IRI's
RDFS	`rdfs:subClassOf`, `rdfs:domain`	directed labeled graph	if `<p,rdfs:domain,C>` and `<s,p,o>` then `<s,rdf:type,C>` [1]	any RDF syntax
OWL2	`owl:Class`, `owl:equivalentClass`	Structural Specification (DL) and directed labeled graph (Full) [1]	iff `<A,rdfs:subClassOf,B>` and `<A,rdfs:subClassOf,B>` then `<A,owl:equivalentClass,B>` [1]	(Subsets of) all RDF syntaxes, Manchester Syntax, Functional Style Syntax.

To summarize; words combined in a grammar form sentences that have semantics according to a set of **inference rules** (or *axioms*). OWL2 is a bit of a strange beast here, since it does not allow for all constructs that RDF and RDFS allow for to construct sentences.

Instances and Schemas

Every language serves to have some reference to the **real world**. In Linked Data land, these are called the "instances" or facts in a knowledge base ("individuals" in OWL-speak).

As a rule of thumb: instances are those entities in a knowledge base that do not have an `rdf:type` relation to any of the terms in the vocabularies of RDF, RDFS or OWL2 (with the exception of `owl:NamedIndividual`).

Sentences that only refer to **terms** are called **schema** information.

- An instance is said to be of a certain **type** if it has a `rdf:type` relation with an instance of `owl:Class` (or `rdfs:Class`) that represents that type. **NB:** Never use `rdfs:Class` but use `owl:Class` instead.
- That type is then called a **class** or **concept** (to be precise: the class represents a concept)

Punning (word play) is needed when a resource is used both as *instance* and as *class*. This is allowed in RDF, RDFS and OWL2 Full, but not in OWL2 DL. However, because the context of use is known, OWL2 DL reasoners neatly distinguish the two cases, thus allowing for "punning".

Vocabularies and Ontologies

Often mentioned in the same breath with RDF are terms such as SKOS, Dublin Core, PROV, FOAF and QUDT. These are so-called **vocabularies** or **ontologies**, and they can be specified by standards-bodies such as the W3C (SKOS, PROV) or the Dublin Core Metadata Initiative (Dublin Core), but they are often less grounded in institutions, such as QUDT (TopQuadrant and NASA) and more strongly FOAF. In the Linked Data cloud, the success (and reliability) of a vocabulary is measured by how widespread its use is.

Vocabularies are reliably distinct from knowledge representation languages in that they only provide new **vocabulary**

terms and do not introduce new inference rules, new syntaxes, or new semantics.

"What!? No new semantics!?"

Well... no **new** semantics, but similar to normal languages, the words we use, and the way we combine them do **have** semantics.

The inference rules of the knowledge representation languages allow for two types of reasoning:

1. Inferring new information between **terms** used in a set of sentences. This is also called T-Box (T for terminology) or schema inference, and serves to infer whether a knowledge base is **internally consistent** and makes **explicit** all implicit information in the knowledge base.
2. Inferring new information about **instances** referred to in the knowledge base. This is also called A-Box (A for assertion) instance inference. Although instances also play a role in determining whether a knowledge base is consistent, the main purpose is usually to infer new **relations** amongst instances, most importantly **sameness** and **type** information. For the latter to work, an ontology needs to specify at least *domain* and *range* information for properties **or** define classes using **necessary and sufficient conditions** (using the `owl:equivalentClass` construct).

The distinction between **vocabularies** and **ontologies** is that the latter typically contain **more schema** information, and therefore offer more means to infer new information: they are more **expressive**.

To confuse matters...

In the libary and cultural heritage communities, it is common to specify large, structured terminologies for annotation purposes. These terminologies are called **knowledge organization systems**. SKOS, the **simple knowledge organization system** is a Linked Data vocabulary that captures the most common relations one can find in such knowledge organization systems (broader, narrower, related) and between them (exact match, close match, etc.).

The confusion is that these knowledge organization systems are often called **vocabularies** as well. They are distinctly different from the vocabularies we discussed before in that the terms they provide are **instances** and do not form part of the schema of the knowledge base. A SKOS knowledge base can thus be seen as a knowledge base about concepts, where, say a Vehicle knowledge base is usually about actual vehicles.[2]

Because a SKOS knowledge base thus has *knowledge* as its domain, it is essentially a `reification' of RDFS and OWL. Fortunately, SKOS is *not a language* and does not add new semantics that is incompatible with the Linked Data family.

In the following we'll use the term **thesaurus** to refer to vocabularies of this type.

Other types of knowledge bases

Indeed, sometimes a knowledge base does not fall into either of these categories. For instance, a knwoledge base that uses its own set of terms to describe resources, but does not specify these terms explicitly in a schema (an ontology).

When specifying mappings, this schema needs to be identified. It is therefore important to determine what parts of the data should be considered part of the schema, and what parts are 'data'. The relations used are typically schema entities.

Furthermore, often datasets use some type of category system for the resources described in it. The mapping designer needs to decide whether it makes sense to define the categories as classes or as instances (SKOS concepts). Generally speaking, it is more straightforward to specify them as SKOS concepts.

SPARQL vs OWL

Although SPARQL is a *query* language, it can also be used to implement rather expressive rules (e.g. using

`INSERT { ... } WHERE { ... }` via the SPARQL update protocol). Several rule-like languages have been implemented that use a SPARQL-like syntax. Examples are SPIN (TopQuadrant) and SPARQL Rules (Clark&Parsia, which partially maps onto SWRL).

To put SPARQL-as-rule-language in to context, it defines its own vocabulary, grammar, semantics and syntax that operates *on top of* the other members of the Linked Data family, but is not a part of it.

The drawback is thus that the rules cannot be exchanged as easily amongst parties because they do not have URIs and cannot be exchanged as RDF graphs. Also, their semantics is incompatible with RDF, RDFS and OWL (closed world vs. open world). Most importantly: reasoning using SPARQL rules is **order dependent**: the rules have to fire in a specific order, often multiple times, to make sure all intended inferences are indeed reached.

That said, it is often convenient to express simple rules used e.g. for combination, visualization or presentation purposes in SPARQL. Especially since it is much more expressive when it concerns data-type and data-value reasoning. Keep in mind that these rules remain application-specific and may cause unwanted and unintended side-effects when used in combination with other, external datasets. Also, consumers of your dataset will not be aware of the SPARQL rules, and thus have a less expressive dataset or knowledge base.

The CB-NL Ontology

CB-NL is an ontology specified in OWL: it focuses on explicit schema-level definitions of concepts.

All concepts are represented as **classes** that restrict a fixed number of properties:

- `cbnlcore:hasAppearance`
- `cbnlcore:hasApplication`
- `cbnlcore:hasFunction`
- `cbnlcore:hasTechnology`

However, CB-NL is not an expressive ontology:

- The range of these properties is not specified, i.e. the instances in the *object* position of properties are not automatically classified as having a certain type.
- The classes are *not* defined using *necessary and sufficient conditions*, i.e. class restrictions are not sufficiently expressive to automatically infer instances as belonging to a certain type.

This means that an OWL reasoner will only infer potential new relations amongst classes.

Mapping Strategies

This section discusses a number of mapping strategies for mapping between ontologies and vocabularies of varying levels of expressiveness. For this we distinguish the following types of knowledge bases:

1. Expressive **ontologies** that use necessary and sufficient conditions suitable for automatic type inferencing.
2. Inexpressive ontologies (or **vocabularies**) that only specify subclass and/or subproperty relations
3. **SKOS vocabularies**, which only specify SKOS concepts and (hierarchical) relations amongst them.

The main distinction between 1. and 2. on the one hand, and 3. on the other, is that the former serve to **represent** classes of records in your database (the instances, or individual data items), whereas the latter is only used to **annotate** these instances.

The second distinction is that SKOS vocabularies typically use a fixed set of standard properties (relations from SKOS, DC or other common vocabularies), whereas the other two often use a mix of standard properties and custom ones (see CB-NL). This means that for the ontologies and vocabularies, properties will have to be mapped as well.

Mappings can be expressed in two ways:

a. Mappings that use constructs (properties) that are part of one of the languages; these typically have the most far-reaching consequences qua semantics: inferences on the first ontology are transferred to the other ontology and vice versa. Examples are the use of `rdf:type`, `rdfs:subClassOf`, `owl:equivalentClass` and `rdfs:subPropertyOf`.
b. Mappings that use constructs (properties) that are part of an ontology or vocabulary; these may or may not have consequences for semantics (no new inferences follow). An example is the use of `skos:exactMatch`. This category includes properties from the languages that do not have formal semantics, such as `rdfs:seeAlso`.

NB: Mappings are **viral** in that they are transitive across multiple mapped ontologies: if A is mapped onto B, and B is mapped onto C, then inferences from A hold for C as well.

Named Graphs

For this reason it is advisable to define mappings using a tool that tracks the provenance of mappings by capturing the mapping in a *Named Graph* that has associated information that expresses the mappings' provenance (authorship, reason for adding etc.). Named graphs can also be used to define contexts of combinations of ontologies. These contexts can then be considered separately from the rest of the RDF graph, allowing for context-dependent semantics. Note that this may cause unwanted side-effects when translating from an ontology/vocabulary to an ontology/vocabulary outside the context.

Requirements for Mapping

- As many entities and concepts in both knowledge bases should be mapped
- Entities and concepts in both knowledge bases may be enriched by the mapping (new inferences follow), but not altered in a way that the knowledge base is misaligned or no longer consistent with respect to the original knowledge base

Ontologies to Ontologies

When mapping ontologies to eachother the goal is to make sure that the schema-information in both ontologies is integrated for concepts and relations that have the same intended meaning.

Because we're dealing with expressive ontologies, the preferred way of mapping is to use mapping type **a**: use constructs from one of the knowledge representation languages.

When specifying mappings it is good practice to consider classes as sets of potential instances. You can then draw Venn diagrams (circles represent sets) to determine how classes from the two ontologies overlap.

In the, we consider two ontologies `a` and `b`:

```
a:X owl:equivalentClass [ rdf:type           owl:Restriction ;
                          owl:onProperty     a:p   ;
                          owl:someValuesFrom a:Y
                        ] .

a:y rdf:type            a:Y .
a:x a:p                 a:y .
```

A reasoner will infer that `<a:x,rdf:type,a:X>`, and that `<a:X,rdf:type,owl:Class>`, `<a:X,rdfs:subClassOf,owl:Thing>` and `<a:p,rdf:type,rdf:Property>`.

And:

```
b:X rdf:type            owl:Class .
```

A reasoner will infer that `<b:X,rdf:type,owl:Class>` and `<b:X,rdfs:subClassOf,owl:Thing>`.

Classes

Situation 1

If `a:X` and `b:X` are intensionally exactly the same; all members of class `a:X` are also members of `b:X` ; all inferences that hold for `a:X` should hold for `b:X` as well.

The preffered way of specifying a mapping is using:

```
a:X owl:equivalentClass b:X .
```

The reasoner will then infer (in addition to the inferences above):

```
b:X owl:equivalentClass a:X .
b:X owl:equivalentClass [ rdf:type           owl:Restriction ;
                          owl:onProperty     a:p   ;
                          owl:someValuesFrom a:Y
                        ] .

a:x rdf:type            b:X .
```

Situation 2

The classes overlap, but some instances of class `b:X` do not belong to `a:X` , but all instances of class `a:X` do not belong to `b:X` .

The preferred way of specifying a mapping is using:

```
a:X owl:subClassOf  b:X .
```

The reasoner will then infer (in addition to the inference above):

```
a:x rdf:type            b:X .
```

Situation 3

The classes overlap, but some instances of class `a:X` do not belong to `b:X` , but all instances of class `b:X` do not belong to `a:X` .

The preferred way of specifying a mapping is using:

```
b:X owl:subClassOf  a:X .
```

The reasoner will then infer (in addition to the inference above):

```
b:X rdfs:subClassOf     a:X .
b:X rdfs:subClassOf     [ rdf:type           owl:Restriction ;
                          owl:onProperty     a:p   ;
                          owl:someValuesFrom a:Y
                        ] .
```

Note that the reasoner does not infer that `a:x` is an instance of class `b:X` because it may lie outside the part of `a:X` that is covered by `b:X` .

Situation 4

The classes overlap, but some instances of class `a:X` do not belong to `b:X` , and some instances of class `b:X`

do not belong to `a:X`.

One cannot specify a direct mapping between the classes, but needs to for instance to:

- Make explicit **how** they overlap, e.g. by introducing a new property whose value distinguishes members of class `b:X` from `a:X`: `b:X` is the union of a part of class `a:X` and those individuals that have some value from `b:Y` for property `b:p`. Of course this is the most **powerful** way of integrating two ontologies.

```
b:X owl:subClassOf  [ owl:unionOf (
                        a:X ,
                        [ rdf:type           owl:Restriction ;
                          owl:onProperty     b:p ;
                          owl:someValuesFrom b:Y
                        ]
                      )
                    ] .
```

- Introduce new classes that form the union and intersection of both classes:

```
m:Xa_or_b   owl:equivalentClass [ owl:unionOf (a:X, b:X) ] .
m:Xa_and_b  owl:equivalentClass [ owl:intersectionOf (a:X, b:X) ] .
```

- Merely state that `a:X` and `b:X` have a common super class:

```
a:X owl:subClassOf  m:X .
b:X owl:subClassOf  m:X .
```

In the first case, the reasoner will infer merely that `b:X` is a subclass of `a:X`.

In the second case, the reasoner will infer that `a:x` is of type `m:Xa_or_b`, but not necessarily of `m:Xa_and_b`, because `a:x` may lie outside the part of `a:X` that is covered by `b:X`.

In the third case, the reasoner will infer that `a:x` is of type `m:X`, but does not infer anything else.

Situation 5

The classes do not overlap, but still belong together somehow. This is best represented using a common super class, as above:

```
a:X owl:subClassOf  m:X .
b:X owl:subClassOf  m:X .
```

The reasoner will infer that `a:x` is of type `m:X`, but does not infer anything else.

Properties

Suppose now that `b:X` is defined much more similar to `a:X` (i.e. it also has an equivalent class restriction on a property `b:p`), and that `b:Y` and `a:Y` are equivalent. It is often the case that classes from different ontologies are defined using similar properties:

```
b:X owl:equivalentClass [ rdf:type           owl:Restriction ;
                          owl:onProperty     b:p   ;
                          owl:someValuesFrom b:Y
                        ] .

b:Y owl:equivalentClass a:Y .
```

Properties can be mapped in the following ways:

Situation 6

If all entities related via `a:p` should also be related via `b:p` and vice versa, then the properties are equivalent:

```
a:p owl:equivalentProperty b:p .
```

This will make the reasoner infer that not only the properties are equivalent, but also the classes, since the restrictions have become equivalent.

Situation 7

If some entities related via `a:p` should also be related via `b:p` but not vice versa, `a:p` is a subproperty of `b:p`:

```
a:p rdfs:subPropertyOf b:p .
```

This will make the reasoner infer that `a:X` is a subclass of `b:X`, since every member of class `a:X` will have a `a:p` relation with som `a:Y`, and because of our mapping, a `b:p` relation as well, which will make that it will also become a member of `b:X`.

Likewise if the inverse mapping is specified.

Situation 8

If no entities related via `a:p` are ever related via `b:p` as well, the properties are disjoint.

```
a:p owl:disjointProperty b:p .
```

Counter intuitive, perhaps, but this has no consequence for the relation between `a:X` and `b:X`. Although both specify that there should be a relation with a `b:Y` / `a:Y` instance, the restrictions do not specify exactly *which* individual: the property mappings only hold for individual pairs of instances, not for the class definitions.

Situation 9

In a situation where the domain and range of a property are specified, e.g.:

```
a:p rdfs:domain a:X ;
    rdfs:range  a:Y .
```

and suppose that we do not specify that `b:Y owl:equivalentClass a:Y` .

The mappings listed above will have the following effect:

- A mapping using `owl:equivalentProperty` will make `b:X` a subclass of `a:X` (since at least some members of `b:X` must be in `a:X`, given the definition of `b:X`)
- A mapping using `a:p rdfs:subPropertyOf b:p` will make `b:X` a superclass of `a:X`
- A mapping using `b:p rdfs:subPropertyOf a:p` will make `b:X` a subclass of `a:X`
- A mapping using `a:p owl:disjointProperty b:p` will not have any consequences.

Situation 10

In some cases, ontology **a** represents a term as a property, while ontology **b** represents it as a class. A common case where this happens is with so-called *roles*, such as `employee', which have a relational nature (you are always an employee of an organisation).

This situation is likely to occur in the case of CB-NL because of the design decision to represent all properties and attributes as classes.

Consider the following ontology **a**:

```
a:X owl:equivalentClass [ rdf:type           owl:Restriction ;
                          owl:onProperty     a:p   ;
                          owl:someValuesFrom a:Y
                        ] .

a:y rdf:type            a:Y .
a:x a:p                 a:y .
```

which is the one from situation #1. And ontology **b**:

```
b:P rdf:type            owl:Class .

b:x b:has_property      b:p .

b:p rdf:type            b:P ;
    b:has_value         b:y .
```

Note that because `b:P` is a property class, we have specific instances for every occurrence of the relation "p": the relation is reified.

1. The first option is mapping through **punning**: we can simply specify that `a:p` and `b:P` are equivalent classes or properties. However, the two contexts will be interpreted separately by reasoners, meaning that the equivalent class `a:p` is treated differently from the property `a:p`.
2. The second option is to take the stance that the value of the property `a:p` (in this case `a:y`) is an instance of the property class (`b:P`), rather than that the instance of the property class is a representation of the occurrence of the property. In that case, one could define a mapping by means of a range restriction: `<a:p,rdfs:range,b:P>`. I would advise against this approach because it confounds two ontological categories: properties and their values.
3. The third option is to define a **role-inclusion axiom**, or property chain, that can infer the existence of an `a:p` relation, given the existence of the reified `b:p` relation instance. Unfortunately this does not work the other way around because of restrictions in the OWL2 semantics.

A role chain is a sequence of relations between instances; given a match with such a chain, the reasoner will infer the exisence of a relation between the instances at both ends of the chain.

For the trick to work, we need to ensure that the instance of `b:P` is identifyable as such in the role chain. This is done by means of a **self restriction**: every instance of the `b:P` class is related to itself via the `b:is_P` property.

```
b:P owl:equivalentClass [ rdf:type          owl:Restriction ;
                          owl:onProperty    b:is_P ;
                          owl:hasSelf       "true"^^xds:boolean
                        ] .
```

Given the ontologies above, the reasoner will now infer that `<b:p,b:is_P,b:p>`.

We then define the role inclusion axiom:

```
a:p owl:propertyChainAxiom ( b:has_property b:is_P b:has_value ) .
```

which states that anything that 'has property' something that is an instance of `b:P` which `has value` some value, is related to that value via the `a:p` relation.

Given our knowledge base:

```
b:x b:has_property      b:p .

b:p b:is_P              b:p .
    b:has_value         b:y .
```

we can then infer that:

```
b:x a:p                 b:y .
```

Vocabularies to Ontologies

Vocabularies, lightweight ontologies, can be mapped to ontologies in much the same way as illustrated in the section above. However, the mappings are generally much simpler since vocabularies typically do not use equivalent class restrictions.

The preferred way of mapping is thus to use mapping type **a**: use constructs from one of the knowledge representation languages.

The reason is that even though the class definitions are usually not strong enough to infer class-membership for instances, the instances may still be explicitly typed as belonging to one or more classes.

Note that vocabularies often **do** specify domain and range for properties. As discussed above, domain and range can easily propagate unexpected inferences. This means that one should be extra careful when dealing with them.

SKOS Thesauri to Vocabularies or Ontologies

SKOS thesauri only operate at the A-Box level: they only contain instances of `skos:Concept` and `skos:ConceptScheme`. However, because SKOS itself *is* a lightweight ontology, mapping SKOS thesauri to one of the other types of ontology using constructs from one of the languages (mapping type **a**) may have consequences. Often the mapping type of choice is therefore **b**, where mappings are specified using the SKOS vocabulary itself.

The SKOS mapping relations are the following:

- `skos:exactMatch` when the intended meaning of the concepts is the same
- `skos:closeMatch` when the two concepts are almost the same
- `skos:broadMatch` when the subject is narrower than the object concept
- `skos:narrowMatch` when the subject is broader than the object concept
- `skos:relatedMatch` when the two concepts are semantically related

These mapping relations are intended to hold between concepts in **different** `skos:ConceptScheme` s.

Situation 11

An interesting way of mapping thesauri to ontologies is by pretending that the `skos:Concept` instances are classes, and vice versa, that the classes in the ontology are instances (through **punning**). One can then use both `owl:equivalentClass` and `rdfs:subClassOf` relations, and the SKOS mapping relations (in this case `skos:exactMatch` and `skos:broadMatch`) to specify the mappings between the two knowledge bases.

The advantage is that from both perspectives (SKOS and RDFS/OWL) the mappings will be valid: the SKOS thesaurus will look like an extremely lightweight ontology, while the ontology will appear to be an under-specified SKOS thesaurus.

The only problem is that there is no mechanical means to check consistency of the combined knowledge bases. Even though SKOS concepts and relations do not really carry much semantics, they do have an intended meaning that should correspond with the mapped ontology.

NOTE An ugly hack, that takes us outside the OWL2 semantics (and the intended semantics for SKOS) would be to

specify `skos:broadMatch` and `skos:broader` as equivalent property of `rdfs:subClassOf`, and `skos:exactMatch` as equivalent to `owl:equivalentClass`. Don't try this at home!

Situation 12

A more puristic approach is to consider that the purpose of the SKOS thesaurus is to annotate resources: SKOS concepts form part of a description, rather than a definition of a resource.

Suppose we have a corpus of resources annotated using a known annotation property, such as `dcterms:subject`, we can then determine the class of the annotated resources by introducing value restrictions from thesaurus `a` on this property for classes in the ontology `b`:

```
_:res   dcterms:subject     a:concept1, a:concept2 .

b:X     owl:equivalentClass [ rdf:type          owl:Restriction ;
                              owl:onProperty    dcterms:subject ;
                              owl:hasValue      a:concept1
                            ] .
```

This way, any resource annotated using `a:concept1` will be inferred as being of type `b:X`. Of course, these mappings can become much more complicated, by e.g. requiring the co-occurrence of two annotations, or by leveraging additional relations that may exist amongst individual resources. For instance, a resource annotated with `dcterms:subject a:stick` with a `a:held_by` relation with a resource annotated with `dcterms:subject a:hooligan` could be classified as `b:Weapon`.

A drawback of this approach is that the mappings only 'fire' when not only the schema, but the data is asserted in the knowledge base as well. This can be unrealistic for some scenarios with a lot of data; but often having only the data-item under consideration in the KB suffices.

SKOS Thesauri to SKOS Thesauri

In general, this is considered the simplest form of mapping.

To reiterate, the SKOS mapping relations are the following:

- `skos:exactMatch` when the intended meaning of the concepts is the same
- `skos:closeMatch` when the two concepts are almost the same
- `skos:broadMatch` when the subject is narrower than the object concept
- `skos:narrowMatch` when the subject is broader than the object concept
- `skos:relatedMatch` when the two concepts are semantically related

These mapping relations are intended to hold between concepts in **different** `skos:ConceptScheme` s.

The advised mapping type is **b**, using terms from the SKOS ontology.

Situation 13

Simply using the SKOS relation often suffices if the task that is intended to be supported by the mappings one that is similar to browsing or searching. Simple SPARQL queries can be used to find related resources annotated using the other thesaurus.

The drawback is that these SPARQL queries do not form part of the knowledge base. However, since SKOS is widely used, the queries are not likely to differ very much from one application to another, giving little room for alternative interpretations.

Situation 14

Another approach is to use the RDFS/OWL reasoning machinery to specify that two SKOS concepts are the same.

The `owl:sameAs` relation comes to mind, and can indeed be used to make two concepts formally identical.

The drawback of this approach is that *everything* said about one concept will now also be said about the other (including authorship, what scheme it belongs to etc.). To make matters worse, this is works *transitively*, propagating the identicity through all mapped thesauri and ontologies.

SKOS mappings were explicitly defined to circumvent these drawbacks: situation #13 is indeed the most unfortunate one.

Situation 15

Finally, one could use the same approach as iterated for situation #11: map SKOS thesauri by reference to the resources they are annotated with.

For instance:

```
a:res   dcterms:subject     a:concept1, a:concept2 .
b:res   dcterms:subject     b:concept1, b:concept2 .

m:C1    owl:equivalentClass [ rdf:type          owl:Restriction ;
                              owl:onProperty    dcterms:subject ;
                              owl:hasValue      a:concept1
                            ] ;
        owl:equivalentClass [ rdf:type          owl:Restriction ;
                              owl:onProperty    dcterms:subject ;
                              owl:hasValue      b:concept1
                            ] .
```

This specifies that any resource annotated with `a:concept1` is a member of class `m:C1`, and is therefore also annotated with `b:concept1`.

This could be considered "a bit much" were it not for the fact that SKOS concepts often do not map directly onto each other. Sometimes a mapping is *1:n* or even *n:m*, rendering the standard SKOS mapping relations less ideal.

This approach is also a good way to refine the `skos:closeMatch` relation.

A drawback of this approach is that the mappings only 'fire' when not only the schema, but the data is asserted in the knowledge base as well. This can be unrealistic for some scenarios with a lot of data; but often having only the data-item under consideration in the KB suffices.

Discussion

This document briefly introduces the concepts underlying the languages RDF, RDFS and OWL2, and the SKOS vocabulary. It distinguishes between expressive ontologies, vocabularies and thesauri.

We then discuss 15 situations where alternative approaches could be used to specify the mappings. For each of these, we tried to list the drawbacks.

This is a **working document**. Please send comments, remarks, requests for additions to Rinke Hoekstra rinke.hoekstra@vu.nl. We'll try to accomodate them.

---

1. I will use the terms “Linked Data” and “Semantic Web” as equivalent. Both have the same origin, but take slightly different perspectives on the same thing: “Semantic Web” is an earlier term from the ‘90s that emphasizes semantics and reuse of schema information, while Linked Data emphasizes the reuse of data item identifiers.↩
2. But not vice versa!↩

3. The semantics of OWL2 DL is defined against the Structural Specification, not RDF Graphs, this is because the OWL2 Direct Semantics relies on restrictions on the allowed combinations of “words” in RDF. The OWL2 Full Semantics *is* expressed against a graph since it does not have these restrictions.↩

4. **and** vice versa!↩

5. Of course this is not particular to SKOS. ↩