W3C Working Draft

Simple part-whole relations in OWL Ontologies

W3C Editor's Draft 24 Mar 2005

This version:
...
Latest version:
...
Previous versions:
This is the first public version

Editors:
Alan Rector, University of Manchester
Chris Welty, IBM Research

Abstract

Representing part-whole relations is a very common issue for those developing ontologies for the Semantic Web. OWL does not provide any built-in primitives for part-whole relations (as it does for the subclass relation), but contains sufficient expressive power to capture most, but not all, of the common cases. The study of part-whole relations is an entire field in itself - "mereology" - this note is intended only to deal with straightforward cases for defining classes involving part-whole relations.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www. w3.org/TR/.

This document will be a part of a larger document that will provide an introduction and overview of all ontology design patterns produced by the Semantic Web Best Practices and Deployment Working Group.

This document is a W3C Working Draft and is expected to change. The SWBPD WG does not expect this document to become a Recommendation. Rather, after further development, review and refinement, it will be published and maintained as a WG Note.

As a candidate Public Working Draft, we encourage public comments. Please send comments to public-swbp-wg@w3.org

Open issues, todo items:

- update faults examples from actual ontology
- fix camel case vs. underlining
- get new picture
- replace "work around" when needed
- add references for "flavours" of part-whole relations

Publication as a draft does not imply endorsement by the W3C Membership. This document is a draft and may be updated, replaced or made obsolete by other documents at any time. It is inappropriate to cite this document as other than work in progress.

General issues

Basics

Many applications require representation of part-whole relations - catalogues of parts, fault diagnosis, anatomy, geography, etc. The study of part-whole relations is a large field in its own right - "mereology" and "mereotopology" and has been the topic of many papers, see the references section for a useful list.

OWL does not contain specific primitives for part-whole relations (as it does for the subclass relation, for example), but it does support sufficient machinery to express most of what one may want to represent about part-whole relations. Where it does not, there are a number of "work-arounds" that suffice in most situations. This note will provide basic schemas for expressing part-whole relations in OWL.

Transitive relations - parts and direct parts.

An important and common requirement for the basic relation from a part to its whole that it is transitive, i.e. if A is part of B, and B is part of C, then A is part of C. OWL provides a general construct for declaring properties to be transitive. If we define a property, say `partOf`, to be transitive, then any reasoner conformant with OWL will draw the conclusions that the parts of C include both A and B.

In many applications, what is needed is not a list of all parts but rather a list of the next level breakdown of parts, the "direct parts" of a given entity. It is therfore often useful to use the property hierarchy to define a subproperty of `partOf` that is not transitive and links each subpart just to the next level. For these examples we shall call this suproperty `partOf_directly`.

Choosing whether to use `partOf` or `hasPart`

OWL supports inverse relations, so we can define an inverse of `partOf`, say `hasPart`. For any two individuals `I1` and `I2`, if `"I1 partOf I2"` then `"I2 hasPart I1"`. However, care must be taken when using inverses in restrictions on classes. To say that "All As are parts of some B" does not imply that "All Bs have some As as parts", i.e. the restriction

```
    A partOf someValuesFrom(B)
```

does not imply

```
    B hasPart someValuesFrom(A)
```

Therefore, if we want to say both that "all As are parts of Bs" and "all Bs have part some A", we have to assert each statement separately. Such pairs of statements are sometimes called "reciprocals".

Unfortunately, all current OWL reasoners scale very badly for large part-whole hierarchies connected by *both* `hasPart` and `partOf`. Therefore, if reasoners are to be used, it is usually necessary to choose to use either `partOf` or `hasPart` but not both. Almost always it is preferable to use `partOf` because the most common queries and class definitions are for the parts of things, e.g. the class of all parts of a car.

Use cases

Parts and wholes are ubiquitous in many applications:

1. A parts inventory for the devices made in a factory in which we want to be able to find the "explosion" of parts required.
2. A fault finding system for an device in which we want to progressively narrow down the functional region of the fault.
3. An anatomy representation such as the Digital Anatomist Foundational Model of Anatomy (??URL??)
4. A document retrieval system, in which documents are divided into chapters, sections, paragraphs etc. (However, note that parthood, as explained in this document, does not take order into account).

Representation Pattern 1: Representing a part-whole hierarchy

1. Define a transitive property `partOf` with an inverse `hasPart` (Choose your own naming conventions to suit).
2. If a level by level explosion is required, define a subproperty `partOf_directly` with an inverse `hasPart_directly`.

3. Choose whether to use the `partOf` or `hasPart` relation as the basic relation amongst classes. If in doubt, choose `partOf`.
4. Exress the part-whole relations amongst individuals using `hasValue()` with `partOf_directly` (if defined, otherwise with `partOf`).
5. Express the part-whole relations amongst classes using `hasSomeValuesFrom()` with `partOf_directly`
6. If there are any universal (`allValuesFrom`) constraints, add those.

There should now be sufficient information to make basic inferences about parts, e.g. to define a class of all parts of the car, car door, etc.

Examples

Consider a (over simplified) catalog of Vehicle parts, all subsumed by the class *Item*

*Car*s have parts *Motor, Headlight, Wheel*

*Motor*s have parts *Crankcase, Carburetor*

*Headlight*s have parts *head_light_bulb, reflector*

A fragment of the N3 for the above example would then be:

```
partOf
      a       owl:TransitiveProperty , owl:ObjectProperty ;
      owl:inverseOf hasPart .
partOf_directly
      a       owl:ObjectProperty ;
      rdfs:subPropertyOf partOf ;
      owl:inverseOf hasPart_directly .
Car
      a       owl:Class ;
      rdfs:subClassOf Item
Motor
      a       owl:Class ;
      rdfs:subClassOf Item ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty partOf_directly ;
                owl:someValuesFrom Car
              ] .
Crankcase
      a       owl:Class ;
      rdfs:subClassOf Item ;
      rdfs:subClassOf
              [ a       owl:Restriction ;
                owl:onProperty partOf_directly ;
                owl:someValuesFrom Motor
              ] .
```

```
    ...
    etc.
```

Discussion

Several issues arise even from such a simple example. To begin with, the representation using existential restrictions (i.e. owl:someValuesFrom) does not clearly communicate all of the semantics we may want for car parts. For example, a strict reading of the definition of the Crankcase class above is that a crankcase is part of *at least one* motor. In point of fact, a crankcase cannot be part of more than one motor. We may be tempted to add a cardinality restriction (e.g. maxCardinality 1) on partOf to the definition of crankcase, but this would be a mistake; since partOf is transitive, a crankcase is also part of the car the motor is part of. Note also that OWL-DL does not allow transitive properties to have any cardinality restrictions. In general it is best to avoid placing restrictions (including range restrictions) on transitive properties at all.

It would make more sense to add a restriction on the partOf_directly property in the definition of these classes, when it is appropriate. A single crankcase cannot be a *direct* part of more than one motor, a motor cannot be a *direct* part of more than one car, etc., so in these cases a maxCardinality restriction would make the semantics more clear. On the other hand, there is always a tradeoff when employing a reasoner between how precise your semantics are and how much information the reasoner has to consider. In this case, adding a cardinality restriction on all the partOf_directly properties would significantly increase the amount of information handed to a reasoner. One must consider precisely what the ontology will be used for to determine which is more important (enforcing semantic constraints vs. classification). The examples in this note are aimed primarily at use-cases in which no instances of the classes are present.

When considering restrictions on the partOf_directly property for different kinds of parts, the issue of using a universal (owl:allValuesFrom) vs. an existential restriction arises. Many different kinds of things have motors (boats, planes, etc.), and in fact even car motors can exist without being part of a car. This indicates that, ontologically, the existential restriction is simply not true We will discuss this further in the subsequent sections.

Representation Pattern 2: Defining classes for Parts

1. Extend ontology with classes of parts for each level in the part hierarchy (e.g. Car Parts, Motor Parts, etc.), in such a way that a taxonomy can be derived automatically.

Examples

Extending the ontology in pattern 1, we can define the classes: `CarPart` and `CarPart_directly.` Informally:

```
CarPart = partOf someValuesFrom(Car)
```

```
     CarPart_directly = partOf_directly someValuesFrom(Car)
```

In N3:

```
    CarPart_directly
          a       owl:Class ;
          owl:equivalentClass
                  [ a       owl:Restriction ;
                    owl:onProperty partOf_directly ;
                    owl:someValuesFrom Car
                  ] .
    CarPart
          a       owl:Class ;
          owl:equivalentClass
                  [ a       owl:Restriction ;
                    owl:onProperty partOf ;
                    owl:someValuesFrom Car
                  ] .
```

A classifier could then infer that `CarPart_directly` subsumes

```
    Motor
    Headlight
    Wheel
```

and that `CarPart` subsumes

```
    Motor
    Crankcase
    Carburator
    Headlight
    Headlight_bulb
    Reflector
    Wheel
```

This simple list may not be what we want, in which case it is necessary systematically to define a class for the parts of each part, e.g.

```
    MotorPart
        a       owl:Class ;
        owl:equivalentClass
             [ a       owl:Restriction ;
               owl:onProperty partOf ;
               owl:someValuesFrom Motor
             ] .
```

If all are defined in this way we get a hierarchy from the classifier:

```
    CarPart
```

```
Motor
  MotorPart
      Headlight
          HeadlightPart
                Headlight_bulb
...
```

Discussion

These classes exemplify one of the main reasons to choose existential restrictions on the direct part properties over universal restrictions (as discussed in the previous pattern). A classifier would not be able to infer the hierarchy above using universal restrictions on the partOf_direct property in the first pattern, unless there were minimum cardinality restrictions on the property as well.

Ontologically, these classes by themselves are reasonable, a "car part" is indeed anything that is part of a car, however when combined with the existential restrictions on the direct properties, a classifier would infer the hierarchy above. These kinds of hierarchies seem harmless at first glance, but in some contexts are completely wrong: not all motors are car parts, some are boat motors, etc. On the other hand, a motor for a 1969 Porsche 911E is generally considered a "car part" regardless of whether it is in a car or not (it may be for sale). When this is not obvious from the scope of the ontology, it is good practice to reflect these issues in the names of the class, e.g. CarHeadlight.

Representation Pattern 3: Faults in parts and wholes

1. Extend ontology with classes of faults that account for the part hierarchy, e.g. allow a reasoner to ~~conluce~~‸ that a fault in a part is a fault in the whole.

Distinguishing parts from kinds

Although both part-whole relations and `subclassOf` generate hierarchies, it is important not to confuse the part-whole hierarchy with the `subclassOf` hierarchy. This is easily done because in many library and related applications, part-whole and subclass relations are deliberately conflated into a single "broader than / narrower than" axis. For example consider the following:

```
Vehicle
  Car
   Wheel
     Tire
      Pneumatic tire
```

"Car" is a kind of "Vehicle", but "Wheel" is a part of a "Car", "Tire" is a part of a "Wheel", but "Pneumatic tire" is a kind of "Tire". Such hierarchies serve well for navigation, however they are not in general true. Statements about "all vehicles" do not necessarily, or even probably, hold for "all tires".

Examples

However, such hierarchies do need to be recreated in situations that obey the rule "A fault of the part is a kind of fault of the whole". For example you can call for assistance with a fault in your car if you puncture a pneumatic tire. The following hierarchy is a correct `subclassOf` or "kind of" hierarchy of a type that we need to reproduce often in OWL:

```
Fault in Car
  Fault in Wheel
   Fault in Tire
    Fault in Pneumatic tire
```

The easy way to say that what we really mean when we talk about a "fault in a car" is a "fault in a car or any of its parts" [1]: If we use the property `hasLocus` to locate the fault in a particular part of the car, then:

```
Fault_in_car =  Fault AND hasLocus somevaluesFrom(Car OR partOf
someValuesFrom(Car))
```

This is shown diagrammaticaly in Figure 1:

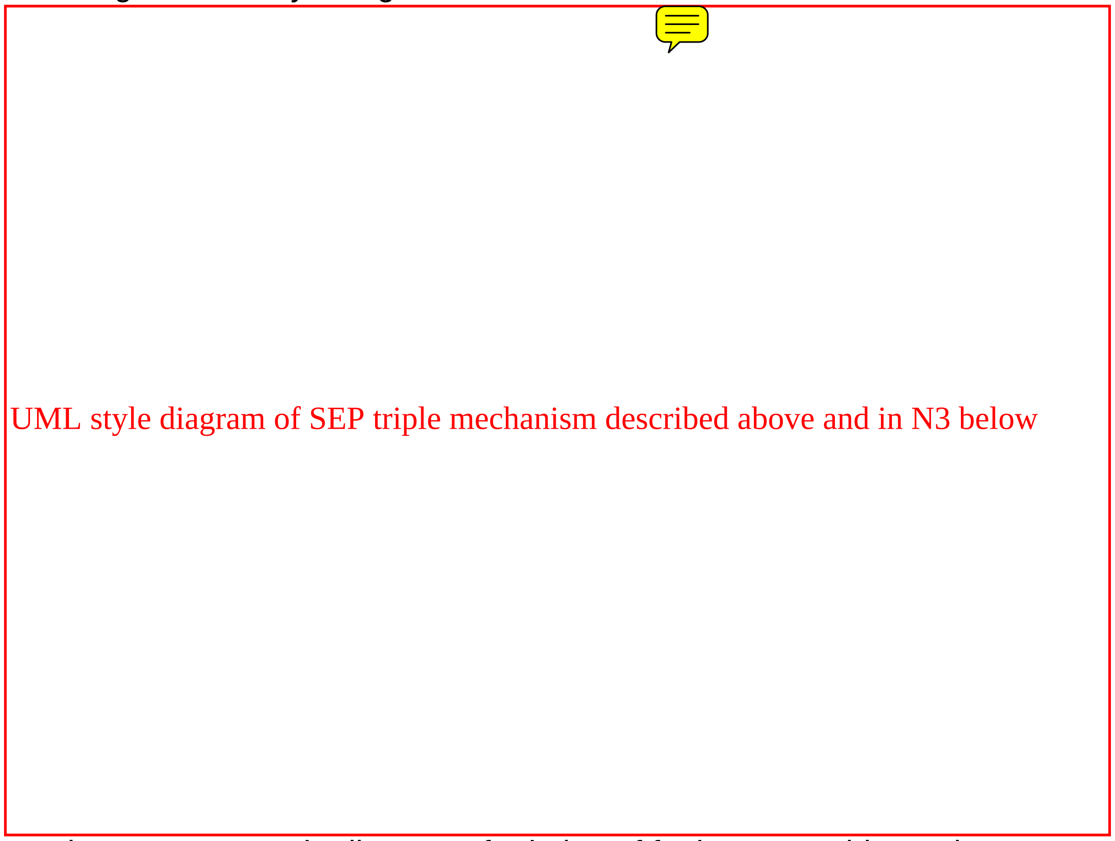
UML style diagram of SEP triple mechanism described above and in N3 below

Figure 1: UML style diagram of relation of faults to to a thing or its parts

Using this pattern, the first two "Fault in ..." classes would be defined as:

```
Fault_in_car
  a  owl:Class ;
```

```
        owl:equivalentClass
          [ a  owl:Class ;
            owl:intersectionOf (Fault [ a  owl:Restriction ;
               owl:onProperty hasLocus ;
               owl:someValuesFrom
                 [ a  owl:Class ;
                   owl:unionOf (Car [ a  owl:Restriction ;
                     owl:onProperty partOf ;
                     owl:someValuesFrom Car
                  ])
               ]
             ])
          ] .

      Fault_in_wheel
        a  owl:Class ;
        owl:equivalentClass
          [ a  owl:Class ;
            owl:intersectionOf (Fault [ a  owl:Restriction ;
               owl:onProperty hasLocus ;
               owl:someValuesFrom
               [ a  owl:Class ;
                owl:unionOf (Wheel[ a  owl:Restriction ;
                   owl:onProperty partOf ;
                   owl:someValuesFrom Wheel            ])
               ]
             ])
          ] .
```

This may look tedious, but can actually be achieved quite simply with scripting tools or the ability to "clone and edit" classes easily.

Discussion

In certain domains, most notably medicine, we generally understand that while body parts (e.g. a heart) can *exist* outside of a body, they do not normally do so. Thus it makes sense to say, in general, "A fault in the heart is a fault in the body," without having a particular heart or body in mind, and it makes sense to reason over classes defined that way. For other domains, most notably manufacturing, it is more common for parts to exist outside of some whole, and so it may not generally be true that a fault in an engine is a fault in a car (if the engine is not in a car), just as it may not be generally true that an engine is a car part. In these cases, the capability to reason over classes may not be that useful, and again the existential restriction on the direct properties may not make sense.

Representation Pattern 4: Reflexive parts

1. Extend ontology with classes that approximate the fact that a whole is often considered

part of itself (reflexivity).

Examples

Classically, the part of relation is reflexive, that is it includes the thing itself, e.g. that a "car is a part of a car".   OWL does not have any built-in primitives for reflexivity (as it does for transitivity and inverses), but as shown above, we can use a pattern in defining classes to approximate this by combining the class with the class of its parts. It is sometimes convenient to define a "..._Reflexive" class for each item, e.g.:

```
    CarPart_reflexive = Car OR CarPart
```

In N3:

```
    CarPart_reflexive
          a       owl:Class ;
          owl:equivalentClass
                  [ a       owl:Class ;
                    owl:unionOf (Car [ a       owl:Restriction ;
                                owl:onProperty partOf ;
                                owl:someValuesFrom Car
                              ])
                  ] .
```

When reflexive parts are defined, it simplifies the definition of faults (in fact, this is often used as a logical argument for why the partOf relation is reflexive):

```
    Fault_in_car = Fault AND hasLocus someValuesFrom(CarPart_reflexive)
```

In N3:

```
    Fault_in_car
      a  owl:Class ;
      owl:equivalentClass
        [ a  owl:Class ;
        owl:intersectionOf (Fault [ a  owl:Restriction ;
           owl:onProperty hasLocus ;
           owl:someValuesFrom CarPart_reflexive
           ])
        ].
```

Discussion

Logically these classes do not give us reflexivity at all, a reflexive property is one that holds between an object and itself, not between an object and something in the same class. It is not possible in OWL to state such a restriction or inference, however, and when reasoning only over the classes and properties in an ontology this will suffice.

OWL code for examples

[N3] [RDF-Abrev] [RDF/XML]

Considerations

- Transitive properties should not be functional or inverse functional. It is not illegal, but it makes no sense, since the whole point is that transitive properties link chains of entitites together. Hence if there is only one value, there is no point in a property being transitive; and if there is a point there will be more than one value. In practice, almost any ontology containing a functional or inverse transitive relation will be unsatisfiable.
- It is possible to approximate a tree by making the `partOf_directly` subproperty non-transitive and functional, with a local range restriction (owl:allValuesFrom) to the "next larger" class of parts.
- If a classifier is to be used, then ontologies containing both `partOf` and `hasPart` will rarely scale beyond a few tens of classes. If a classifier is not to be used, then this need not be considered.
- The patterns described here can be generalised to work with individuals. This will be covered in a further note.

Additional Background

Other relations that follow the same pattern as faults

A number of other relations follow the same pattern as faults, e.g. "Repairs on a part are kinds of repairs on the whole". However, not all relations follow this pattern, e.g. "Purchase of a part is *not* purchase of the whole"

Relation to clasic Mereology

The classic study of parts and wholes, mereology, has three axioms: the part-of relation is

- Transitive - "parts of parts are parts of the whole" - If A is part of B and B is part of C, then A is part of C
- Reflexive - "Everything is part of itself" - A is part of A
- Antisymmetric - "Nothing is a part of its parts" - if A is part of B and A != B then B is not part of A.

OWL does have built-in primitives for antisymmetric or reflexive properties, nor is there any ~~work around~~ for them. In most cases this causes no problems, but it does mean that if you create a cycle in the part-of hierarchy (usually by accident) it will go unnoticed by the classifier (although it may cause the classifier to run forever.)

Furthermore, in mereology, since everything is a part of itself, we have to define "proper parts" as "parts not equal to the whole". Whereas in OWL we have to do the reverse: i.e. define "parts" (analogous to "proper parts") and then define "reflexive parts" in terms of "parts".

Relations that are not simple part-whole relations in the sense above

There are a number of relations easily confused with part-whole relations. Interested readers should consult [Flavours of part of]. However, a brief list includes:

- Containment- the fact that I am contained in my room does not mean that I am part of my room
- Membership-as flocks of geese and committees. Membership is not transitive. For example, the goose's leg is part of the goose but not part of the flock of geese. Slightly more awkwardly, even though we often talk of members of a committee being "part of a committee", being a member of a subcommittee that is part of a committee may, or may not, confer membership in the committee as a whole. Admittedly, whether membership is a part-whole relation is subject to debate.
- Connections and branches-That the lamp is connected to the main electricity system does not make it part of that system. Similarly, the tributary is not part of the river, rather a brach of the river. If we want to talk about parts, we usually speak of the "river system".
- Constituents- more controversially, many ontologists distiguish between the relation between clay and a statue made of clay - the clay "constitutes" or "is a constituent of" the statue, rather than being part of the statue in the same sense that the arm or leg is part of the statue. At the very least, there are a set of different issues involved in this relationship that are beyond the scope of this document.
- subClassOf- As discussed in Pattern 3, being a part of something is not the same as being a subclass of it.

More on `partOf` and `hasPart`

In some contexts it is "more universal" to use `partOf,` in others to use `hasPart.` For example, all cars have wheels, but not all wheels are parts of cars. On the other hand, all leaves are parts of plants (at least at some time), but not all plants have leaves. The inability of existing classifiers to cope with ontologies mixing `partOf` and `hasPart` is a significant limitation. In the usual case where `partOf` is used, the best option may then be not to enter a saying that "all wheels are parts of cars" but rather to define notions of as "Wheel of a car".

```
Wheel_of_car
   a   owl:Class ;
   owl:equivalentClass
    [a owl:Class ;
    owl:intersectionOf (Wheel [a owl:Restriction ;
       owl:onProperty partOf ;
       owl:someValuesFrom Car ])
    ]).
```

By defintion, all `Wheel_of_cars` are parts of cars, and any `Wheel` that is a part of a car is a `Wheel_of_car.` This is a work around, and not ideal, but at least all statements in the ontology are logically true.

Flavours of part-whole relations

Many authors discuss different subtypes of of the part_of relation. In most cases these can be represented as subproperties of `partOf,` but the various flavours of part-whole relation are beyond the scope of this note. See [Flavours of part of].

Notes

[1] This mechanism is a variant of the SEP triple approach first suggested by Hahn and Schulz : Hahn, U., Schulz, S. and Romacker, M. Part-whole reasoning: a case study in medical ontology engineering. *IEEE Intelligent Systems and their Applications*, 14 (5). 59-67

References

[Specified Values]
Representing Specified Values in OWL: "value partitions" and "value sets", Alan Rector, Editor, W3C Working Draft, 3 August 2004, http://www.w3.org/TR/swbp-specified-values/ .

[OWL Overview]
OWL Web Ontology Language Overview, Deborah L. McGuinness and Frank van Harmelen, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-owl-features-20040210/ . Latest version available at http://www.w3.org/TR/owl-features/ .

[OWL Guide]
OWL Web Ontology Language Guide, Michael K. Smith, Chris Welty, and Deborah L. McGuinness, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-owl-guide-20040210/ . Latest version available at http://www.w3.org/TR/owl-guide/ .

[OWL Semantics and Abstract Syntax]
OWL Web Ontology Language Semantics and Abstract Syntax, Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-owl-semantics-20040210/ . Latest version available at http://www.w3.org/TR/owl-semantics/ .

[RDF Primer]
RDF Primer, Frank Manola and Eric Miller, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-primer-20040210/ . Latest version available at http://www.w3.org/TR/rdf-primer/ .

[RDF Semantics]
RDF Semantics, Pat Hayes, Editor, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-mt-20040210/ . Latest version available at http://www.w3.

org/TR/rdf-mt/ .

[RDF Vocabulary]

RDF Vocabulary Description Language 1.0: RDF Schema, Dan Brickley and R. V. Guha, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/ . Latest version available at http://www.w3.org/TR/rdf-schema/ .

[Flavours of part of]

Odell, J.J. Six different kinds of composition. *Journal of Object Oriented Programming*, 5 (8). 10-15.
Winston, M., Chaffin, R. and Hermann, D. A taxonomy of part-whole relations. *Cognitive Science*, 11. 417-444
Artale, A., Franconi, E. and Pazzi, L. Part-whole relations in object-centered systems: An overview. Data and Knowledge Engineering, 20. 347-383

Changes

- renamed relations in examples (need to update OWL source files) using more camelCase.
- Cleaned up formatting and made pattern sections structurally consistent with each other.
- cleaned up partonomy, removed cylinder, made accurate wrt cars
- added a discussion section for each pattern that mentions potential problems