ISO 15926 Part 12 ontology: Examples for DL profile

johanw

Id:

Contents

1 What is this?

This document describes usage examples for the *DL profile* ontology version of ISO 15926 part 12.

2 Definitions

2.1 Declarations

2.1.1 Prefixes

```
## Prefixes
Prefix: lci: <http://standards.iso.org/iso/15926/>
# note, not same as the the namespace of the CD, which looks inappropriate
Prefix: ex: <http://example.org/ex/>
Prefix: rdl: <http://example.org/rdl/>
Prefix: owl: <http://www.w3.org/2002/07/owl#>
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: xml: <http://www.w3.org/XML/1998/namespace>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Prefix: skos: <http://www.w3.org/2004/02/skos/core#>
Prefix: pav: <http://purl.org/pav/>
Prefix: foaf: <http://xmlns.com/foaf/0.1/>
```

2.1.2 Ontology

```
## Ontology declaration

Ontology: <http://standards.iso.org/iso/15926/-12/tech/ontology/examples-DL-profile>
    Annotations: rdfs:label "Modelling examples for ISO 15926-12, DL profile",
      owl:versionInfo "$Date: started 2016-08-04$",
      rdfs:comment "This ontology contains examples of modelling patterns for the DL profile of ISO 15926-12."
    Import: <http://standards.iso.org/iso/15926/-12/tech/ontology/DL-profile>
```

2.2 Classes

Class details

2.2.1 *lci:Activity*

. rdl:NailDriving

rdfs:comment An activity, the driving of a nail into a material.

2.2.2 *lci:Function*

foaf:depiction `http://mimir.gitlab.io/iso15926-12/hammer-function.png`

2.2.3 *lci:PhysicalObject*

. lci:InanimatePhysicalObject

. . rdl:Artefact

. . . rdl:Hammer

. . . . rdl:BigHammer

Class: rdl:BigHammer
  SubClassOf: rdl:hasMass some (lci:qualityQuantifiedAs
    some (lci:datumUOM value rdl:kilogram and lci:datumValue some xsd:float[> 1]))

2.2.4 *lci:ScalarQuantityDatum*

. rdl:MassMeasurementDatum

. rdl:PressureMeasurementDatum

. rdl:TemperatureMeasurementDatum

2.2.5 *lci:PhysicalQuantity*

. rdl:Mass

. rdl:Pressure

. rdl:Temperature

2.2.6 *lci:Role*

2.2.7 *lci:Scale*

We introduce some example individuals to represent scales (units of measure): `kilogram`, `pascal`, bar, `kelvin`, `celsius`.

Individual: rdl:kilogram
  Types: lci:Scale
Individual: rdl:pascal
  Types: lci:Scale
Individual: rdl:bar
  Types: lci:Scale
Individual: rdl:kelvin
  Types: lci:Scale
Individual: rdl:celsius
  Types: lci:Scale

2.3 Object relations (object properties)

Object property details

2.3.1 *lci:hasQuality*

. rdl:hasPhysicalQuantity

. . rdl:hasMass

. . rdl:hasPressure

. . rdl:hasTemperature

2.3.2 *lci:qualityQuantifiedAs*

. rdl:qualityMeasuredAs

2.3.3 *rdl:hasMassMeasurement*

rdfs:comment This relation demonstrates the use of OWL property chains.

```
ObjectProperty: rdl:hasMassMeasurement
  SubPropertyChain: rdl:hasMass o rdl:qualityMeasuredAs
```

2.3.4 *rdl:hasParticipant*

. rdl:hasAgent

```
ObjectProperty: rdl:hasAgent
  InverseOf: rdl:agentIn
```

2.3.5 *lci:participantIn*

. is tool in (rdl:toolIn)

rdfs:comment This relation is to express participation in an Activity as a tool.

rdfs:comment This relation is used in examples with Hammer individual to illustrate how functional realization is linked to the right kind of participation in (nail-driving) activities.

. rdl:agentIn

rdfs:comment On the 'has_agent' relation, see e.g. p. 11 of «Relations in biomedical ontologies», Smith et al. 2005.

```
ObjectProperty: rdl:agentIn
  Domain: lci:Person or lci:Organisation
  Range: lci:Activity
```

2.3.6 *lci:hasFunction*

2.3.7 *lci:hasRole*

2.3.8 *lci:realizedIn*

rdfs:comment Inspired by BFO's «realized in» (BFO_0000054)

```
ObjectProperty: lci:realizedIn
  Domain: lci:Function
  Range: lci:Activity
```

2.3.9 *lci:roleOf*

2.3.10 *rdl:functionalIn*

rdfs:comment A shortcut relation to represent a chain of «has function» and «realized in».

```
ObjectProperty: rdl:functionalIn
  SubPropertyChain: lci:hasFunction o lci:realizedIn
```

2.4 Data relations (data properties)

```
## Data property details
```

2.4.1 *lci:qualityQuantityValue*

. rdl:hasMass_in_kilogram

```
DataProperty: rdl:hasMass_in_kilogram
  Domain: lci:PhysicalObject
```

2.4.2 *rdl:datumTimestamp*

rdfs:comment Example relation for recording the time a measurement is taken.

2.5 Annotation relations (annotation properties)

```
## Annotation property details
```

2.5.1 *lci:tplUOM*

2.5.2 *lci:tplQuality*

2.5.3 *lci:tplQuantification*

2.5.4 *foaf:depiction*

rdfs:comment This FOAF annotation property is for providing illustrations. In the current context, the main interest is in showing modelling diagrams. With a valid image URL, a thumbnail will be displayed in the Protégé editor.

3 Examples

```
##
## Test data
##
```

3.1 Physical qualities, quantified

3.1.1 A big hammer and a small hammer

The following declarations describe `hbig` and `hsmall` as individuals in the `Hammer` class, with measured weights of 4.7 and 0.3 kg, respectively.

```
Individual: ex:hbig
  Types: rdl:Hammer
  Annotations: foaf:depiction <http://mimir.gitlab.io/iso15926-12/big-hammer.png>
  Facts: rdl:hasMass ex:hbig_mass

Individual: ex:hbig_mass
  Types: rdl:Mass
```

Facts: rdl:qualityMeasuredAs ex:hbig_mass_datum

Individual: ex:hbig_mass_datum
Types: rdl:MassMeasurementDatum
Facts: lci:datumUOM rdl:kilogram, lci:datumValue 4.7f

Individual: ex:hsmall
Types: rdl:Hammer
Facts: rdl:hasMass ex:hsmall_mass

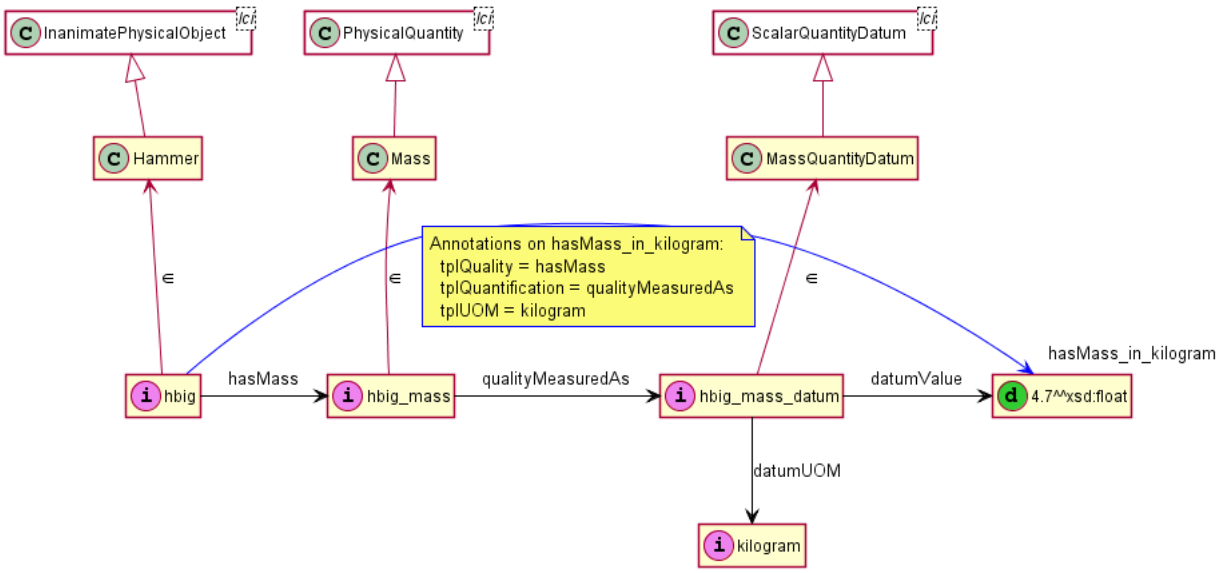
Individual: ex:hsmall_mass
Types: rdl:Mass
Facts: rdl:qualityMeasuredAs ex:hsmall_mass_datum

Individual: ex:hsmall_mass_datum
Types: rdl:MassMeasurementDatum
Facts: lci:datumUOM rdl:kilogram, lci:datumValue .3f

The following diagram illustrates the main points of the *hammer* mass representation.

The «shortcut» relation `rdl:hasMass_in_kilogram` is emphasised with a blue arrow. For precision, this relation should be interpreted in terms of the *template* terminology, as an instance of a shortcut pattern for quantified physical properties. The relation may then be annotated with pointers to its intended template role fillers, i.e., `hasMass`, `qualityMeasuredAs`, and `kilogram`, to allow for retrieval of the relevant information using SPARQL, without interference with the DL reasoning requirements.

(We could add *InanimatePhysicalObject hasMass only Mass*, and *Mass qualityMeasuredAs only MassQuantityDatum* to the diagram?)



3.1.2 SWRL rule to infer *shortcut* relations

The detailed model of physical qualities illustrated above allows for several degrees of freedom. While the relation `hasMass` will typically be characterized as functional, there may be many ways to quantify the mass of an individual, by different measurements or estimates as well as with conventional declarations. This is more freedom than is useful in practical cases where we assume our (mass, etc.) data is reliable and where requirements can be directly matched to them.

Shortcut relations that gloss over the possible variations are suitable for contexts that allow for simplifying assumptions. The following SWRL rule allows for shortcut relationships to be inferred from the basic model. This kind of rule is supported by the HermiT and Pellet DL reasoners.

Rule: rdl:hasMass(?x, ?y), rdl:qualityMeasuredAs(?y, ?z), lci:datumUOM(?z, rdl:kilogram), lci:datumValue(?z,?u) –>
→rdl:hasMass_in_kilogram(?x, ?u)

Ideally, shortcuts should be supported by a formal framework that gives them precise characterization. The *template* approach of ISO 15926-7 can be adapted for the DL profile to provide a robust implementation that also works with existing semantic technology solutions.

3.2 STARTED Functions

3.2.1 Background: introducing functions to the ontology

The DL profile doesn't include the ISO 15926-2 entity type «functional physical object». This is partly due to the lack of clarity in how the entity type is supposed to be used: for instance, the defining phrase «A FunctionalPhysicalObject has functional, rather than material, continuity as its basis for identity» itself *uses* the word «function», and is no help in explaining what functions are.

Most of the physical things that we wish to describe in a store of industrial data will have a function – they are there to *do something*. This includes structural elements of a factory, equipment, and instruments. We could almost identify the notions of «physical artefact», *made* objects, and «functional physical object». Physical things that do *not* have a function in the sense of being made for a purpose include streams of liquid, volumes of gas, and other amounts of raw materials.

(It may be noted that many Non-physical things, such as a regulations, acts, and documents, also have functions. It would therefore make sense to include «functional object» in the upper ontology as a superclass to «functional *physical* object».)

Some objects have functions that are simple, such as a nut serving to secure a bolt in its place, while others have complex and generic functions, such as a control mechanism or a robot arm. The functions may be documented, and it's common to talk about functions changing over time. Observations like these indicate that it makes sense to represent functions as individuals in their own right: there is reason to include «function» as an upper ontology class.

Because ISO 15926 provides little detail on the ontology of functions, we need to look elsewhere for guidance on how to represent them. A useful account, which inspires the following proposal, is contained in the paper *Function, Role, and Disposition in Basic Formal Ontology*.

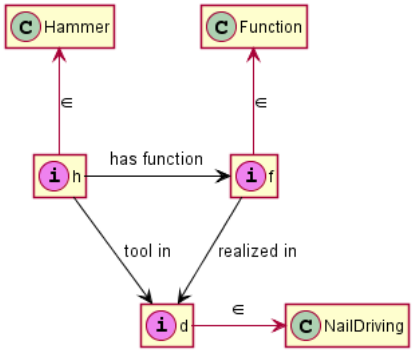
Assume for simplicity that a *hammer* has a single function: to drive nails. Following Arp and Smith, this function can be *realized* in certain *processes* (in ISO 15926, processes will be instances of the Activity type), «in virtue of» the physical make-up of the hammer. The hammer was shaped with shaft and hammerhead precisely in order to serve its nail-driving function. We can add to this description that when the hammer is acting functionally in a process, it must participate in a suitably active role, for instance named a *tool* role. Passive participation in a process will not be an expression of function: for instance, if a nail is driven into the wooden shaft of a hammer x, then x is indeed a participant in the nail-driving activity, but obviously this doesn't count as a realization of x's function. This means we need to pay attention which kinds of participation count as realization of function.

3.2.2 Modelling pattern

A description of function could look as follows: *A Hammer's function is realized precisely when it is used as a tool to drive a nail.* While this may appear trivial, the shape of the sentence can guide us to a modelling pattern for ontology. Here is a slightly more stringent form:

A Hammer x has a function that is only realized in Nail-driving activities where x has the tool role.

A diagram shows the basic pattern. A hammer h that is the tool used in nail-driving activity d, and the function of h is realized in the nail-driving.



We add this example to the ontology, using `hbig` as an example.

```
Individual: ex:nailing # "d" in the diagram
  Types: rdl:NailDriving
  Annotations: rdfs:label "An hbig nail driving"

Individual: ex:hbig_f # "f" in the diagram
  Annotations: rdfs:label "Function of hbig" ,
   foaf:depiction <http://mimir.gitlab.io/iso15926-12/hammer_function.png>
  Types: lci:Function
  Facts: lci:realizedIn ex:nailing

Individual: ex:hbig
  Facts: lci:hasFunction ex:hbig_f, rdl:toolIn ex:nailing
```

The point that hammer functions are only realized in nail driving processes where the hammer is *active* as a tool is clearly important. However, the inbound pattern of arrows from *h* to *d* is one that can not be lifted into description logic class constraints – it is an example of a limitation that ontology designers frequently have to work around. This means we can't fully capture the relationship between functions and the ways they can be realized using OWL.

The chain of relationships *has_function o realized_in* captures a relation «serves its function in process». Name this relation *functionalIn*. We then wish to say, if a hammer is functional in a process, then it participates as a tool in that process. A SWRL rule can be readily provided for this constraint. Some reasoners, including *HermiT* and *Pellet*, will in fact take rules in this form into account during reasoning.

```
Rule: rdl:Hammer(?x), rdl:functionalIn(?x,?y) -> rdl:toolIn(?x,?y)
```

3.2.3 A registry of functions in reference data

With an ontology that supports explicit talk about functions, we obtain new abilities to characterize industrial artefacts and processes. Here are some examples.

1. The relation *functionalIn* can be used to connect classes of equipment to the classes of activities they are intended to serve. For example, a pump is functional in *pumping* activities, and not (to borrow an example from the Part 2 text) in anchoring.

2. Functions can be used to express requirements. Consider two kinds of hammers, the claw hammer and the ball-peen hammer. The claw hammer has a nail-extraction function which the ball-peen hammer lacks. If you need tools to both pound nails and extract them again, this may be expressed as a *functional requirement*: you need something that has both functions, and either a claw hammer or a set of ball-peen hammer and pincers will serve. Extrapolating from this, functions may be used to characterize the *capabilities* of equipment, such as the capacity of a class of pumps, and a class of equipment can then be tested against a functional requirement.

3. When a function is represented as an individual, it can enter into explicit relations to other entities, such as documents that describe it

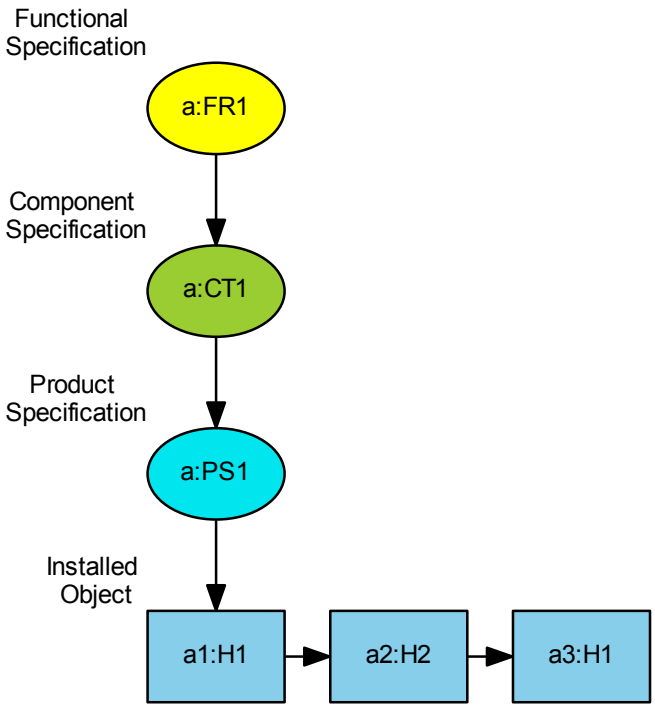
3.3 STARTED Requirements versus solutions

3.3.1 Targets

- use reasoning to find out whether an individual is compatible with a requirement stated as a class constraint
  - typically, we want to avoid equivalence declarations for classes because the reasoning is too expensive when there's a large number of classes
- for complex designs with many individual parts, use *substitution* to check whether a graph of individuals is
  - compatible with the requirements
  - complete, in the sense of instantiating every required part
- support representation of replaceable parts

3.3.2 Schematic representation

The following drawing is intended to be a simplified representation of «Requirements Network Structure using IEC 81346», as modelled in ontology. Let *a* be an individual component in a design, and *FR1*, *CT1*, *PS1*, and so forth stand for classes at levels of detail that match functional, component, and product specifications. The downward sequence represents phases of increasingly detailed requirements. We write *a:F* for, «*a* is an *F*», so that the yellow node labelled *a:FR1* means, the functional specification lays down a requirement for *a* to be of type *FR1*. At the bottom of the diagram we have *a1*, *a2*, and so forth, representing physical individuals that are installed to fill the role specified for *a*, with *a1* installed first, then replaced by *a2*, etc. These physical individuals have types *H1* and *H2*, intended to represent types of purchased items.



The modelling example described in the following paragraphs should demonstrate two aspects.

1. The sequence of replaced parts is represented in a form suitable for storage and retrieval.

2. We can use automated reasoning to check whether a replaceable part conforms to the design requirements.

3.3.3 Industry classes for the example: Electric motors

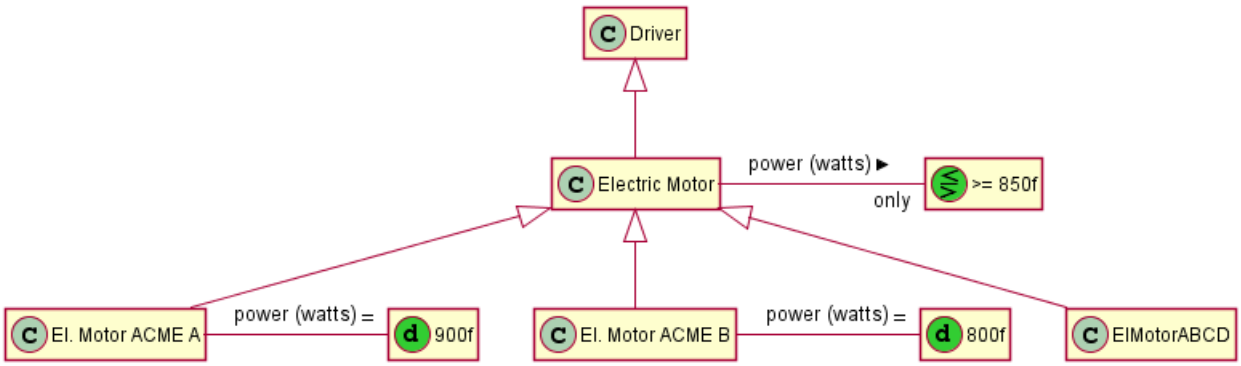
We will assume the design describes an electric motor, with classes for: Driver for the functional specification, Electric Motor for component, Electric Motor ABCD for product, and ACME A and ACME B as examples of product types (models made by ACME).

The assumption is that the design proceeds from very generic requirements to a detailed specification, where the products ACME A and ACME B are finally chosen for installation. As an example of a requirement, we say the Component specification requires at least 850 watts of output power. The ACME A model delivers 900 watts and is suitable, but ACME B delivers only 800 watts, as an example of a non-conformant choice.

We declare the ontology resources – five classes and one data property.

```
DataProperty: rdl:DriverPower_watts
  Domain: rdl:Driver
Class: rdl:Driver
  SubClassOf: rdl:Artefact
  Annotations: rdfs:label "Driver ... (FR1)"
Class: rdl:ElMotor
  SubClassOf: rdl:Driver
  Annotations: rdfs:label "Electric Motor ... (CT1)"
Class: rdl:ElMotorABCD
  SubClassOf: rdl:ElMotor, rdl:DriverPower_watts only xsd:float[>= 850]
  Annotations: rdfs:label "El Motor ABCD ... (PS1)"
Class: rdl:ElMotorACME_A
  SubClassOf: rdl:ElMotor, rdl:DriverPower_watts value 900f
  Annotations: rdfs:label "El Motor ACME A (PS1A)"
Class: rdl:ElMotorACME_B
  SubClassOf: rdl:ElMotor, rdl:DriverPower_watts value 800f
  Annotations: rdfs:label "El Motor ACME B (PS1B)"
```

The following diagram illustrates how the main classes are related. Note that the requirements at each stage of specification (FR, CT, and PS) will in real cases involve many more classes than are shown here.

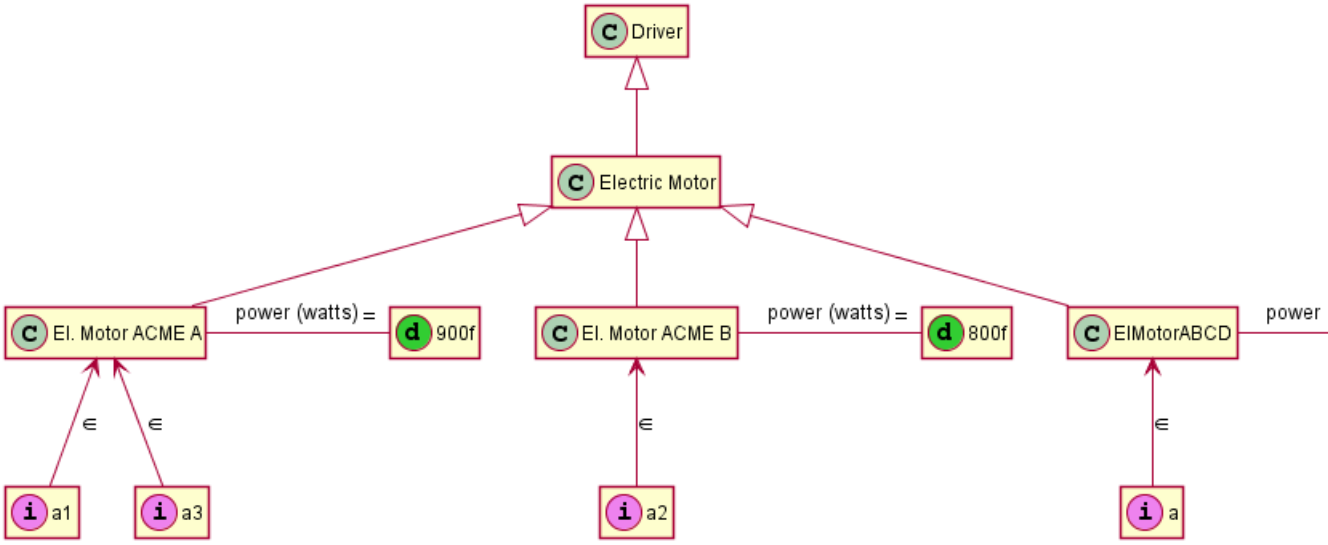


3.3.4 Individuals for the example: design and replaceable parts

We will assume *a* is the individual that represents our motor during design, and that *a1*, *a2*, and *a3* are replaceable individuals installed to fill the role of *a* in the assembly.

```
Individual: ex:a
  Annotations: rdfs:label "El Motor a"
  Types: rdl:Driver, rdl:ElMotor, rdl:ElMotorABCD
Individual: ex:a1
  Annotations: rdfs:label "a1 ACME A"
```

Types: rdl:ElMotorACME_A
Individual: ex:a2
Annotations: rdfs:label "a2 ACME B"
Types: rdl:ElMotorACME_B
Individual: ex:a3
Annotations: rdfs:label "a3 ACME A"
Types: rdl:ElMotorACME_A



3.3.5 Design and installed parts

The design model provides requirements for objects like *a*. We postulate that for each object in the design, there is a type of *role* that can be filled by installed parts.

ObjectProperty: rdl:role_of_design
Annotations: rdfs:label "role as design artefact"
Domain: lci:Role
Range: rdl:Artefact
Class: ex:DesignRole_a
Annotations: rdfs:label "Design role a"
SubClassOf: lci:Role, rdl:role_of_design value ex:a

A replaceable part will occupy a designed role for a limited time only. We introduce data properties to attribute start and end times to the individual roles. (These are short-cuts. In a detailed account of designs and parts, these simple relations should be supported by modelling to reflect that the roles obtain as a result of installation and removal processes.)

DataProperty: rdl:role_start
Domain: lci:Role
Range: xsd:dateTime
DataProperty: rdl:role_end
Domain: lci:Role
Range: xsd:dateTime

Each part that is installed as *a* takes on the role of *a* in the design, but at different times.

Individual: ex:a1
Facts: lci:hasRole ex:a1_as_a
Individual: ex:a1_as_a
Types: ex:DesignRole_a
Facts: rdl:role_start "2016−01−01T10:00:00Z"^^xsd:dateTime, rdl:role_end "2016−01−10T10:00:00Z"^^xsd:dateTime

Individual: ex:a2
Facts: lci:hasRole ex:a2_as_a
Individual: ex:a2_as_a

```
  Types: ex:DesignRole_a
  Facts: rdl:role_start "2016−02−01T10:00:00Z"^^xsd:dateTime, rdl:role_end "2016−02−10T10:00:00Z"^^xsd:dateTime

Individual: ex:a3
  Facts: lci:hasRole ex:a3_as_a
Individual: ex:a3_as_a
  Types: ex:DesignRole_a
  Facts: rdl:role_start "2016−03−01T10:00:00Z"^^xsd:dateTime, rdl:role_end "2016−03−10T10:00:00Z"^^xsd:dateTime
```

3.3.6 Testing for conformance: Substitution

We want to use automated reasoning to check whether the requirements laid down in a design are satisfied by the installed parts. This can be done by selecting «concrete» individuals and substituting them for the targeted design objects. For the example given, we substitute the replaceable parts *a1 ACME A*, *a2 ACME B*, and *a3 ACME A* for the design object *El Motor a*,

The effect of substitution is that we *combine* all the requirements of the design with all the characteristics of the product specimens. If there is a conflict, the reasoner will discover an inconsistency. In complex cases, we benefit from the reasoner's ability to find not only obvious clashes, but also any implicit conflicts that may be very difficult to identify without the help of automated reasoning.

Testing for consistency

To substitute product individuals for design objects, we for instance use SPARQL «construct» queries. A query might select the individuals with «installed-as» roles assigned for a given point in time, and replace any design objects with the installed objects. This transforms the combination of design model and installed-object history into a snapshot representing the asset.

We can simulate substitution in an ontology tool like Protégé by manually adding identity statements in the user interface: «the product named *a1* is Same Individual As the design object *a*». The OWL primitive for identity «=» is owl:sameAs; in Manchester Syntax it is expressed as follows.

```
Individual: ex:a
  SameAs: ex:a1
```

For the example, we find that

- Identifying *a* with *a1* or *a3*, no inconsistency is reported
- Identifying *a* with *a2*, we obtain an inconsistency. This is due to the design requirement that *a* delivers at least 850 watts of output power (rdl:DriverPower_watts). The individual *a2* is of type *El Motor ACME B*, which delivers only 800 watts.

Testing for completeness

The consistency check is only one of two basic tests of conformance with a design. We also need to check whether a solution is *complete* in the sense of providing a product individual for each object in the design. In other words, we need to check whether there are missing pieces.

This kind of «referential integrity» test can not be implemented using OWL reasoning. A SPARQL «ASK» query is however readily constructed to check whether a proposed solution provides a product for every design object.

3.3.7 TODO Rewrite as extended example: «Stream 101» examples

The following examples are taken from a document draft, «The Part 12 DL profile», by Andreas Nakkerud in discussion with David Leal and Arild Waaler.

Model of stream with two measurements

The case describes a stream `s_101` with two quality measurements, of pressure and temperature.

In order to support this example with reference classes, we have introduced the following resources as example «reference library» content.

- classes `Pressure`, `Temperature`, `PressureMeasurementDatum`, `TemperatureMeasurementDatum`
- individuals `bar`, `celsius` (replacing `pascal`, `kelvin` from the source document for simplicity)
- relations `hasPressure`, `hasTemperature` for qualities, and `datumTimestamp`. (not needed: `has_pressure_measu` and `has_temperature_measurement`;)

```
Individual: ex:S_101
  Types: lci:Stream
  Facts: rdl:hasPressure ex:S_101_pressure, rdl:hasTemperature ex:S_101_temperature
Individual: ex:S_101_pressure
  Types: rdl:Pressure
  Facts: rdl:qualityMeasuredAs ex:pm_000001
Individual: ex:S_101_temperature
  Types: rdl:Temperature
  Facts: rdl:qualityMeasuredAs ex:tm_000001
Individual: ex:pm_000001
  Types: rdl:PressureMeasurementDatum
  Facts: lci:datumUOM rdl:bar, lci:datumValue 14.0f, rdl:datumTimestamp "2016-01-01T12:00:00Z"^^xsd:dateTime
Individual: ex:tm_000001
  Types: rdl:TemperatureMeasurementDatum
  Facts: lci:datumUOM rdl:celsius, lci:datumValue 700.0f, rdl:datumTimestamp
      →"2016-01-01T12:01:00Z"^^xsd:dateTime
```

Stream in critical conditions

We define the class of pressure measurements of at least 15 bar, and «pressure critical stream» as a stream with a pressure in that range. (Note, the original document uses pascal as the unit of measure. That seems unnecessary here.)

```
Class: rdl:PressureMeasurementDatum_min_15_bar
  EquivalentTo: rdl:PressureMeasurementDatum
    and (lci:datumUOM value rdl:bar) and (lci:datumValue some xsd:float[>= 15])

Class: rdl:PressureCriticalStream
  EquivalentTo: lci:Stream and
    rdl:hasPressure some (rdl:qualityMeasuredAs some rdl:PressureMeasurementDatum_min_15_bar)
```

We define the class of temperature measurements of at least 850 celsius, and «temperature critical stream» as a stream with a temperature in that range. (Note, the original document uses kelvin as the unit of measure.)

```
Class: rdl:TemperatureMeasurementDatum_min_850_celsius
  EquivalentTo: rdl:TemperatureMeasurementDatum
    and (lci:datumUOM value rdl:celsius) and (lci:datumValue some xsd:float[>= 850])

Class: rdl:TemperatureCriticalStream
  EquivalentTo: lci:Stream and
    rdl:hasTemperature some (rdl:qualityMeasuredAs some rdl:TemperatureMeasurementDatum_min_850_celsius)
```

Either of these suffice for a «stream in critical condition» – or more precisely a stream that was, at some some unspecified time, in a critical condition.

```
Class: rdl:CriticalStream
  EquivalentTo: rdl:PressureCriticalStream or rdl:TemperatureCriticalStream
```

Now (still following the original text), assume any combination of >=700 celsius and >=13 bar is also a critical condition. This can't be captured in a single class definition because we don't have variable bindings to restrict the times that measurements are taken. We can however use SPARQL.

(Note. This query has a filter that assumes timestamps are *equal*, which is not actually the case for the sample data about `S_101` above (they differ by one minute). To get a match running the query in Protégé, which we use for discussion and demonstration, edit the timestamps to be equal. The query is left like this because the Protégé SPARQL support is very limited and doesn't support date arithmetic at all.)

```
PREFIX rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf−schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX lci: <http://standards.iso.org/iso/15926/>
PREFIX rdl: <http://example.org/rdl/>

select * {
{ ?s a rdl:CriticalStream }
union
{
  ?s a lci:Stream .
  ?s rdl:hasTemperature/rdl:qualityMeasuredAs
    [ lci:datumUOM rdl:celsius ; lci:datumValue ?temp ; rdl:datumTimestamp ?temp_ts ] ;
   rdl:hasPressure/rdl:qualityMeasuredAs
    [ lci:datumUOM rdl:bar ; lci:datumValue ?pres ; rdl:datumTimestamp ?pres_ts ] .
  filter ( ?temp_ts = ?pres_ts )
}}
```

A range of alternative approaches to representing critical conditions is available. One useful approach involves approximating a performance curve using disjunction of conjunctions of requirements. (TODO write this out.) This will work *in OWL itself* given that measurement data is aggregated appropriately so that they apply to adequately similar times, although typically not with great performance (disjunctions are expensive to compute). Another approach could employ SWRL «built-in» comparison operators, which are supported by some reasoners and rule engines.

3.4 TODO SKOS concepts to represent coding schemes and meta-data

Target: Describe how SKOS can be used with ISO 15926 to represent both ontology resources and various non-semantic terminology schemes as OWL individuals. This is primarily for use in managing meta-data and various annotations.

3.4.1 SKOS ontology, and adjustments for use in OWL

SKOS was published as a W3C recommendation in 2009. It is a schema that «provides a standard way to represent knowledge organization systems using the Resource Description Framework (RDF)» (SKOS homepage).

With a focus on terms rather classes in the logical semantics sense, the purpose of SKOS is expressly different from, and complementary to, that of ISO 15926. «SKOS is an area of work developing specifications and standards to support the use of knowledge organization systems (KOS) such as thesauri, classification schemes, subject heading systems and taxonomies within the framework of the Semantic Web» (ibid.). Because SKOS has wide usage, re-using with ISO 15926 can promote interoperability and exchange for terminological information. We describe here how this is possible with some minor adjustments.

The RDF version of SKOS is an OWL ontology that contains the following classes, in addition to a set of object properties and a single data property.

- Collection
  - 'Ordered Collection'

- Concept
- 'Concept Scheme'
- rdf:List

Unfortunately, 'Ordered Collection', with the accompanying object property 'has member list', depends on treating rdf:List as a class in the ontology. This is incompatible with OWL, and using rdf:List runs the risk of unpredictable behaviour with the OWL software libraries. It's however not essential to make use of ordered collections, as the main features of SKOS are still available without them.

Looking at the SKOS documentation, we find that Collection and Concept Scheme fit well as special kinds of ISO 15926 InformationObject. A suitable taxonomy of classes to include with ISO 15926 is therefore the following; skos:Concept is left as a top level (immediately subordinate to owl:Thing).

- skos:Concept
- lci:InformationObject
  - Collection
  - 'Concept Scheme'

3.4.2 What SKOS can bring

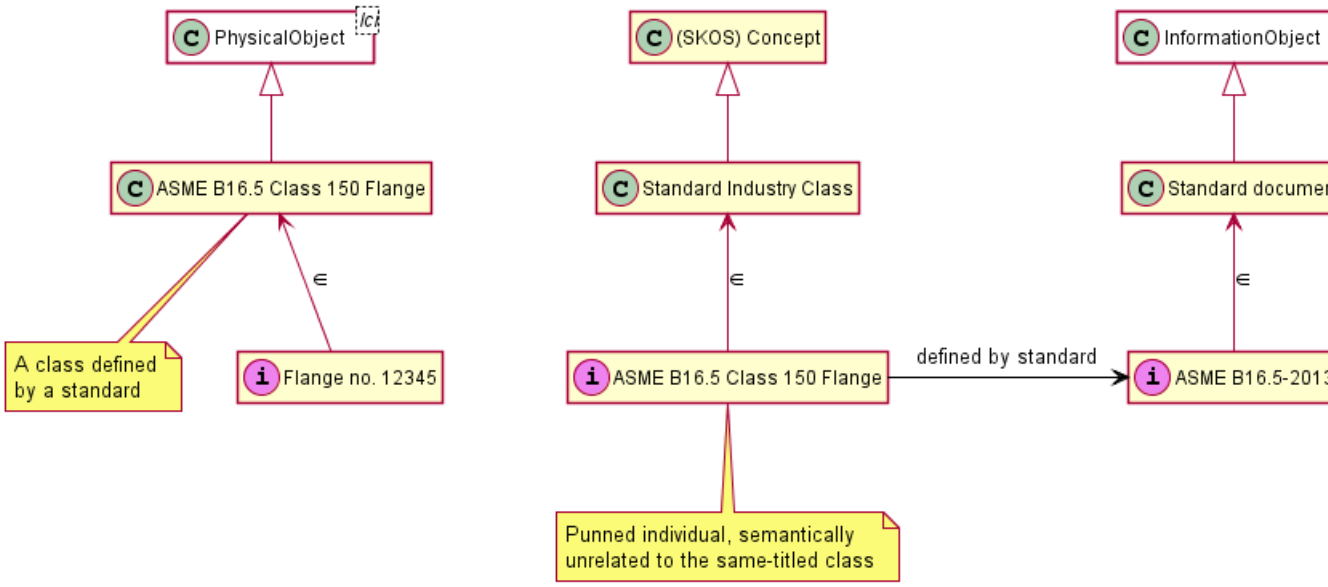
SKOS is useful whenever we need to manage terms, codes, and identifiers for which we have no straightforward categorization in terms of OWL categories. This applies to most existing *code lists*, as we find in abundance in any information system. A code list will typically not distinguish between terms that represent individuals from those that represent classes or relations – and so forth.

Any term in a non-ontological terminology can safely be treated as an OWL individual. Classifying a term as a skos:Concept makes it available in the ontology language, so it can enter into relationships to individuals in the ISO 15926 categories (such relationships should be then used with little commitment to precise interpretation).

In a different but related move, it is sometimes beneficial to make use of OWL's *punning* support and refer to classes as individuals. Here there is again no suitable first-order ISO 15926 entity ready to serve as classifier. We suggest that skos:Concept can be fruitfully applied also here. Subclasses of skos:Concept may be introduced to group classes in a way that is accessible to automated validation.

3.4.3 TODO Example

We show an example of referring from a punned class, represented as a SKOS Concept, to an industry standard that defines the class.



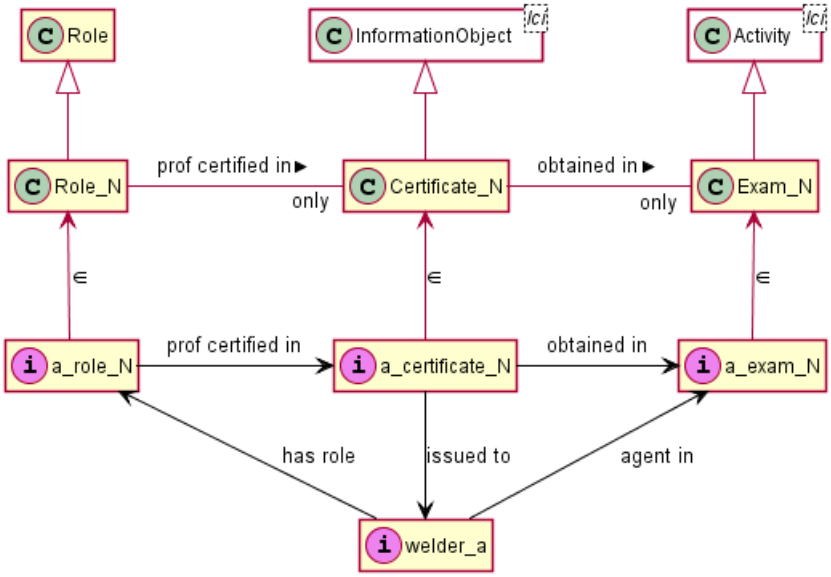
3.5 STARTED Roles and qualifications

Target: Describe how we can model roles that people and organisations take in activities, and the qualifications required to take various roles. Preferably, qualifications should be represented as requirements in a very similar way to the requirements on equipment etc.

3.5.1 STARTED Person with exam, certificate, qualified role

The following diagram illustrates how we can relate qualified roles to certificates and the exams that are required. We use the letter N to stand for a type of activity that requires qualification; as an arbitrary example, a type of *welding*, so that having a rdl:role_N role is to possess a certain welder's qualification.

Detailed production records are part of the documentation of process plants, and the types of individuals shown in this diagram will typically all be present in such records. That is to say, there is a realistic need for this degree of detail to be modelled.



OWL code to represent this diagram can be given as follows. *Notes*.

- We leave some classes out of the diagram to keep it compact: introducing rdl:Certificate, a kind of rdl:InformationObject, rdl:RestrictedProfessionalRole specializing Role, and rdl:Test specializing

Activity.

- rdl:obtainedIn, as a relation between certificates and examinations, deserves a generic superproperty with appropriate domain and range constraints, perhaps in upper ontology.

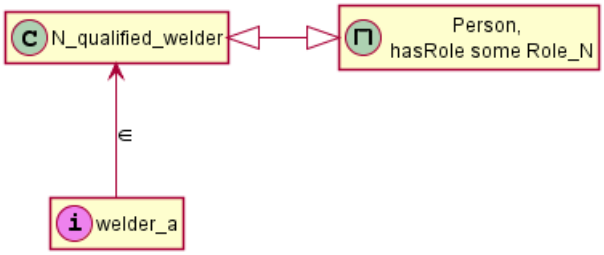
```
Class: rdl:RestrictedProfessionalRole
  SubClassOf: lci:Role
Class: rdl:Certificate
  SubClassOf: lci:InformationObject
Class: rdl:Test
  SubClassOf: lci:Activity
ObjectProperty: rdl:obtainedIn
  Domain: rdl:Certificate
  Range: rdl:Test
ObjectProperty: rdl:prof_certifiedIn
  Domain: rdl:RestrictedProfessionalRole
  Range: rdl:Certificate
ObjectProperty: rdl:issuedTo
  SubPropertyOf: lci:interests
Class: lci:Role_N
  SubClassOf: rdl:RestrictedProfessionalRole, rdl:prof_certifiedIn only rdl:Certificate_N
Class: rdl:Certificate_N
  SubClassOf: rdl:Certificate, rdl:obtainedIn only rdl:Exam_N
Individual: ex:welder_a
  Types: lci:Person
  Facts: lci:hasRole ex:a_role_N, rdl:agentIn ex:a_exam_N
Individual: ex:a_role_N
  Types: lci:Role_N
  Facts: rdl:prof_certifiedIn ex:a_certificate_N
Individual: ex:a_certificate_N
  Types: rdl:Certificate_N
  Facts: rdl:obtainedIn ex:a_exam_N
Individual: ex:a_exam_N
  Types: rdl:Exam_N
```

We can define a class of welder according to the role granted by a qualifying exam. (This follows the recommended pattern for BFO, where roles are considered more basic than classes that relate to qualifications.)

```
Class: rdl:N_qualified_welder
  EquivalentTo: lci:Person and lci:hasRole some lci:Role_N
```

For many purposes, a simple classification will suffice to express qualification, as shown in the following diagram. We can consider this a shortcut of the fully detailed pattern above.

Note the use of Manchester syntax to include an anonymous, complex class in the diagram. Is this acceptable?



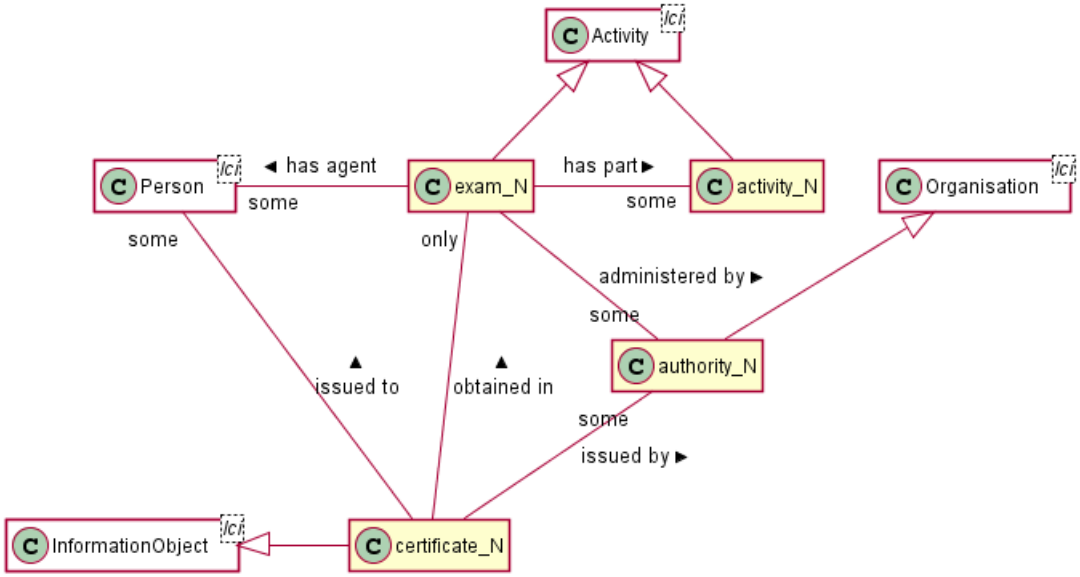
3.5.2 The exam – obtaining a certificate

Modelling task: Characterize an examination. The examinee has successfully completed a set task (an Activity) and is therefore granted a certificate. (We can *not* model the relationship between the examination activity and the fact, itself, that a qualification is true of the graduate, since we can't talk about facts.)

The focus here is on the exam activity itself, and the role of *agent* that the examinee has in the exam – a form of *participation*.

An exam can be represented in further detail. In the following diagram, we say that any «N exam» includes an «N» activity part, that the exam is administered by an authority, and that a certificate will be issued by an authority.

With minor modifications, this pattern should be applicable to various kinds of certifications, including for qualified suppliers and service providers; i.e., an audit is analogous to an exam.



OWL declarations follow. *Notes*.

- The *person* that sits an exam (is agent in the exam activity) should also be the one to whom the certificate is issued. This requires a «diamond» relationship which can't be captured in OWL classes.
- The authority that *administers* the exam doesn't need to be identical to the authority that *issues* the certificate.
- The class rdl:Certificate_N is given additional constraints here

```
ObjectProperty: rdl:administeredBy
  SubPropertyOf: lci:interests
ObjectProperty: rdl:issuedBy
  SubPropertyOf: lci:interests
Class: rdl:Authority_N
  SubClassOf: lci:Organisation
Class: rdl:Activity_N
  SubClassOf: lci:Activity
Class: rdl:Exam_N
  SubClassOf: lci:Activity, lci:hasPart some rdl:Activity_N,
    rdl:administeredBy some rdl:Authority_N, rdl:hasAgent some lci:Person
Class: rdl:Certificate_N
  SubClassOf: rdl:issuedBy some rdl:Authority_N,
    rdl:issuedTo some lci:Person
```

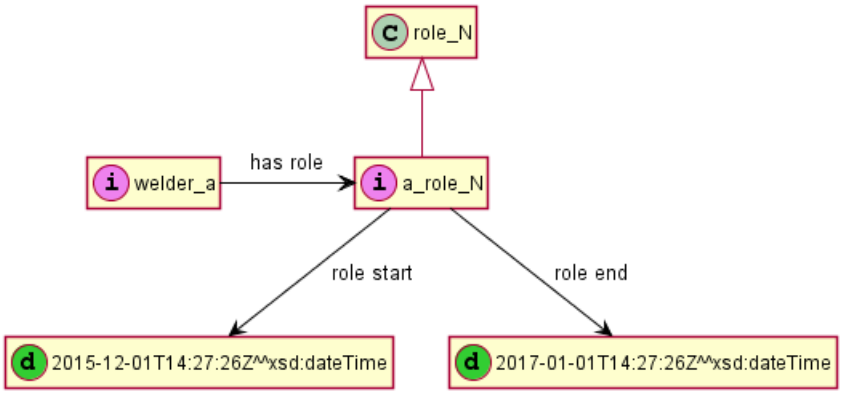
3.5.3 TODO Valid duration

Note. There's a W3C working group for temporal issues in OWL, including the various interactions of temporal intervals, at `https://www.w3.org/TR/owl-time/`. We should try to avoid requirements on the representation of time in Part 12 that can be in conflict with the future recommendations issued by this working group.

The following can be added as datatype properties with xsd:dateTime range.

- certificate issued date
- exam completed date
- role valid from–to date

The following figure illustrates the use of data properties to assign start and end times to a role, as for instance to say when the role is valid.



OWL declarations:

```
Individual: ex:a_role_N
  Facts: rdl:role_start "2015−12−01T14:27:26Z"^^xsd:dateTime,
    rdl:role_end "2017−01−01T14:27:26Z"^^xsd:dateTime
```

There is a straightforward way to use OWL data ranges for time period restrictions: we can check date values of instances against date ranges for classes. The following OWL code declares, using equivalence statements,

- RoleValid2016 as the class of roles that have start and end validity dates that cover the whole of 2016 (as given by role_start, role_end) ,
- N_qualified_welder_2016 as any N_qualified_welder qualified throughout 2016 (using RoleValid2016).

Note that this kind of use of EquivalentTo will often yield slow reasoning performance.

```
Class: lci:RoleValid2016
  EquivalentTo: lci:Role and
    (rdl:role_start some xsd:dateTime[<= 2016−01−01T00:00:00Z]) and
    (rdl:role_end some xsd:dateTime[>= 2017−01−01T00:00:00Z])
Class: rdl:N_qualified_welder_2016
  EquivalentTo: rdl:N_qualified_welder and
    lci:hasRole some (lci:Role_N and lci:RoleValid2016)
```

With an equivalence definition as given here, the reasoner (tested with HermiT and Pellet) will classify our example individual ex:welder_a as an rdl:N_qualified_welder_2016.

```
Individual: ex:welder_a
  Types: rdl:N_qualified_welder_2016 # inferred by reasoner
```

We see in this example that it's reasonably simple to introduce classes with time period restrictions, and then let the reasoner figure out whether qualifications on individuals are valid during these periods.

An OWL limitation

Here is a typical case:

- Activity *n* happens on March 20
- Professional *a* has a certificate for doing *n* type tasks, given in start and end dates for the relevant role (or similar: with a different modelling, the dates may be said to apply to the certificate; etc.)
- We would like to have the reasoner figure our whether *a* is qualified to do *n* on March 20

Unfortunately, the reasoning in such a case involves comparing dates – the particular dates of qualifications with the particular date of the activity. This can not be done in OWL itself. Some approaches could be

- using SWRL *extension* rules, such as the SWRL *built-ins* swrlb:lessThan (for '<') and swrlb:greaterThan (see `http://www.daml.org/swrl/proposal/builtins.html`),
- using SPARQL ASK queries,
- by introducing, ad-hoc, a class to represent the «date range during which professional *a* is qualified».

Here is a sketch of a SWRL rule to test qualified dates. Let `x < y` stand for `swrlb:lessThan(?x, ?y)`.

```
Rule: activity(x), at−date(x, t),
  agent(y), qualification−start(y, t1), qualification−end(y, t2),
  t1 < t, t < t2 −> can−do(y, x)
```

Each of these has disadvantages.

- SWRL extensions, including the «built-ins», are non-logical. They are supported by the Pellet reasoner and can be readily used with rule engines, but they are not supported by, e.g., HermiT.
- SPARQL queries can not be straightforwardly included in an ontology. See, however, SPIN for a way to integrate queries (`http://spinrdf.org/`), supported by TopQuadrant products as well as the RDF4J (formerly Sesame) framework (`http://rdf4j.org/`).
- Adding ad-hoc classes pollutes the ontology with countless classes that may impact performance.

3.5.4 TODO Being qualified for a type of activities

Modelling task: Characterize qualification for a category of tasks, like '*a* is qualified to drive'.

Note. Any talk of qualification will invoke modal notions of permission and obligation which can not be fully captured in OWL.

We can get partly around the «no modalities» constraint by introducing object properties with «hidden» modal meaning.

- *driving has_agent* only *qualified_driver* <- here has_agent should have been the modal «ought-to-have_agent» – in a variant of, we interpret statements in the ontology as normative, describing a world where the requirements are fulfilled
  - (where every qualified_driver has_certificate some drivers_licence)