ISO 15926-14:2020(E)

Industrial automation systems and integration — Integration of life-cycle data for process plants including oil and gas production facilities — Part 14: Industrial top-level ontology

WD Proposal

A READI deliverable

Authors
DNV GL: Dirk Walther (Editor), Johan Wilhelm Klüwer and Francisco Martin-Recuerda
University of Oslo/SIRIUS: Arild Waaler and Daniel Lupp
Siemens: Maja Milicic Brandt, Stephan Grimm and Aneta Koleva
Shell: Mesbah Kahn
Aker Solutions: Lillian Hella
POSC Caesar Association: Nils Sandsmark

SEPTEMBER 30, 2020

Working Draft(WD) Proposal for ISO 15926-14:2020(E)

ISO/TC 184/SC 4/WG 3

Secretariat: ANSI

Title (Industrial automation systems and integration — Integration of life-cycle data for process plants including oil and gas production facilities — Part 14: Industrial top-level ontology)

WD stage

Warning for WDs and CDs

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Contents

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 184, “Automation systems and integration”, Subcommittee SC 4, “Industrial data”.

A list of all parts in the ISO 15926 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

The primary purpose of ISO 15926 is to provide a foundation ontology to support the integration and sharing of data related to the lifecycle of a process plant in such a way that it is consistent, unambiguous, and minimizing the number of ways something could be expressed.

The lifecycle dimension of ISO 15926 was captured by a conceptualisation that is intrinsically temporal. This so-called 4D approach underlies the data model of ISO 15926.

Originally, ISO 15926 was developed as a data model that provided a top-level ontology plus a meta model that allowed the ontology to be extended by reference data. However, more recently, with the advent of OWL 2 there has been interest in OWL versions of the top-level ontology. This has resulted in ISO/TS 15926-12.

In all this, reasoning has not been a priority. However, it is not surprising, given the reasoning support for OWL ontologies using Direct Semantics, that there should be interest in developing ontologies to support the process industries that take advantage of this capability.

This standard contains the specification of an OWL 2 Direct Semantics [5] ontology that builds on the ISO 15926-2 Data model [1]. Building such an ontology requires several adaptations of the data model in ISO 15926-2 to the direct semantics framework of OWL 2. In particular, lifecycle modelling needs a different representation in this standard than the 4D approach to lifecycle modelling in ISO 15926-2. The reason for this is that the 4D perspective can only be captured in a direct semantics ontology in an incomplete way. An OWL 2 implementation of a 4D approach can hence not come along with strong reasoning services.

The approach to lifecycle modeling in this standard is based on three modelling patterns that can be used to design a lifecycle information ontology. The three modelling patterns are not temporal in nature, but temporal information can be added through time-related properties, thus enabling a full lifecycle information model with incorporated time. The modelling patterns are illustrated through an interpretation of the lifecycle information model of ISO/IEC 81346-1.

The ontology in this standard is designed so as to provide classes and properties with the possibility of efficient reasoning support. To achieve this, the ontology picks up on best practice modelling in the wider industrial ontology community. Examples of best practice modelling include:

1. The application of ontology modularisation techniques and the organization of ontologies in strict dependent hierarchies, where domain independent ontologies occupy the top levels.
2. Recommendations on best practice modelling from authors of well-known upper ontologies like ISO/IEC 21838-2, the Basic Formal Ontology (BFO)[1] and DOLCE.[2]
3. A careful use of OWL 2 metamodeling based on SKOS specification [10].
4. Restrict OWL 2 interpretation to OWL 2 Direct Semantics [5].
5. Limit the use of certain OWL 2 constructors like property cardinality restrictions [9].

These principles have several consequences for an OWL implementation ontology that is based on ISO/TS 15926-2. One consequence is that the information that ISO/TS 15926-2 has covered at the class of class level needs to be treated at the level of classes by the OWL implementation ontology. In order to be targeted by OWL 2 reasoning, the representation of this information has to be changed.

[1] https://www.iso.org/standard/74572.html

[2] http://wonderweb.man.ac.uk/dissemination.shtml

EXAMPLE A particular mass such as "10 kilograms" is regarded as a class in ISO 15926-2. This information must be represented in an OWL ontology through individuals and object properties in order to enable reasoning support, as illustrated in Appendix E.

This standard reconciles the needs of the 15926 community to converge and align with existent W3C recommendations for semantic technologies. OWL 2 W3C[3] recommendation [6] defines five different syntaxes, including the Manchester syntax [7] which is used in this document.

The use of the ontology in this standard as an upper ontology is illustrated by modelling various real-world use cases from industry.

A group of oil and gas operators and contractors in Norway has more than five years of experience using an ontology that is essentially equivalent to the ontology in this standard. The ontology is successfully serving a multitude of large industry projects. A key learning from the experience gathered in these projects is that assistance from automated reasoning is crucial for managing the complexity of domains and disciplines. Reasoning proved crucial for building reference data that could be easily reused and that could serve a wide range of applications. In particular, automated reasoning enables engineers and developers to discover implicit facts, such as duplicate classes, and hidden inconsistencies in reference data. Reasoning also plays an essential role in end-user tools that use the ontology for the interpretation of requirements and verification of designs. The end-user tools help the engineers to simplify the correct application of standards, enabling a sound continuous extension of the ontology without any need for ontology engineers to be involved.

[3] https://www.w3.org/

Purpose

The purpose of ISO 15926-14 is to meet needs for OWL 2 ontologies that are based on ISO 15926-2, that enable efficient reasoning and that capture lifecycle information.

A specific purpose is to demonstrate lifecycle modelling through a representation of the lifecycle model of ISO/IEC 81346-1.

Another specific purpose is to exemplify how this standard can be used to develop industrial ontologies through various real-world use cases from industy.

Title (Industrial automation systems and integration — Integration of life-cycle data for process plants including oil and gas production facilities — Part 14: Industrial top-level ontology)

1 Scope

This standard contains the specification of ISO 15926-14. This is an ontology compliant with OWL 2 Direct Semantics that builds on the ISO 15926-2 Data model [1]. The ontology is intended to be used in conjunction with reasoning services over OWL 2 ontologies.

ISO 15926-14 is an upper ontology, i.e., it defines general and domain independent terms to facilitate interoperability of ontologies across multiple domains. To facilitate use in the development lifecycle phase, ISO 15926-14 includes terms for defining restrictions identified in the design phase and terms for the transition from design to procurement. This includes in particular class terms for physical objects, systems, functions and functional objects.

Using these terms one can capture the evolution of functional objects from an early design phase to the functional locations of tag numbers and capture the distinction between a tag number and a physical object installed at the tag. This feature is exploited in the modelling patterns for lifecycle information.

The approach to lifecycle information though modelling patterns is exemplified through a representation of the lifecycle model of ISO/IEC 81346-1.

Reasoning services in scope include:

- Checking consistency of an ontology
- Reasoning over formalized class definitions, in particular for detection of duplicate classes
- Checking consistency of classes with respect to restrictions, in particular checking consistency of product classes with respect to classes that capture restrictions from design
- Automated classification of objects based on property values

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 15926-2:2003, *Industrial automation systems and integration — Integration of life-cycle data for process plants including oil and gas production facilities — Part 2: Data model*

ISO/IEC 21838-2, *Information technology — Top-level ontologies (TLO) — Part 2: Basic Formal Ontology (BFO)*

OWL 2 Web Ontology Language, Structural Specification and Functional-Style Syntax (Second Edition). W3C World Wide Web Consortium Recommendation 11 December 2012 (https://www.w3.org/TR/owl2-syntax/)

OWL 2 Web Ontology Language, Direct Semantics (Second Edition). W3C World Wide Web Consortium Recommendation 11 December 2012 (https://www.w3.org/TR/owl2-direct-semantics/)

OWL 2 Web Ontology Language, Manchester Syntax (Second Edition). W3C World Wide Web Consortium Working Group Note 11 December 2012 (https://www.w3.org/TR/owl2-manchester-syntax/)

SKOS Simple Knowledge Organization System Reference. W3C World Wide Web Consortium Recommendation 18 August 2009 (https://www.w3.org/TR/skos-reference/)

3 Terms and definitions

No terms and definitions are listed in this document.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

4 Basic concepts and assumptions

The standard is based on the fundamental concepts and assumptions laid out in ISO 15926-2. In addition, we need to account for certain assumptions regarding modelling lifecycle information and reasoning.

A lifecycle information model addresses the objects of an information model as they are created and evolve through the life cycle until they cease to exist.

A lifecycle information model will typically have to address challenges that are not easily solved by more traditional data models. These may in particular include the following:

- Representation of objects referenced by different identifier systems. For example, a motor can in design documentation be identified by a tag number and, in the maintenance system, be identified by an equipment number.
- Structuring of asset information into a coherent whole. Today, technical information is, as a rule, fragmented and spread over a large number of different sources and formats. A lifecycle ontology can provide a structure into which the different fragments of information can be placed, so the relationship between the fragments is transparent.
- Evolution of resources, including objects being split and merged.

To support such representation three modelling patterns have been identified.

The Specification pattern. This pattern, addressed in Section F.1.1, makes use of metamodeling features of OWL to treat specifications as expressions on a meta-level with respect to the underlying ontology.

The Instantiation pattern. The object property *installedAs* is used to relate a functional object to an installed physical object. Section F.1.2 addresses how properties of the installed object can be verified against the properties of the functional object which capture the specification; see also Section G.6.

The Resource evolution pattern. This pattern makes use of metamodeling features of OWL to describe how resources change over time while avoiding to model time in the ontology; cf. Section F.1.3. Three operations that change resources are distinguished: transformation, merge and split.

Note that the modelling patterns are provided to support the design of lifecycle ontologies. As a case in point, Annex F provides an interpretation of the lifecycle model of ISO/IEC 81346 as an ontology based on this part using the three modelling patterns described above.

5 Relation to reference data

The following subsections are to clarify the relationship of this standard to other parts of ISO 15926.

5.1 Relation to Part 2

This standard is based on the data model of Part 2, but with a different approach to modelling lifecycle information.

The lifecycle dimension of ISO 15926-2 is captured by a 4D conceptualisation that is intrinsically temporal. For instance, the concept of a possible individual is understood as a thing that could exist in space and time and a lifecycle stage is modelled as an essentially temporal relationship between two possible individuals.

The 4D perspective is not completely expressible in Description Logic and hence cannot be supported by complete reasoning procedures in OWL 2. OWL 2 lacks, in particular, built-in support for temporal language constructions. Therefore three modelling patterns have been identified as a replacement for the representation of lifecycle information.

- The Installation modelling pattern in this standard captures the relation between a functional object and an installed physical object. This relation would in the framework of Part 2 be captured as a relation between a possible individual and an actual individual.
- The Specification modelling pattern captures modelling that according to Part 2 would be done through *class of class* constructions.
- The Resource evolution pattern captures modelling that according to Part 2 would be done by relating temporal snapshots of objects in the 4D space. JOHAN: point to earlier versions of something using the dedicated annotation properties under *originatesFrom*: *transformFrom*, *mergeFrom* and *splitFrom*.

5.2 Relation to Part 4

To be completed (also explain relationship to READI reference data)

Core reference data is, as a rule, designed with specific modelling patterns in mind and a wide range of such patterns is described in the ISO 15926-2 documentation, with extensions in subsequent parts; see Annex B for more details.

Reference data may be incorporated into Part 14 compliant ontologies, provided that care is taken to ensure consistency with the restrictions on the upper ontology.

5.3 Relation to Part 7, 8, 11

To be completed.

5.4 Relationship to Part 12

ISO 15926-14 was developed in parallel with ISO/TS 15926-12 [2]. ISO/TS 15926-12 is an *OWL 2 RDF-based Semantics* ontology that deviates as little as possible from the original ISO 15926-2. In particular Part 12 adopts the same 4D conception as Part 2, with only minor changes such as introducing the term non-actual individuals.

Since ISO 15926-14 does not share the 4D conception of ISO/TS 15926-12, it is not a conformant profile of ISO 15926-12.

6 Ontology

The ontology in this standard is formulated in the OWL2 Web Ontology Language [9] and it is intended to be used for reasoning under OWL 2 Direct Semantics [5], which corresponds to the semantics of the Description Logic $\mathcal{SROIQ}(\mathcal{D})$ [11, 12].

The ontology is specifically designed to lay the foundation for efficient reasoning. The possibility of using reasoning is tightly linked to the language constructs in OWL 2 that enable specification of so-called complex classes. One can view complex classes as classes with formalised definitions that one can reason over. To facilitate the definition of complex classes several semantically significant features are built into the ontology as object properties rather than as primitive classes. Where complex classes can easily be defined from object properties, these classes are not explicitly included in the ontology.

The terms of the ontology are documented in Annex A. Every term is presented in a dedicated section containing a definition in OWL 2 Manchester Syntax [7] as well as references to use cases in Annex G, where the use of the term is exemplified.

The definitions in the ontology file referenced in Annex C take precendence over what is documented in Annex A.

Conventions for naming shortcut data properties and inverse object properties are provided in Annex D.

Reasoning services include classification of objects, consistency checking and duplicate detection. Annex E provides a detailed account on reasoning services with the ontology.

7 Use Cases

The use of the ontology in this standard as an upper ontology is illustrated by modelling various real-world use cases from industry. A description of each of the use cases together with a modelling example is provided in Annex G.

The following use cases are included:

1. Events and alarms
2. Physical-spatial
3. Piping and Instrumentation Diagram (P&ID)
4. Software
5. Product catalog
6. Aspect-based reference designation system
7. Requirements
8. Bill of Material (BoM) and Bill of Process (BoP)
9. Physical quantities and units of measurement

Annex A
(normative)

ISO 15926-14 Ontology

This section defines and documents the entities of the ISO 15926-14 upper ontology. The ontology imposes only minimal constraints and category distinctions, to not get in the way of building application ontologies for a broad range of use cases. Figure 1 shows the classes, object properties, and data properties included. Annex B includes a mapping between the ontology in this document and relevant parts of ISO 15926-2. Annex C includes the ontology file. Several term definitions and descriptions provided in this section are to be completed.

The ISO 15926-14 OWL interpretation and adaption of ISO 15926-2 is informed by the research literature in applied ontologies, and by the upper ontologies ISO/IEC 21838-2 (BFO) and DOLCE. For example, the classes *Function* and *InformationObject* have been recast to closely match their same-named counterparts in BFO and IAO. We believe the ontology as presented here is consistent with the original intentions of ISO 15926-2. Unless otherwise stated, the meaning of classes follows their meaning in ISO 15926-2.

Observe that the class hierarchy in Figure 1 contains merely three ultimate classes: *Activity*, *Aspect* and *Object*. Furthermore, the number of subclasses at the level below is limited. This feature is pragmatically built in to make the top-level ontology easier to grasp for new users.

The classes of are defined and documented in Section 7.2, the object properties in Section 7.3, the data properties in Section 7.4 and the annotation properties in Section 7.5.

TODO Note that we are making references to other parts of ISO 15926 in the form of annotation properties that point to classes in the “lis2” and “lis12” namespaces, i.e., entities in the Part 2 (PCA rendering 2008) and Part 12 (ISO 2018) ontologies. These are embedded in the Part 14 OWL ontology, to be conveniently usable in mappings.

Several industrial uses cases are presented in Annex C to illustrate the use of the ISO 15926-14 ontology. For the sake of better readability, namespaces have been removed. OWL 2 axioms are expressed in the OWL 2 Manchester syntax [7]. Although simple and informal, the examples cover a broad range of representation tasks.

A.1 Declarations

A.1.1 Prefixes

```
Prefix: lis: <http://standards.iso.org/iso/15926/part14/>
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: lis2: <http://rds.posccaesar.org/2008/02/OWL/ISO-15926-2_2003#>
Prefix: lis12: <http://standards.iso.org/iso/15926/ontology/life-cycle-integration/>
Prefix: obo: <http://purl.obolibrary.org/obo/>
```

A.1.2 Ontology

```
Ontology: <http://standards.iso.org/iso/15926/part14>
<http://standards.iso.org/iso/15926/part14/1.1>
     Annotations: rdfs:label "ISO 15926-14 upper ontology",
       owl:versionInfo "Revised 2020-09-10",
       pav:previousVersion <http://standards.iso.org/iso/15926/part14/1.0>,
       rdfs:comment "The ISO 15926-14 upper ontology is an OWL 2 DL rendering of the ISO 15926-2
data model."
```

A.2 Classes

The ISO 15926-14 ontology has 33 OWL classes. Figure 1 presents a taxonomy of the classes. The following are the three ultimate classes:

- **Activity** covers processes in which objects participate.
- **Aspect** covers entities that represent attributes and properties of objects.
- **Object** covers the main, independent, entities of interest.

The classes *Activity*, *Aspect* and *Object* are set to be mutually disjoint.

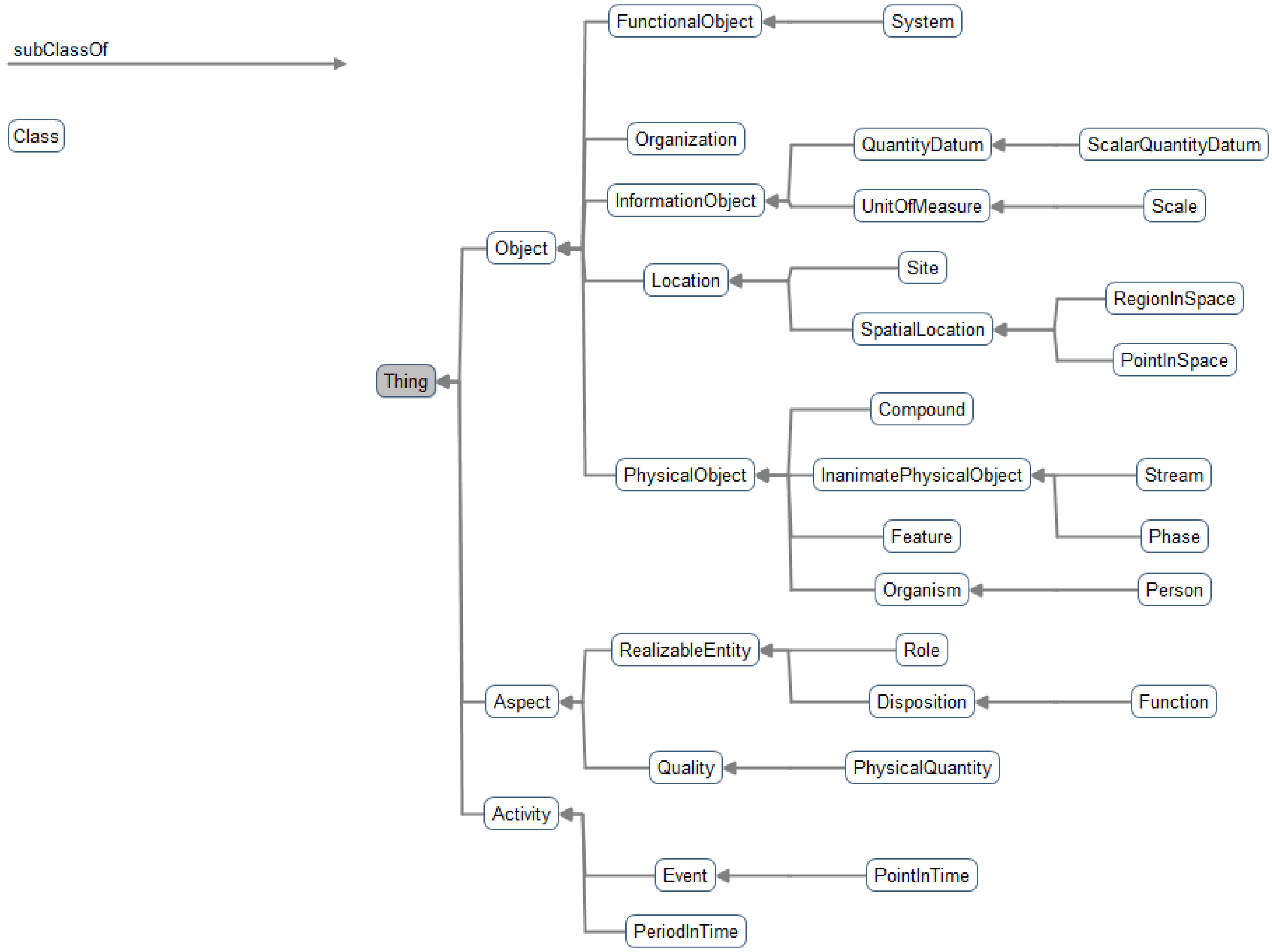


Figure 1 – Classes in ISO 15926-14 ontology

The following subsections provide details on their respective subclasses.

A.2.1 Activity

```
Class: lis:Activity
   Annotations:
       rdfs:label "Activity", skos:prefLabel "Activity"
```

Description to be completed.

The class *Activity* distinguishes two subclasses, *Event* and *PeriodInTime*.

A.2.1.1 Activity > Event

```
Class: lis:Event
   Annotations:
       rdfs:label "Event", skos:prefLabel "Event"
   SubClassOf:
       lis:Activity
```

Description to be completed.

For examples of *Event* in context, see use cases B.1 and B.10.

The class *Event* has one subclass, *PointInTime*.

A.2.1.1.1 Activity > Event > PointInTime

```
Class: lis:PointInTime
   Annotations:
       rdfs:label "PointInTime", skos:prefLabel "PointInTime"
   SubClassOf:
       lis:Event
```

Description to be completed.

A.2.1.2 Activity > PeriodInTime

```
Class: lis:PeriodInTime
   Annotations:
       rdfs:label "PeriodInTime", skos:prefLabel "PeriodInTime"
   SubClassOf:
       lis:Activity
```

Description to be completed.

A.2.2 Aspect

```
Class: lis:Aspect
   Annotations:
       rdfs:label "Aspect", skos:prefLabel "Aspect"
```

Description to be completed.

The class *Aspect* distinguishes two subclasses, *Quality* and *RealizableEntity*.

A.2.2.1 Aspect > Quality

```
Class: lis:Quality
   Annotations:
       rdfs:label "Quality", skos:prefLabel "Quality"
   SubClassOf:
       lis:Aspect
```

The class *Quality* and its subclass *PhysicalQuantity* are directly inspired by corresponding classes included in the DOLCE and BFO upper ontologies. An alternative I have considered is *Property*, which may be re-interpreted vs. Part 2 into ranging over individual properties, but «quality» *seems* to be a term that better matches colloquial use (for referring to individual qualities). Furthermore, in the CD version of Part 12, «Property» has been deprecated for «Quantity», which is not obviously a better choice for the entities in question.

Description to be completed.

The class *Quality* has one subclass, *PhysicalQuantity*.

A.2.2.1.1 Aspect > Quality > PhysicalQuantity

```
Class: lis:PhysicalQuantity
   Annotations:
       rdfs:label "PhysicalQuantity", skos:prefLabel "PhysicalQuantity"
   SubClassOf:
       lis:Quality
```

This class implements the ISO 15926-2 notion of Property, inspired by corresponding classes included in DOLCE and BFO. Mass, Pressure, and Temperature are examples of PhysicalQuantity subclasses likely to be included in any industrial ontology.

For examples of *PhysicalQuantity* in context, see use cases B.5 and B.10.

Description to be completed.

A.2.2.2 Aspect > RealizableEntity

```
Class: lis:RealizableEntity
   Annotations:
       rdfs:label "RealizableEntity",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/BFO_0000017>,
       skos:prefLabel "RealizableEntity"
   SubClassOf:
       lis:Aspect
```

Description to be completed.

The class *RealizableEntity* distinguishes two subclasses, *Disposition* and *Role*.

A.2.2.2.1 Aspect > Realizability > Disposition

```
Class: lis:Disposition
   Annotations:
       rdfs:comment "Inspired by the BFO class \"disposition\" (BFO_0000016).",
       rdfs:label "Disposition",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/BFO_0000016>,
       skos:prefLabel "Disposition"
   SubClassOf:
       lis:RealizableEntity
```

Description to be completed.

The class *Disposition* has one subclass, *Function*.

A.2.2.2.1.1 Aspect > Realizability > Disposition > Function

```
Class: lis:Function
   Annotations:
       rdfs:comment "Inspired by the BFO class \"function\" (BFO_0000034).",
       rdfs:label "Function",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/BFO_0000034>,
       skos:prefLabel "Function"
   SubClassOf:
       lis:Disposition
```

Description to be completed.

For examples of *Function* in context, see use cases B.1, B.3, B.4, B.5, and B.7.

A.2.2.2.2 Aspect > Realizability > Role

```
Class: lis:Role
   Annotations:
       rdfs:comment "This class is motivated in the Part 2 'role' entity type, and in the same-
named BFO class. Part 2 is not very specific about the meaning of roles, but the examples are
clear enough. There is still much disagreement in the ontology field about how roles should be
understood and modelled.",
       rdfs:label "Role", skos:prefLabel "Role"
   SubClassOf:
       lis:RealizableEntity
```

The use of this class should follow the advice of the same-named BFO class.

Description to be completed.

A.2.3 Object

```
Class: lis:Object
   Annotations:
       rdfs:label "Object", skos:prefLabel "Object"
```

The class *Object* distinguishes five subclasses, *FunctionalObject*, *InformationObject*, *Location*, *Organization* and *PhysicalObject*. These subclasses are set to be mutually disjoint.

```
DisjointClasses:
   lis:InformationObject,lis:Location,lis:Organization,lis:PhysicalObject
```

A.2.3.1 Object > FunctionalObject

```
Class: lis:FunctionalObject
   Annotations:
       rdfs:comment "A functional object is part of a system, and has a function whose realisation
contributes to the performance of the system as a whole.",
       rdfs:label "FunctionalObject",
       skos:example "An item on a Process Flow Diagram (PFD) or Process and Instrumentation
Diagram (P&ID) should be classified as a FunctionalObject.",
       skos:note "A class of artefacts such as Pump is not a subclass of FunctionalObject: a pump
that is not in service is not part of a system. However, an individual functional location should
in general be given some high-level artefact classification in addition to its description as part
of a system.",
       skos:prefLabel "FunctionalObject"
   SubClassOf:
       lis:Object,
       lis:functionalPartOf some lis:System,
       lis:hasFunction some lis:Function
```

Every object in an industrial plant is there for a purpose, and the objects are arranged into systems of “functional objects”. Plant design assigns one or more functions to each functional object. The complex function of a system is realized when its functional objects participate in activities according to their designed purpose.

The class *FunctionalObject* has one subclass, *System*.

A.2.3.1.1 Object > FunctionalObject > System

```
Class: lis:System
   Annotations:
       rdfs:comment "A system is a complex of functional parts working together. Each part
contributes to the realisation of the system's function (though not necessarily every part in
every performance of the system).",
       rdfs:label "System",
       skos:note "A functional location that does not itself have functional parts is not a
system.",
       skos:prefLabel "System"
   SubClassOf:
       lis:FunctionalObject,
       lis:hasFunctionalPart some lis:FunctionalObject
```

Description to be completed.

A.2.3.2 Object > InformationObject

```
Class: lis:InformationObject
   Annotations:
       rdfs:label "InformationObject", skos:prefLabel "InformationObject"
   SubClassOf:
       lis:Object
```

Description to be completed.

For examples of *InformationObject* in context, see use cases B.1, B.4, B.5, B.10.

The class *InformationObject* is set to be mutually disjoint with the classes *Location*, *PhysicalObject* and *Organization*.

The class *InformationObject* distinguishes two subclasses, *QuantityDatum* and *UnitOfMeasure*.

A.2.3.2.1 Object > InformationObject > QuantityDatum

```
Class: lis:QuantityDatum
   Annotations:
       rdfs:comment "This class is inspired by the class \"measurement datum\" of the Information
Artefact Ontology. The change of wording from \"measurement\" to \"quantity\" is intended to
support cases where measurement is not involved, such as with nominal values.",
       rdfs:label "QuantityDatum",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/IAO_0000109>,
       skos:prefLabel "QuantityDatum"
   SubClassOf:
       lis:InformationObject
```

The bulk of industrial data consists of measured, estimated, and nominal values of physical quantities that apply to objects. Members of this class represent the fixing of physical quantities to numbers on a scale. The representation pattern for this class is inspired by that of "measurement datum" (IAO_0000109) of the Information Artefact Ontology (IAO).

The class *QuantityDatum* has one subclass, *ScalarQuantityDatum*.

A.2.3.2.1.1 Object > InformationObject > QuantityDatum > ScalarQuantityDatum

```
Class: lis:ScalarQuantityDatum
   Annotations:
       rdfs:comment "A scalar quantity datum has a unique unit of measure and a unique numeric
value. This class is inspired by the class \"scalar measurement datum\" of the Information
Artefact Ontology.",
       rdfs:label "ScalarQuantityDatum",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/IAO_0000032>,
       skos:prefLabel "ScalarQuantityDatum"
   SubClassOf:
       lis:QuantityDatum,
       lis:datumUOM some lis:UnitOfMeasure,
       lis:datumValue some rdfs:Literal
```

A scalar quantity datum specializes QuantityDatum to require a unique unit of measure and a unique numeric value, following “scalar measurement datum” (IAO_0000032) of IAO.

The class *ScalarQuantityDatum* is subsumed by the domains of the object property *datumUOM* and the data property *datumValue*, respectively.

A.2.3.2.2 Object > InformationObject > UnitOfMeasure

```
Class: lis:UnitOfMeasure
   Annotations:
       rdfs:label "UnitOfMeasure", skos:prefLabel "UnitOfMeasure"
   SubClassOf:
       lis:InformationObject
```

Description to be completed.

The class *UnitOfMeasure* has one subclass, *Scale*.

A.2.3.2.2.1 Object > InformationObject > UnitOfMeasure > Scale

```
Class: lis:Scale
   Annotations:
       rdfs:label "Scale", skos:prefLabel "Scale"
   SubClassOf:
       lis:UnitOfMeasure
```

Members of this class are units of measure for quantifiable physical qualities, such as “kilogram”, “pascal”, “bar”, “kelvin”, “Celsius”.

A.2.3.3 Object > Location

```
Class: lis:Location
   Annotations:
       rdfs:label "Location", skos:prefLabel "Location"
   SubClassOf:
       lis:Object
```

Description to be completed.

For examples of *Location* in context, see use cases B.2 and B.7.

The class *Location* is set to be mutually disjoint with the classes *InformationObject*, *PhysicalObject* and *Organization*.

The class Location distinguishes two subclasses, *Site* and *SpatialLocation*.

A.2.3.3.1 Object > Location > Site

```
Class: lis:Site
   Annotations:
       rdfs:comment "From BFO: \"b is a site means: b is a three-dimensional immaterial entity
that is (partially or wholly) bounded by a material entity or it is a three-dimensional immaterial
part thereof. (axiom label in BFO2 Reference: [034-002])\"",
       rdfs:comment "This class is inspired by the class \"site\" of the Information Artefact
Ontology.",
       rdfs:label "Site", skos:prefLabel "Site"
   SubClassOf:
       lis:Location
```

Description to be completed.

For examples of *Site* in context, see use case B.2.

A.2.3.3.2 Object > Location > SpatialLocation

```
Class: lis:SpatialLocation
   Annotations:
       rdfs:label "SpatialLocation", skos:prefLabel "SpatialLocation"
   SubClassOf:
       lis:Location
```

Description to be completed.

The class *SpatialLocation* distinguishes two subclasses, *PointInSpace* and *RegionInSpace*.

A.2.3.3.2.1 Object > Location > SpatialLocation > PointInSpace

```
Class: lis:PointInSpace
   Annotations:
       rdfs:label "PointInSpace", skos:prefLabel "PointInSpace"
   SubClassOf:
       lis:SpatialLocation
```

Description to be completed.

A.2.3.3.2.2 Object > Location > SpatialLocation > RegionInSpace

```
Class: lis:RegionInSpace
   Annotations:
       rdfs:label "RegionInSpace", skos:prefLabel "RegionInSpace"
   SubClassOf:
       lis:SpatialLocation
```

Description to be completed.

A.2.3.4 Object > Organization

```
Class: lis:Organization
   Annotations:
       rdfs:label "Organization", skos:prefLabel "Organization"
   SubClassOf:
       lis:Object
```

Description to be completed.

For examples of *Organization* in context, see use case B.8.

The class *Organization* is set to be mutually disjoint with the classes *InformationObject*, *Location and PhysicalObject*.

A.2.3.5 Object > PhysicalObject

```
Class: lis:PhysicalObject
   Annotations:
       rdfs:label "PhysicalObject", skos:prefLabel "PhysicalObject"
   SubClassOf:
       lis:Object
```

Physical objects are the main citizens in an industrial ontology. Objects in this category are characterized with material and geometrical properties. They also have intended functions, which are individual entities dependent on their bearers.

The class *PhysicalObject* is set to be mutually disjoint with the classes *InformationObject*, *Location* and *Organization*.

The class *PhysicalObject* distinguish four subclasses, *Compound*, *Feature*, *InanimatePhysicalObject* and *Organism*.

A.2.3.5.1 Object > PhysicalObject > Compound

```
Class: lis:Compound
   Annotations:
       rdfs:label "Compound", skos:prefLabel "Compound"
   SubClassOf:
       lis:PhysicalObject
```

Description to be completed.

A.2.3.5.2 Object > PhysicalObject > Feature

```
Class: lis:Feature
   Annotations:
       rdfs:label "Feature", skos:prefLabel "Feature"
   SubClassOf:
       lis:PhysicalObject
```

Description to be completed.

A.2.3.5.3 Object > PhysicalObject > InanimatePhysicalObject

```
Class: lis:InanimatePhysicalObject
   Annotations:
       rdfs:label "InanimatePhysicalObject", skos:prefLabel "InanimatePhysicalObject"
   SubClassOf:
       lis:PhysicalObject
```

Description to be completed.

The class *InanimatePhysicalObject* distinguish two subclasses, *Phase* and *Stream*.

A.2.3.5.3.1 Object > PhysicalObject > InanimatePhysicalObject > Phase

```
Class: lis:Phase
   Annotations:
       rdfs:label "Phase", skos:prefLabel "Phase"
   SubClassOf:
       lis:InanimatePhysicalObject
```

Description to be completed.

A.2.3.5.3.2 Object > PhysicalObject > InanimatePhysicalObject > Stream

```
Class: lis:Stream
   Annotations:
       rdfs:label "Stream", skos:prefLabel "Stream"
   SubClassOf:
       lis:InanimatePhysicalObject
```

Description to be completed.

A.2.3.5.4 Object > PhysicalObject > Organism

```
Class: lis:Organism
   Annotations:
       rdfs:label "Organism", skos:prefLabel "Organism"
   SubClassOf:
       lis:PhysicalObject
```

Description to be completed.

The class *Organism* has one subclass, *Person*.

A.2.3.5.4.1 Object > PhysicalObject > Organism > Person

```
Class: lis:Person
   Annotations:
       rdfs:label "Person", skos:prefLabel "Person"
   SubClassOf:
       lis:Organism
```

Description to be completed.

A.3 Object properties

The ISO 15926-14 ontology has 63 object properties. Figure 2 presents an overview over all object properties.

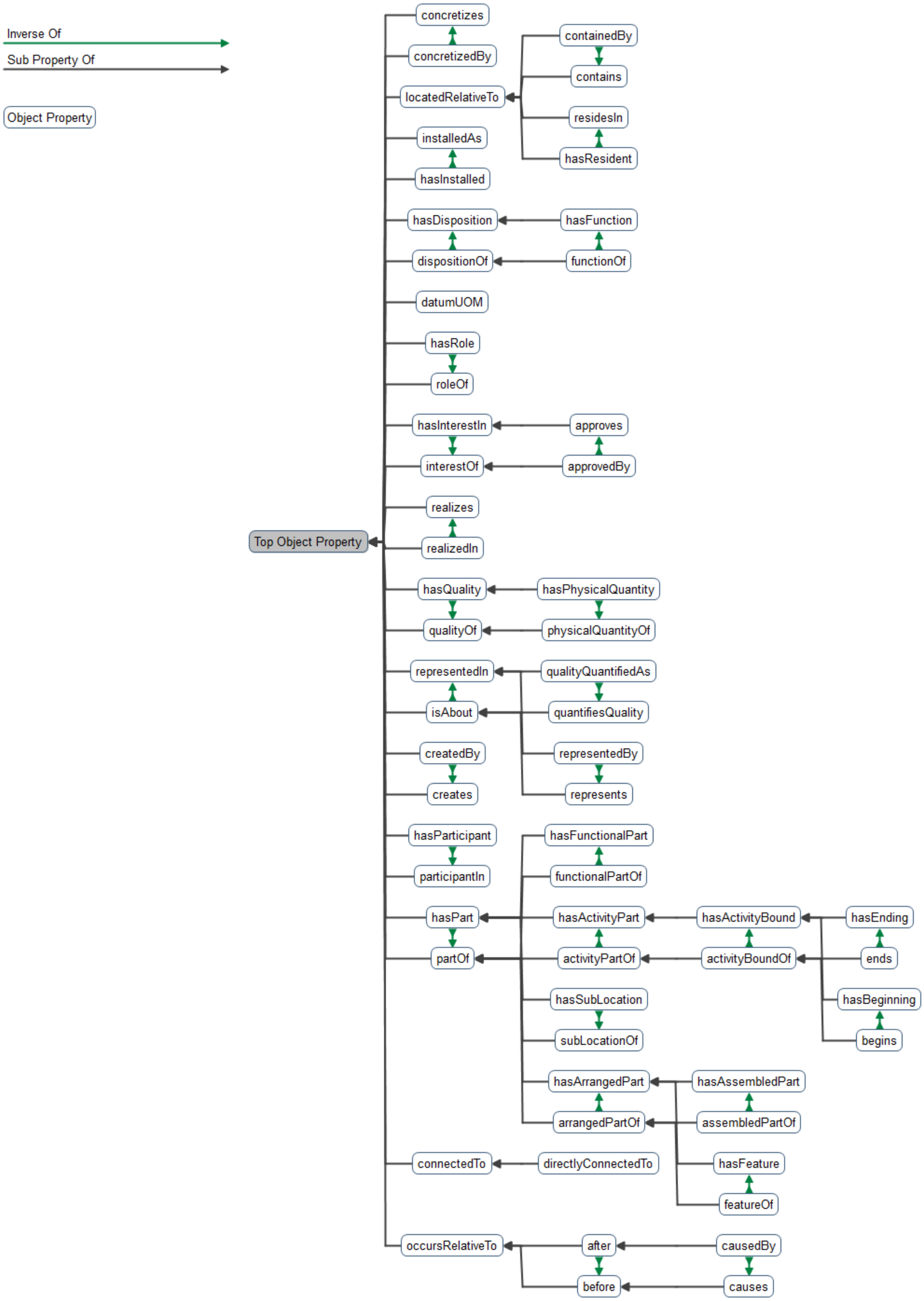


Figure 2 - Object properties of ISO 15926-14 ontology

The object properties are listed following their respective super-properties and their inverses.

A.3.1 concretizedBy

```
ObjectProperty: lis:concretizedBy
   Annotations:
       rdfs:label "concretizedBy", skos:prefLabel "concretizedBy"
   InverseOf:
       lis:concretizes
```

Description to be completed.

The object property *concretizedBy* is set to be the inverse of the object property *concretizes*.

A.3.2 concretizes

```
ObjectProperty: lis:concretizes
   Annotations:
       rdfs:comment "Inspired by BFO's \"concretizes\". Note that the ISO 15926-14 definition
diverges slightly from that in BFO, mainly in employing Feature where BFO has \"specifically
dependent continuant\".",
       rdfs:comment "TODO. Consider splitting this into two relations, one concretizesInFeature
and another concretizesInActivity, to avoid a disjunctive domain.",
       rdfs:label "concretizes",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/BFO_0000164>,
       rdfs:seeAlso <http://purl.obolibrary.org/obo/RO_0000059>,
       skos:prefLabel "concretizes"
   Domain:
       lis:Activity or lis:Feature
   Range:
       lis:InformationObject
   InverseOf:
       lis:concretizedBy
```

Description to be completed.

The object property *concretizedBy* is set to be the inverse of the object property *concretizes*.

A.3.3 connectedTo

```
ObjectProperty: lis:connectedTo
   Annotations:
       rdfs:label "connectedTo", skos:prefLabel "connectedTo"
   Characteristics:
       Symmetric
   Domain:
       lis:PhysicalObject
```

Description to be completed.

The object property *connectedTo* has one sub-property, *directlyConnectedTo*.

A.3.3.1 connectedTo > directlyConnectedTo

```
ObjectProperty: lis:directlyConnectedTo
   Annotations:
       rdfs:label "directlyConnectedTo", skos:prefLabel "directlyConnectedTo"
   Characteristics:
       Symmetric
   SubPropertyOf:
       lis:connectedTo
```

Description to be completed.

A.3.4 createdBy

```
ObjectProperty: lis:createdBy
   Annotations:
       rdfs:label "createdBy", skos:prefLabel "createdBy"
   InverseOf:
       lis:creates
```

Description to be completed.

The object property *createdBy* is set to be the inverse of the object property *creates*.

A.3.5 creates

```
ObjectProperty: lis:creates
   Annotations:
       rdfs:comment "Use this relation to express that an activity brings a physical object into
being. (Derived from class_of_cause_of_beginning_of_class_of_individual in Part 2).",
       rdfs:label "creates", skos:prefLabel "creates"
   Domain:
       lis:Activity
   Range:
       lis:PhysicalObject
   InverseOf:
       lis:createdBy
```

Description to be completed.

The object property *creates* is set to be the inverse of the object property *createdBy*.

A.3.6 datumUOM

```
ObjectProperty: lis:datumUOM
   Annotations:
       rdfs:comment "Relation (functional) to assign unit of measure to measurement data.",
       rdfs:label "datumUOM", skos:prefLabel "datumUOM"
   Characteristics:
       Functional
   Domain:
       lis:QuantityDatum
   Range:
       lis:UnitOfMeasure
```

Used to assign a unit of measure to a quantity datum.

Description to be completed.

A.3.7 dispositionOf

```
ObjectProperty: lis:dispositionOf
   Annotations:
       rdfs:label "dispositionOf", skos:prefLabel "dispositionOf"
   InverseOf:
       lis:hasDisposition
```

Description to be completed.

The object property *dispositionOf* is set to be the inverse of the object property *hasDisposition*.

The object property *dispositionOf* has one sub-property, *functionOf*.

A.3.7.1 dispositionOf > functionOf

```
ObjectProperty: lis:functionOf
   Annotations:
       rdfs:label "functionOf", skos:prefLabel "functionOf"
   SubPropertyOf:
       lis:dispositionOf
   InverseOf:
       lis:hasFunction
```

Description to be completed.

The object property *functionOf* is set to be the inverse of the object property *hasFunction*.

A.3.8 hasDisposition

```
ObjectProperty: lis:hasDisposition
   Annotations:
       rdfs:comment "Inspired by BFO's \"has disposition\" (RO_0000091).",
       rdfs:label "hasDisposition",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/RO_0000091>,
       skos:prefLabel "hasDisposition"
   Range:
       lis:Disposition
   InverseOf:
       lis:dispositionOf
```

Description to be completed.

The object property *hasDisposition* is set to be the inverse of the object property *dispositionOf*.

The object property *hasDisposition* has one sub-property, *hasFunction*.

A.3.8.1 hasDisposition > hasFunction

```
ObjectProperty: lis:hasFunction
   Annotations:
       rdfs:comment "Inspired by BFO's \"has function\" (RO_0000085).",
       rdfs:label "hasFunction",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/RO_0000085>,
       skos:prefLabel "hasFunction"
   SubPropertyOf:
       lis:hasDisposition
   Range:
       lis:Function
   InverseOf:
       lis:functionOf
```

The object property *hasFunction* is used to assign individual functions to physical objects. In class constraints, this allows for physical objects to be grouped according to purpose.
Description to be completed.

The object property *hasFunction* is set to be the inverse of the object property *functionOf*.

A.3.9 installedAs

```
ObjectProperty: lis:installedAs
   Annotations:
       rdfs:label "installedAs", skos:prefLabel "installedAs"
   Domain:
       lis:PhysicalObject
   Range:
       lis:FunctionalObject
   InverseOf:
       lis:hasInstalled
```

Description to be completed.

The object property *installedAs* is set to be the inverse of the object property *hasInstalled*.

A.3.10 hasInstalled

```
ObjectProperty: lis:hasInstalled
   Annotations:
       rdfs:label "hasInstalled", skos:prefLabel "hasInstalled"
   InverseOf:
       lis:installedAs
```

Description to be completed.

The object property *hasInstalled* is set to be the inverse of the object property *installedAs*.

A.3.11 interestOf

```
ObjectProperty: lis:interestOf
   Annotations:
       rdfs:comment "Derived from \"LifecycleStage\" of Part 2, this is a superproperty suitable
for various intentional relationships, such as planning, approving, or ordering. The Part 2 name
\"lifecycle stage\" is likely to confuse, but the intended use of this type is clear enough from
this Part 2 annotation to ClassOfLifecycleStage: \"EXAMPLE Planned, required, expected, and
proposed can be represented by instances of [class_of_lifecycle_stage].\"",
       rdfs:label "interestOf", skos:prefLabel "interestOf"
   InverseOf:
       lis:hasInterestIn
```

Description to be completed.

The object property *interestOf* is set to be the inverse of the object property *hasInterestIn*.

The object property *interestOf* has one sub-property, *approvedBy*.

A.3.11.1 interestOf > approvedBy

```
ObjectProperty: lis:approvedBy
   Annotations:
       rdfs:comment "Relation for stating that some item or activity was approved by an entity,
typically a person or an organisation.",
       rdfs:label "approvedBy", skos:prefLabel "approvedBy"
   SubPropertyOf:
       lis:interestOf
   InverseOf:
       lis:approves
```

Description to be completed.

The object property *approvedBy* is set to be the inverse of the object property *approves*.

A.3.12 hasInterestIn

```
ObjectProperty: lis:hasInterestIn
   Annotations:
       rdfs:label "hasInterestIn", skos:prefLabel "hasInterestIn"
   InverseOf:
       lis:interestOf
```

Description to be completed.

The object property *hasInterestIn* is set to be the inverse of the object property *interestOf*.

The object property *hasInterestIn* has one sub-property, *approves*.

A.3.12.1 hasInterestIn > approves

```
ObjectProperty: lis:approves
   Annotations:
       rdfs:label "approves", skos:prefLabel "approves"
   SubPropertyOf:
       lis:hasInterestIn
   InverseOf:
       lis:approvedBy
```

Description to be completed.

The object property *approves* is set to be the inverse of the object property *approvedBy*.

A.3.13 partOf

```
ObjectProperty: lis:partOf
   Annotations:
       rdfs:label "partOf", skos:prefLabel "partOf"
   InverseOf:
       lis:hasPart
```

Description to be completed.

The object property *partOf* is set to be the inverse of the object property *hasPart*.

The object property *partOf* has four sub-properties, *activityPartOf*, *arrangedPartOf*, *functionalPartOf* and *subLocationOf*.

A.3.13.1 partOf > activityPartOf

```
ObjectProperty: lis:activityPartOf
   Annotations:
       rdfs:label "activityPartOf", skos:prefLabel "activityPartOf"
   SubPropertyOf:
       lis:partOf
   InverseOf:
       lis:hasActivityPart
```

Description to be completed.

The object property *activityPartOf* is set to be the inverse of the object property *hasActivityPart*.

The object property *activityPartOf* has one sub-property, *activityBoundOf*.

A.3.13.1.1 partOf > activityPartOf > activityBoundOf

```
ObjectProperty: lis:activityBoundOf
   Annotations:
       rdfs:label "activityBoundOf", skos:prefLabel "activityBoundOf"
   SubPropertyOf:
       lis:activityPartOf
   InverseOf:
       lis:hasActivityBound
```

Description to be completed.

The object property *activityBoundOf* is set to be the inverse of the object property *hasActivityBound*.

The object property *activityBoundOf* has two sub-properties, *begins* and *ends*.

A.3.13.1.1.1 partOf > activityPartOf > activityBoundOf > begins

```
ObjectProperty: lis:begins
   Annotations:
       rdfs:label "begins", skos:prefLabel "begins"
   SubPropertyOf:
       lis:activityBoundOf
   InverseOf:
       lis:hasBeginning
```

Description to be completed.

The object property *begins* is set to be the inverse of the object property *hasBeginning*.

A.3.13.1.1.2 partOf > activityPartOf > activityBoundOf > ends

```
ObjectProperty: lis:ends
   Annotations:
       rdfs:label "ends", skos:prefLabel "ends"
   SubPropertyOf:
       lis:activityBoundOf
   InverseOf:
       lis:hasEnding
```

Description to be completed.

The object property *ends* is set to be the inverse of the object property *hasEnding*.

A.3.13.2 partOf > arrangedPartOf

```
ObjectProperty: lis:arrangedPartOf
   Annotations:
       rdfs:label "arrangedPartOf", skos:prefLabel "arrangedPartOf"
   SubPropertyOf:
       lis:partOf
   InverseOf:
       lis:hasArrangedPart
```

Description to be completed.

The object property *arrangedPartOf* is set to be the inverse of the object property *hasArrangedPart*.

The object property *arrangedPartOf* has two sub-properties, *assembledPartOf* and *featureOf*.

A.3.13.2.1 partOf > arrangedPartOf > assembledPartOf

```
ObjectProperty: lis:assembledPartOf
   Annotations:
       rdfs:label "assembledPartOf", skos:prefLabel "assembledPartOf"
   SubPropertyOf:
       lis:arrangedPartOf
   InverseOf:
       lis:hasAssembledPart
```

Description to be completed.

The object property *assembledPartOf* is set to be the inverse of the object property *hasAssembledPart*.

A.3.13.2.2 partOf > arrangedPartOf > featureOf

```
ObjectProperty: lis:featureOf
   Annotations:
       rdfs:label "featureOf", skos:prefLabel "featureOf"
   SubPropertyOf:
       lis:arrangedPartOf
   InverseOf:
       lis:hasFeature
```

Description to be completed.

The object property *featureOf* is set to be the inverse of the object property *hasFeature*.

A.3.13.3 partOf > functionalPartOf

```
ObjectProperty: lis:functionalPartOf
   Annotations:
       rdfs:comment "Where x is a functional part of a system y, the realisation of one or more
functions of x contributes to the performance of y.",
       rdfs:label "functionalPartOf",
       rdfs:seeAlso "lis:FunctionalObject",
       skos:prefLabel "functionalPartOf"
   SubPropertyOf:
       lis:partOf
   Domain:
       lis:FunctionalObject
   Range:
       lis:System
   InverseOf:
       lis:hasFunctionalPart
```

Description to be completed.

The object property *functionalPartOf* is set to be the inverse of the object property *hasFunctionalPart*.

A.3.13.4 partOf > subLocationOf

```
ObjectProperty: lis:subLocationOf
   Annotations:
       rdfs:label "subLocationOf", skos:prefLabel "subLocationOf"
   SubPropertyOf:
       lis:partOf
   InverseOf:
       lis:hasSubLocation
```

Description to be completed.

The object property *subLocationOf* is set to be the inverse of the object property *hasSubLocation*.

A.3.14 hasPart

```
ObjectProperty: lis:hasPart
   Annotations:
       rdfs:label "hasPart", skos:prefLabel "hasPart"
   InverseOf:
       lis:partOf
```

This property, and its inverse hasPart, expresses part-whole relationships, as in a physical break-down structure of a plant.

Description to be completed.

The object property *hasPart* is set to be the inverse of the object property p*artOf*.

The object property *hasPart* has four sub-properties, *hasActivityPart*, *hasArrangedPart*, *hasFunctionalPart,* and *hasSubLocation*.

A.3.14.1 hasPart > hasActivityPart

```
ObjectProperty: lis:hasActivityPart
   Annotations:
       rdfs:label "hasActivityPart", skos:prefLabel "hasActivityPart"
   SubPropertyOf:
       lis:hasPart
   Domain:
       lis:Activity
   Range:
       lis:Activity
   InverseOf:
       lis:activityPartOf
```

Description to be completed.

The object property *hasActivityPart* is set to be the inverse of the object property *activityPartOf*.

The object property *hasActivityPart* has one sub-property, *hasActivityBound*.

A.3.14.1.1 hasPart > hasActivityPart > hasActivityBound

```
ObjectProperty: lis:hasActivityBound
   Annotations:
       rdfs:label "hasActivityBound", skos:prefLabel "hasActivityBound"
   SubPropertyOf:
       lis:hasActivityPart
   InverseOf:
       lis:activityBoundOf
```

Description to be completed.

The object property *hasActivityBound* is set to be the inverse of the object property *activityBoundOf*.

The object property *hasActivityBound* has two sub-properties*,* *hasBeginning* and *hasEnding*.

A.3.14.1.1.1 hasPart > hasActivityPart > hasActivityBound > hasBeginning

```
ObjectProperty: lis:hasBeginning
   Annotations:
       rdfs:label "hasBeginning", skos:prefLabel "hasBeginning"
   SubPropertyOf:
       lis:hasActivityBound
   InverseOf:
       lis:begins
```

Description to be completed.

The object property *hasBeginning* is set to be the inverse of the object property *begins*.

A.3.14.1.1.2 hasPart > hasActivityPart > hasActivityBound > hasEnding

```
ObjectProperty: lis:hasEnding
   Annotations:
       rdfs:label "hasEnding", skos:prefLabel "hasEnding"
   SubPropertyOf:
       lis:hasActivityBound
   InverseOf:
       lis:ends
```

Description to be completed.

The object property *hasEnding* is set to be the inverse of the object property *ends.*

A.3.14.2 hasPart > hasArrangedPart

```
ObjectProperty: lis:hasArrangedPart
   Annotations:
       rdfs:comment "In line with intended use, for the DL profile this relation has a domain
restricted to physical objects.",
       rdfs:label "hasArrangedPart", skos:prefLabel "hasArrangedPart"
   SubPropertyOf:
       lis:hasPart
   Domain:
       lis:PhysicalObject
   InverseOf:
       lis:arrangedPartOf
```

Description to be completed.

The object property *hasArrangedPart* is set to be the inverse of the object property *arrangedPartOf*.

The object property *hasArrangedPart* has two sub-properties, *hasAssembledPart* and *hasFeature*.

A.3.14.2.1 hasPart > hasArrangedPart > hasAssembledPart

```
ObjectProperty: lis:hasAssembledPart
   Annotations:
       rdfs:comment "This is the recommended (super-) relation for capturing physical breakdown of
mechanical assemblies.",
       rdfs:label "hasAssembledPart", skos:prefLabel "hasAssembledPart"
   SubPropertyOf:
       lis:hasArrangedPart
   InverseOf:
       lis:assembledPartOf
```

Description to be completed.

The object property *hasAssembledPart* is set to be the inverse of the object property *assembledPartOf*.

A.3.14.2.2 hasPart > hasArrangedPart > hasFeature

```
ObjectProperty: lis:hasFeature
   Annotations:
       rdfs:comment "Example of usage: stating that an entity has a surface suitable for
connection, such as a flange face.",
       rdfs:label "hasFeature", skos:prefLabel "hasFeature"
   SubPropertyOf:
       lis:hasArrangedPart
   InverseOf:
       lis:featureOf
```

Description to be completed.

The object property *hasFeature* is set to be the inverse of the object property *featureOf*.

A.3.14.3 hasPart > hasFunctionalPart

```
ObjectProperty: lis:hasFunctionalPart
   Annotations:
       rdfs:label "hasFunctionalPart", skos:prefLabel "hasFunctionalPart"
   SubPropertyOf:
       lis:hasPart
   InverseOf:
       lis:functionalPartOf
```

Description to be completed.

The object property *hasFunctionalPart* is set to be the inverse of the object property *functionalPartOf*.

A.3.14.4 hasPart > hasSubLocation

```
ObjectProperty: lis:hasSubLocation
   Annotations:
       rdfs:label "hasSubLocation", skos:prefLabel "hasSubLocation"
   SubPropertyOf:
       lis:hasPart
   Domain:
       lis:Location
   Range:
       lis:Location
   InverseOf:
       lis:subLocationOf
```

Description to be completed.

The object property *hasSubLocation* is set to be the inverse of the object property *subLocationOf*.

A.3.15 participantIn

```
ObjectProperty: lis:participantIn
   Annotations:
       rdfs:label "participantIn", skos:prefLabel "participantIn"
   InverseOf:
       lis:hasParticipant
```

Used to express participation of an object in an Activity. Use cases should introduce subproperties to capture the type of participation – as resource, agent, output, etc.

Description to be completed.

The object property *participantIn* is set to be the inverse of the object property *hasParticipant*.

A.3.16 hasParticipant

```
ObjectProperty: lis:hasParticipant
   Annotations:
        rdfs:comment "This is the recommended superrelation for types of participation by entities
in activities -- the agent, the matter being acted upon, etc.",
       rdfs:label "hasParticipant", skos:prefLabel "hasParticipant"
   Domain:
       lis:Activity
   InverseOf:
       lis:participantIn
```

Description to be completed.

The object property *hasParticipant* is set to be the inverse of the object property *participantIn*.

A.3.17 qualityOf

```
ObjectProperty: lis:qualityOf
   Annotations:
       rdfs:label "qualityOf", skos:prefLabel "qualityOf"
   InverseOf:
       lis:hasQuality
```

Description to be completed.

The object property *qualityOf* is set to be the inverse of the object property *hasQuality*.

The object property *qualityOf* has one sub-property, *physicalQuantityOf*.

A.3.17.1 qualityOf > physicalQuantityOf

```
ObjectProperty: lis:physicalQuantityOf
   Annotations:
       rdfs:label "physicalQuantityOf", skos:prefLabel "physicalQuantityOf"
   SubPropertyOf:
       lis:qualityOf
   InverseOf:
       lis:hasPhysicalQuantity
```

Description to be completed.

The object property *physicalQuantityOf* is set to be the inverse of the object property *hasPhysicalQuantity*.

A.3.18 hasQuality

```
ObjectProperty: lis:hasQuality
   Annotations:
       rdfs:label "hasQuality", skos:prefLabel "hasQuality"
   Range:
       lis:Quality
   InverseOf:
       lis:qualityOf
```

Used to assign individual qualities to physical objects. In class constraints, this allows for categorization of physical objects according to material and geometrical aspects.
Description to be completed.

The object property *hasQuality* is set to be the inverse of the object property *qualityOf*.

The object property *hasQuality* has one sub-property, *hasPhysicalQuantity*.

A.3.18.1 hasQuality > hasPhysicalQuantity

```
ObjectProperty: lis:hasPhysicalQuantity
   Annotations:
       rdfs:label "hasPhysicalQuantity", skos:prefLabel "hasPhysicalQuantity"
   SubPropertyOf:
       lis:hasQuality
   Domain:
       lis:PhysicalObject
   Range:
       lis:PhysicalQuantity
   InverseOf:
       lis:physicalQuantityOf
```

Description to be completed.

The object property *hasPhysicalQuantity* is set to be the inverse of the object property *physicalQuantityOf*.

A.3.19 roleOf

```
ObjectProperty: lis:roleOf
   Annotations:
       rdfs:comment "Inspired by BFO's \"role of\" (RO_0000081)",
       rdfs:label "roleOf",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/RO_0000081>,
       skos:prefLabel "roleOf"
   InverseOf:
       lis:hasRole
```

Description to be completed.

The object property *roleOf* is set to be the inverse of the object property *hasRole*.

A.3.20 hasRole

```
ObjectProperty: lis:hasRole
   Annotations:
       rdfs:comment "Inspired by BFO's \"has role\" (RO_0000087)",
       rdfs:label "hasRole",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/RO_0000087>,
       skos:prefLabel "hasRole"
   Range:
       lis:Role
   InverseOf:
       lis:roleOf
```

Description to be completed.

The object property *hasRole* is set to be the inverse of the object property *roleOf*.

A.3.21 installedAs

```
ObjectProperty: lis:installedAs
   Annotations:
       rdfs:label "installedAs", skos:prefLabel "installedAs"
   Domain:
       lis:PhysicalObject
   Range:
       lis:FunctionalObject
   InverseOf:
       lis:hasInstalled
```

Description to be completed.

The object property *installedAs* is set to be the inverse of the object property *hasInstalled*.

A.3.22 hasInstalled

```
ObjectProperty: lis:hasInstalled
   Annotations:
       rdfs:label "hasInstalled", skos:prefLabel "hasInstalled"
   InverseOf:
       lis:installedAs
```

Description to be completed.

The object property *hasInstalled* is set to be the inverse of the object property *installedAs*.

A.3.23 isAbout

```
ObjectProperty: lis:isAbout
   Annotations:
       rdfs:label "isAbout",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/IAO_0000136>,
       skos:prefLabel "isAbout"
   InverseOf:
       lis:representedIn
```

Description to be completed.

The object property *isAbout* is set to be the inverse of the object property *representedIn*.

The object property *isAbout* has two sub-properties, *quantifiesQuality* and *represents*.

A.3.23.1 isAbout > quantifiesQuality

```
ObjectProperty: lis:quantifiesQuality
   Annotations:
       rdfs:label "quantifiesQuality", skos:prefLabel "quantifiesQuality"
   SubPropertyOf:
       lis:isAbout
   InverseOf:
       lis:qualityQuantifiedAs
```

Description to be completed.

The object property *quantifiesQuality* is set to be the inverse of the object property *qualityQuantifiedAs*.

A.3.23.2 isAbout > represents

```
ObjectProperty: lis:represents
   Annotations:
       rdfs:label "represents", skos:prefLabel "represents"
   SubPropertyOf:
       lis:isAbout
   InverseOf:
       lis:representedBy
```

Description to be completed.

The object property *represents* is set to be the inverse of the object property *representedBy*.

A.3.24 representedIn

```
ObjectProperty: lis:representedIn
   Annotations:
       rdfs:comment "Also see \"is about\" IAO_0000136 of the Information Artifact Ontology, which
may be better named for a maximally general relation of \"aboutness\" (but note that \"is about\"
goes in the opposite direction of \"representedIn\").",
       rdfs:label "representedIn",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/IAO_0000136>,
       skos:prefLabel "representedIn"
   Range:
       lis:InformationObject
   InverseOf:
       lis:isAbout
```

Description to be completed.

The object property *representedIn* is set to be the inverse of the object property *isAbout*.

The object property *representedIn* has two sub-properties, *qualityQuantifiedAs* and *representedBy*.

A.3.24.1 representedIn > qualityQuantifiedAs

```
ObjectProperty: lis:qualityQuantifiedAs
   Annotations:
       rdfs:comment "This relation is inspired by the relation \"is quality measured as\" of the
Information Artefact Ontology. The term \"quantified\" replaces \"measured\" to support cases
where measurement is not involved, as in e.g. estimates.",
       rdfs:label "qualityQuantifiedAs", skos:prefLabel "qualityQuantifiedAs"
   SubPropertyOf:
       lis:representedIn
   Domain:
       lis:Quality
   Range:
       lis:QuantityDatum
   InverseOf:
       lis:quantifiesQuality
```

Used to relate physical aspects of individuals to the information objects that measure (or estimate, etc.), them.

Description to be completed.

A.3.24.2 representedIn > representedBy

```
ObjectProperty: lis:representedBy
   Annotations: 
       rdfs:comment "In Part 2, this is the top level relation from things to information objects. 
For this Part, we introduce representedIn, as the more generic inverse of isAbout.",
       rdfs:label "representedBy", skos:prefLabel "representedBy"
   SubPropertyOf: 
       lis:representedIn
   Range: 
       lis:InformationObject
   InverseOf: 
       lis:represents
```

Description to be completed.

The object property *representedBy* is set to be the inverse of the object property *represents*.

A.3.25 locatedRelativeTo

```
ObjectProperty: lis:locatedRelativeTo
   Annotations: 
       rdfs:label "locatedRelativeTo", skos:prefLabel "locatedRelativeTo"
   Characteristics: 
       Symmetric
```

Description to be completed.

The object property *locatedRelativeTo* has four sub-properties, *containedBy*, *contains*, *hasResident* and *residesIn*.

A.3.25.1 locatedRelativeTo > containedBy

```
ObjectProperty: lis:containedBy
   Annotations: 
       rdfs:label "containedBy", skos:prefLabel "containedBy"
   SubPropertyOf: 
       lis:locatedRelativeTo
   InverseOf: 
       lis:contains
```

Description to be completed.

The object property *containedBy* is set to be the inverse of the object property *contains*.

A.3.25.2 locatedRelativeTo > contains

```
ObjectProperty: lis:contains
   Annotations:
       rdfs:comment "For the DL profile, we restrict this relation to physical objects. Note that
this rules out using \"lis:contains\" for spatial locations.",
       rdfs:label "contains", skos:prefLabel "contains"
   SubPropertyOf:
       lis:locatedRelativeTo
   Domain:
       lis:PhysicalObject
   Range:
       lis:PhysicalObject
   InverseOf:
       lis:containedBy
```

Description to be completed.

The object property *contains* is set to be the inverse of the object property *containedBy*.

A.3.25.3 locatedRelativeTo > residesIn

```
ObjectProperty: lis:residesIn
   Annotations:
       rdfs:comment "x residesIn y if x is a physical object that is located in the location y.",
       rdfs:label "residesIn", skos:prefLabel "residesIn"
   SubPropertyOf:
       lis:locatedRelativeTo
   Domain:
       lis:PhysicalObject
   Range:
       lis:Location
   InverseOf:
       lis:hasResident
```

Description to be completed.

The object property *residesIn* is set to be the inverse of the object property *hasResident*.

A.3.25.4 locatedRelativeTo > hasResident

```
ObjectProperty: lis:hasResident
   Annotations:
       rdfs:label "hasResident", skos:prefLabel "hasResident"
   SubPropertyOf:
       lis:locatedRelativeTo
   InverseOf:
       lis:residesIn
```

Description to be completed.

The object property *hasResident* is set to be the inverse of the object property *residesIn*.

A.3.26 occursRelativeTo

```
ObjectProperty: lis:occursRelativeTo
   Annotations:
       rdfs:comment "This relation is introduced for the DL profile as a top relation for various
temporal relations between activities.",
       rdfs:label "occursRelativeTo", skos:prefLabel "occursRelativeTo"
   Characteristics:
       Symmetric
   Domain:
       lis:Activity
   Range:
       lis:Activity
```

Description to be completed.

The object property *occursRelativeTo* has two sub-properties, *before* and *after*.

A.3.26.1 occursRelativeTo > after

```
ObjectProperty: lis:after
   Annotations:
       rdfs:comment "Use this relation to state that one activity after before another.",
       rdfs:label "after", skos:prefLabel "after"
   SubPropertyOf:
       lis:occursRelativeTo
   InverseOf:
       lis:before
```

Description to be completed.

The object property *after* is set to be the inverse of the object property *before*.

A.3.26.2 occursRelativeTo > before

```
ObjectProperty: lis:before
   Annotations:
       rdfs:comment "Use this relation to state that one activity occurs before another.",
       rdfs:label "before", skos:prefLabel "before"
   SubPropertyOf:
       lis:occursRelativeTo
   InverseOf:
       lis:after
```

Description to be completed.

The object property *before* is set to be the inverse of the object property *after*.

The object property *before* has one sub-property, *causes*.

A.3.26.2.1 occursRelativeTo > before > causes

```
ObjectProperty: lis:causes
   Annotations:
       rdfs:label "causes", skos:prefLabel "causes"
   SubPropertyOf:
       lis:before
```

Description to be completed.

A.3.27 realizedIn

```
ObjectProperty: lis:realizedIn
   Annotations:
       rdfs:comment "Inspired by BFO's \"realized in\" (BFO_0000054)",
       rdfs:label "realizedIn",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/BFO_0000054>,
       skos:prefLabel "realizedIn"
   Domain:
       lis:RealizableEntity
   Range:
       lis:Activity
   InverseOf:
       lis:realizes
```

Used to state that a function is realized in an activity. In class constraints, this allows for characterization of intended performance of physical objects.

Description to be completed.

The object property *realizedIn* is set to be the inverse of the object property *realizes*.

A.3.28 realizes

```
ObjectProperty: lis:realizes
   Annotations:
       rdfs:label "realizes",
       rdfs:seeAlso <http://purl.obolibrary.org/obo/BFO_0000055>,
       skos:prefLabel "realizes"
   InverseOf:
       lis:realizedIn
```

Description to be completed.

The object property *realizes* is set to be the inverse of the object property *realizedIn*.

A.4 Data properties

The ISO 15926-14 ontology has three data properties. Figure 3 presents an overview of the data properties.

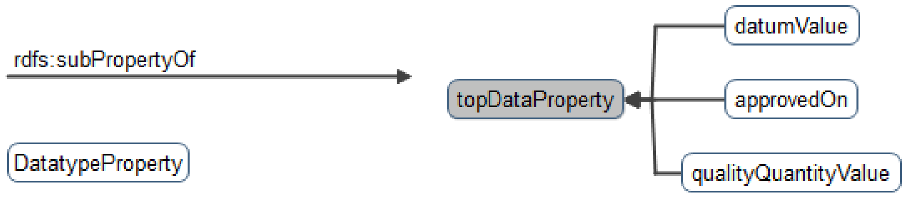


Figure 3 - Data properties of ISO 15926-14

A.4.1 approvedOn

```
DataProperty: lis:approvedOn
   Annotations:
       rdfs:comment "This is a super-property for stating the time that an entity was approved,
derived from Part 2 \"approval\". Introduce sub-properties to match different contexts and types
of approval. The range of sub-properties should be xsd:date or xsd:dateTime.",
       rdfs:label "approvedOn", skos:prefLabel "approvedOn"
```

This is a super-property for stating the time that an entity was approved.

Description to be completed.

A.4.2 datumValue

```
DataProperty: lis:datumValue
   Annotations:
       rdfs:comment "This relation is inspired by the relation \"has measurement value\" of the
Information Artefact Ontology.",
       rdfs:label "datumValue", skos:prefLabel "datumValue"
```

This property is used to assign, generally numerical, data typed values to measurements (estimates, etc.).

Description to be completed.

A.4.3 qualityQuantityValue

```
DataProperty: lis:qualityQuantityValue
   Annotations:
       rdfs:comment "This is a super-property for \"shortcut\" relations that combine a quality,
the weak lis:qualityQuantifiedAs, and a unit of measure into a simple data property. For instance,
\"mass in kilograms\" can be introduced as such a data property, for expressing the mass of an
entity on the kilogram scale. lis:qualityQuantifiedAs is weak in the sense that it doesn't
distinguish between designed or estimated, and measured, values.",
       rdfs:label "qualityQuantityValue", skos:prefLabel "qualityQuantityValue"
```

This is a super-property for “shortcut” relations that combine a quality and a unit of measure into a simple data property. Such shortcuts are highly recommended for large ontologies, where complexity of the model needs to be kept low to allow for tractable reasoning. For instance, “mass in kilograms” can be introduced as such a data property.

Description to be completed.

A.5 Annotation properties

The ISO 15926-14 ontology has two sets of annotation properties.

- The *originatesFrom* annotation property, and subproperties, is for recording changes to resources between versions of an ontology,
- The *relatedEntity* annotation property is for providing light-weight pointers to semantically similar entities in external vocabularies, in particular ontologies.

Figure 4 presents an overview of the annotation properties.

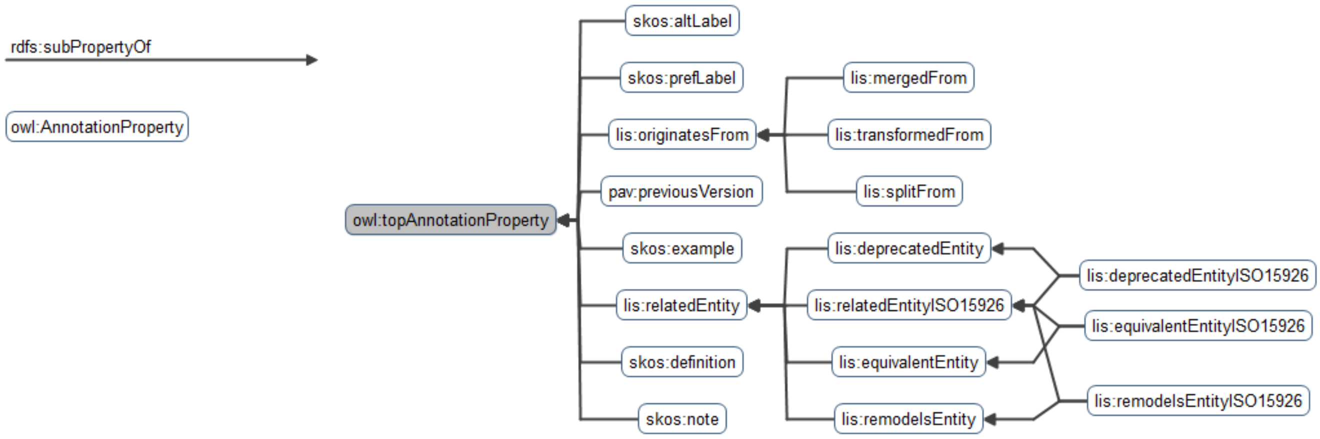


Figure 4 - Annotation properties of ISO 15926-14

A.5.1 originatesFrom

```
AnnotationProperty: lis:originatesFrom
   Annotations:
       rdfs:comment "This is a super-property for associating a resource with a resource it
originated from.",
       rdfs:label "originatesFrom", skos:prefLabel "originatesFrom"
```

This is a super-property for associating a resource with a resource it originated from.

The annotation property *originatesFrom* has three sub-properties, *transformedFrom*, *mergedFrom* and *splitFrom*.

A.5.1.1 originatesFrom > transformedFrom

```
AnnotationProperty: lis:transformedFrom
   Annotations:
       rdfs:comment "This annotation property is used for stating that the current resource
originates from another resource by transformation.",
       rdfs:label "transformedFrom", skos:prefLabel "transformedFrom"
   SubPropertyOf:
       lis:originatesFrom
```

This annotation property is used for stating that the current resource originates from another resource that has been transformed.

Description to be completed.

For an example of *transformedFrom* in context, see use case F.6.

A.5.1.2 originatesFrom > mergedFrom

```
AnnotationProperty: lis:mergedFrom
   Annotations:
       rdfs:comment "This annotation property is used for stating that the current resource
originates from one or several other resources by merging.",
       rdfs:label "mergedFrom", skos:prefLabel "mergedFrom"
   SubPropertyOf:
       lis:originatesFrom
```

This annotation property is used for stating that the current resource originates from one or several other resources that have been merged.

Description to be completed.

For an example of *mergedFrom* in context, see use case F.6.

A.5.1.3 originatesFrom > splitFrom

```
AnnotationProperty: lis:splitFrom
   Annotations:
       rdfs:comment "This annotation property is used for stating that the current resource
originates from another resource by splitting.",
       rdfs:label "splitFrom", skos:prefLabel "splitFrom"
   SubPropertyOf:
       lis:originatesFrom
```

This annotation property is used for stating that the current resource originates from another resource that has been split.

Description to be completed.

For an example of *splitFrom* in context, see use case F.6.

A.5.2 relatedEntity

```
AnnotationProperty: lis:relatedEntity
    Annotations:
        rdfs:comment "Annotation property for referring from OWL ontology resources to entities in
external vocabularies (typically, but not necessarily, ontologies), where definitions, examples,
or explications of the external entity contribute to understanding and defining the local
resource.",
        rdfs:label "relatedEntity"
```

A.5.2.1 relatedEntity > relatedEntityISO15926

```
AnnotationProperty: lis:relatedEntityISO15926
    Annotations:
        rdfs:comment "Annotation property for referring from OWL ontology resources in ISO 15926-14 to entities in other parts of ISO 15926, where definitions, examples, or explications of the Part 2 entity contribute to understanding and defining the Part 14 resource.",
        rdfs:label "relatedEntityISO15926",
        skos:example "The Part 14 class Activity is clearly related to, but not equivalent to, the Part 2 entity ClassOfActivity, and this annotation property should be used to record that fact."
    SubPropertyOf:
        lis:relatedEntity
```

A.5.2.2 relatedEntity > remodelsEntity

```
AnnotationProperty: lis:remodelsEntity
    Annotations:
        rdfs:comment "Annotation property for referring from OWL ontology resources to entities in external vocabularies, where the local resource, together with OWL constructs, can express intuitively the same facts as the referenced entity.",
        rdfs:label "remodelsEntity"
    SubPropertyOf:
        lis:relatedEntity
```

A.5.2.2.1 relatedEntity > remodelsEntity > remodelsEntityISO15926

```
AnnotationProperty: lis:remodelsEntityISO15926
    Annotations:
        rdfs:comment "Annotation property for referring from OWL ontology resources in ISO 15926-14 to entities in other parts of ISO 15926 (primarily Part 2), where the Part 14 entity type, together with OWL constructs, can express intuitively the same facts as the referenced entity.",
        rdfs:label "remodelsEntityISO15926",
        skos:example "Range restrictions on physical quantities can be captured with OWL and Part 14 using \"facet\" datatype restrictions. These will correspond to Part 2 Property Range expressions."
    SubPropertyOf:
        lis:relatedEntityISO15926,
        lis:remodelsEntity
```

A.5.2.3 relatedEntity > equivalentEntity

```
AnnotationProperty: lis:equivalentEntity
    Annotations:
        rdfs:comment "Annotation property for referring from OWL ontology resources to entities in external vocabularies, where the local resource is equivalent in meaning to that of the referenced entity.",
        rdfs:label "equivalentEntity"
    SubPropertyOf:
        lis:relatedEntity
```

A.5.2.3.1 relatedEntity > equivalentEntity > equivalentEntityISO15926

```
AnnotationProperty: lis:equivalentEntityISO15926
    Annotations:
        rdfs:comment "Annotation property for referring from ontology resources in ISO 15926-14 to
entities in other parts of ISO 15926, where the Part 14 resource is equivalent in meaning to that
of the referenced entity.",
        rdfs:label "equivalentEntityISO15926",
        skos:example "The Part 14 class Activity has, intuitively, the same meaning as the Part 2
entity Activity, and this annotation property should be used to record that fact."
    SubPropertyOf:
        lis:equivalentEntity,
        lis:relatedEntityISO15926
```

A.5.2.4 relatedEntity > deprecatedEntity

```
AnnotationProperty: lis:deprecatedEntity
    Annotations:
        rdfs:comment "Annotation property for referring from OWL ontology resources to
semantically similar entities in external vocabularies, where the limitations of OWL, as as first-
order language, imply that representing the intended meaning of the referenced entity is not
feasible.",
        rdfs:label "deprecatedEntity"
    SubPropertyOf:
        lis:relatedEntity
```

A.5.2.4.1 relatedEntity > deprecatedEntity > deprecatedEntityISO15926

```
AnnotationProperty: lis:deprecatedEntityISO15926
    Annotations:
        rdfs:comment "Annotation property for referring from ontology resources in ISO 15926-14 to
semantically similar entities in other parts of ISO 15926 (primarily Part 2), where the
limitations of OWL, as as first-order language, imply that Part 14 is not capable of expressing
the intended meaning of the referenced entity.",
        rdfs:label "deprecatedEntityISO15926",
        skos:example "Modal notions can not be captured in OWL. Therefore, Part 2 entities that
are essentially modal, such as those referring to \"actuality\" or \"materialization\", need to be
left out of Part 14."
    SubPropertyOf:
        lis:deprecatedEntity,
        lis:relatedEntityISO15926
```

Annex B
(informative)

Mapping of entity types from ISO 15926-14 to ISO 15926-2

An electronic insert to ISO 15926-14 documents the relationship between entity types in ISO 15926-2 and ontology resources in ISO 15926-14: the MS Excel spreadsheet

N-2782_mapping between ISO 15926-14 and ISO 15926-2.xlsx.

The spreadsheet includes 4 worksheets. Both the EXPRESS schema and the accompanying documentation of ISO 15926-2 have been considered in creating the re-representation in ISO 15926-14.

Worksheet "Remodelled in OWL" includes two tables. The first lists ISO 15926-2 entity types for which natural translations into OWL resources have been found: there are 36 of these. The second gives a tentative indication of how, for a selection of ISO 15926-2 entity types that resist direct translation, patterns of OWL resources to may be employed to obtain models with closely equivalent interpretations.

Worksheet "Metamodelling using SKOS" lists ISO 15926-2 "class of class" entity types for which corresponding classes may be introduced, extending the ISO 15926-14 ontology, using the OWL *punning* mechanism for representing classes, object properties, and data properties as individuals.

Worksheet "Represent in RDL" lists ISO 15926-2 entity types that are deemed inappropriate for inclusion in the upper ontology of ISO 15926-14.

Worksheet "Out of scope" includes four tables. The first lists ISO 15926-2 entity types for which standard OWL devices provide native replacements. The second lists entity types for mathematical concepts, which are better included in ISO 15926-3. The third lists entity types that are replaced by OWL primitives for identifiers, natural languages, and data types. The fourth lists entity types that are modal in nature, and therefore not suitable for inclusion in a the first-order language of OWL DL.

Annex C
(informative)

Reference to electronic version of ontology

The version of the ontology that is defined by this document is attached:

LIS-14.ttl

The ontology is provided as an OWL/RDF file in *Turtle*[4] format.

Editor's Note: Extend this section with information on any format details, as well as description logic complexity of the ontology.

[4] https://www.w3.org/TR/turtle/

Annex D
(informative)

Naming conventions

The following conventions are aimed to help users with the naming of terms.

To be completed.

D.1 Data property shortcuts

Data property shortcuts serve the purpose …

Example: Hammer, data property *mass_kilogram* or *mass_measured_kilogram*

Pattern: (closest) type for the quality + '_' + unit of measure

Refer to use case on units of measure.

To be completed.

D.2 Inverse relations

To be completed.

Annex E
(informative)

Reasoning

E.1 Automated classification from property values

This example illustrates how one can use OWL 2 reasoning to classify objects (a.k.a. individuals in the OWL terminology) by means of formalized definitions of classes. Assume, for the sake of the example, that a hammer considered to be "big" if its weight in kg is more than 1 and that it is otherwise "small". Assume two objects, referred to as "hbig" and "hsmall" with measured weights of, respectively, 4.7 and 0.3 kg. We want to use reasoning to infer that hbig is big and hsmall is small. Furthermore, the representation should distinguish these measurements from other information about the hammers' weight like different measurements and information in data sheets.

Intuitively, a hammer has a mass and it may have different data associated to its mass, e.g. measured in different points of time or with different units (kilograms, stones). Figure 2 summarizes some main points of the example, which is explained in detail in the rest of the section[5].

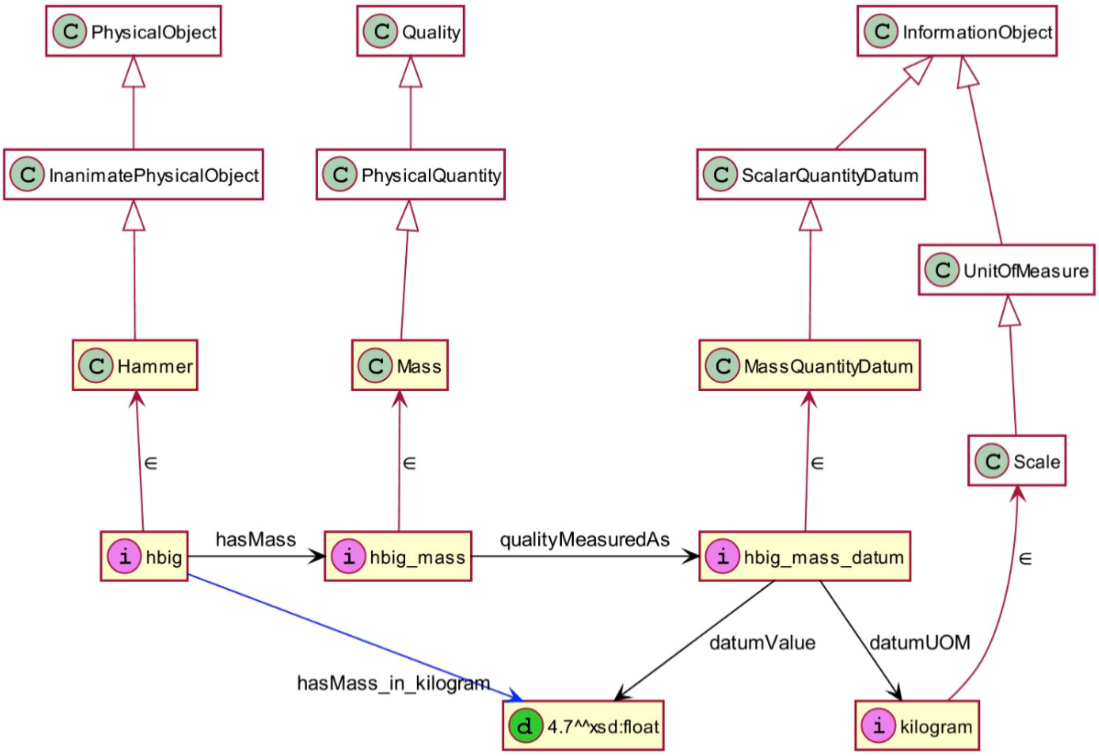


Figure 5- Hammer mass representation

[5] This figure has been revised and extended in Section G.9 to discuss additional details of individuals of type *ScalarQuantityDatum*

E.2 Information objects

In order to make the appropriate inferences in OWL the weight information must be represented as properties of an object. ISO 15926-14 takes this object to be an *InformationObject*. More precisely *InformationObject* is a superclass of *QuantityDatum*, and *QuantityDatum* a superclass of *ScalarQuantityDatum*. The latter two classes are inspired by the Information Artefact Ontology and its classes "measurement datum" and "scalar measurement datum"; the change of wording from "measurement" to "quantity" is intended to support cases where measurement is not involved, such as with nominal values in data sheets. Here are the formal definitions:

```
Class: QuantityDatum
       SubClassOf: InformationObject
Class: ScalarQuantityDatum
       SubClassOf: QuantityDatum,
              datumValue some rdfs:Literal and datumUOM some UnitOfMeasure
```

A scalar quantity datum has a unique unit of measure and a unique numeric value. The *datumUOM* is an object property used to carry information about the unit of measure:

```
ObjectProperty: datumUOM
       Domain: QuantityDatum
       Range: UnitOfMeasure
       Characteristics: Functional
```

Note that *UnitOfMeasure* is itself a subclass of *InformationObject*.

Going back to the example, the object that carries the information "4.7 kg" carries information about the hammer's mass. To capture this point, we introduce a subclass of *ScalarQuantityDatum* called *MassQuantityDatum*:

```
Class: MassQuantityDatum
       SubClassOf: ScalarQuantityDatum,
              datumValue some float and datumUOM some Scale and datumTime some xsd:date
```

The object property *datumTime* not predefined in P14, but we can freely add it on demand.

The value of *datumValue* has the XSD datatype float. Similarly, *date* is also an XSD datatype. Wherever available, XSD datatypes should be used in ontologies based on ISO 15926-14. The object property *datumTime* is not predefined in ISO 15926-14 and can be added on demand with a datatype that has the appropriate granularity for the case at hand. For instance, *xsd:date* is a calendar date while *xsd:dateTime* includes hour, minute, second.

The object property *datumUOM* links to a *Scale*, one of whose instances is "kilogram". Note that "kilogram" is an object (or: OWL individual). When comparing the types of the object property restrictions of *MassQuantityDatum* to the corresponding restrictions of *ScalarQuantityDatum* we see that the values of the former are more restricted than the values of latter: *float* is more restrictive than *rdfs:Literal* and *Scale* is more restrictive than *UnitOfMeasure*. By narrowing down the space of admissible values in this way, more errors in the data can potentially be detected by a reasoner.

The information objects of the example are thus represented as follows:

```
Individual: hbig_mass_datum
       Types: MassQuantityDatum
       Facts: datumUOM kilogram, datumValue 4.7f
Individual: hsmall_mass_datum
       Types: MassQuantityDatum
       Facts: datumUOM kilogram, datumValue .3f
```

E.3 Qualities

The datum individuals record measurements of the hammers' mass. The intention of ISO 15926-14 is to take *Mass* as a subclass of *PhysicalQuantity*. The class *Quality* and its subclass *PhysicalQuantity* in are directly inspired by corresponding classes included in the DOLCE and BFO upper ontologies.

A Hammer, its relation to mass, and the relation between the mass and *QuantityDatum* in evidence could be formally represented as follows:

```
Class: Hammer
       SubClassOf: hasMass some Mass
Class: Mass
       SubClassOf: qualityQuantifiedAs some MassQuantityDatum
```

In a real example, one would expect to inherit the restriction with the *hasMass* property from a superclass of *Hammer* instead of being defined for *Hammer;* this will work since such restrictions propagate downwards through the subclass hierarchy. Note that there can be several distinct instances of *MassQuantityDatum* related to a given instance of *Mass*. The hammer and mass individuals in Figure 2 are represented in this way:

```
Individual: hbig
       Types: Hammer
       Facts: hasMass hbig_mass
Individual: hbig_mass
       Types: Mass
       Facts: qualityMeasuredAs hbig_mass_datum
Individual: hsmall
       Types: Hammer
       Facts: hasMass hsmall_mass
Individual: hsmall_mass
       Types: Mass
       Facts: qualityMeasuredAs hsmall_mass_datum
```

Note that the *qualityQuantifiedAs* property is recorded by means of the subproperty *qualityMeasuredAs.* This way one can capture the distinction between measurements and nominal values of data sheets.

E.4 Necessary and sufficient conditions

By stipulation a big hammer is a hammer that weights more than 1 kilogram, while small hammers must weight 1 kilogram or less. This could be formally represented as follows:

```
Class: BigHammer
       SubClassOf: hasMass some (qualityQuantifiedAs
              some (datumUOM value kilogram and datumValue some float[> 1]))
Class: SmallHammer
       SubClassOf: hasMass some (qualityQuantifiedAs
              some (datumUOM value kilogram and datumValue some float[<= 1]))
```

The above two OWL 2 axioms represent necessary but not sufficient conditions for establishing class membership of an individual. For example, the individuals *hbig* and *hsmall*, as declared above, will not be classified as *BigHammer*, respectively *SmallHammer*, as one would expect.

In order to enable the desired inference, sufficient conditions are also required. This could easily be achieved by declaring *BigHammer* as *EquivalentTo* the restriction instead of *SubClassOf*. An alternative would be to add a reversed *SubClassOf* axioms (i.e. the other side of the equivalence).

```
Class: hasMass some (qualityQuantifiedAs
              some (datumUOM value kilogram and datumValue some float[> 1]))
  SubClassOf: BigHammer
Class: hasMass some (qualityQuantifiedAs
              some (datumUOM value kilogram and datumValue I float[<= 1]))
       SubClassOf: SmallHammer
```

Apart from enabling additional inferences, sufficient conditions have typically a representation closer to *rules* which will enhance OWL 2 reasoning. Necessary conditions may also be considered as restrictions over the data (i.e. integrity constraints) which could also be represented as rules. Interested readers can refer to the following references [4, 15].

E.5 Adding shortcuts

Simplifying a representation can sometimes speed up reasoning considerably. The data property *has_mass_in_kilogram* shortcuts the model in Figure 2.

For large scale ontologies efficiency and scalability of reasoning will sometimes require simplifications in the ontology, typically by leaving out classes and properties that will slow down the reasoning engine. ISO 15926-14 includes some useful shortcuts that allows semantically significant information to be represented at a higher level of granularity, should this be needed.

E.6 Functions

Most of the physical things that we wish to describe in a store of industrial data will have a *function* – an intended purpose. This includes structural elements of a factory, equipment, and instruments.

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". The shape of the sentence can guide us to a modelling example for an ontology-based representation: "A Hammer *x* has a function that is realized in nail-driving activities where *x* has the tool role".

Figure 3 shows a graphical representation of the modelling example. Note that *NailDriving* is a subclass of *Activity* where a hammer *h* has a function *f* which is realized in the nail-driving activity *d.* Ensuring that hammer functions are only realized in nail driving processes where the hammer is active as a tool is clearly important (i.e. the link between *h* and *d*).

We can include appropriate constraints on the individuals by means of a constraint on *hbig_f*:

```
Individual: hbig_f
      Facts:  memberOf (realizedIn only (nailing and inverse(toolIn) value hbig))
```

But there is no way to express this constraint at the level of *class Hammer* to the effect that any member of that class meets the constraint, due to the back-reference implicit in the appearance of *hbig*.

If one wishes to characterize systems beyond the part/whole breakdown (cf the example in Section 8) this has to be done via functions and their realizations. This allows for the interdependencies between the functional parts of the systems to be modelled and be reasoned about.

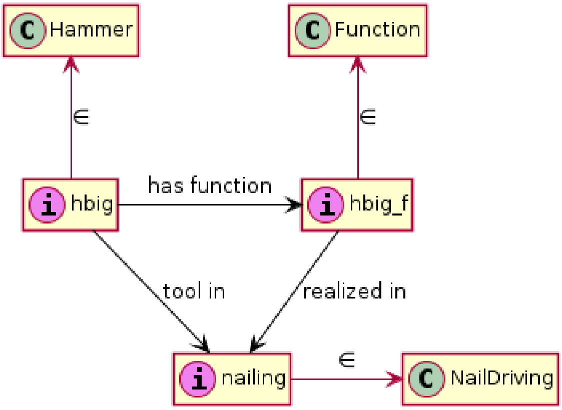


Figure 6 – Hammer function example

E.7 Example: Checking conformance with requirements

This example illustrates how reasoning can assist in checking conformance of product types with restrictions from design, and how product individuals can be related to functional objects through object properties.

E.8 System, Functional objects and Physical objects

The starting point is a system *s*, with functional parts *a* and *b* represented by the object property *functionalPartOf*. Both the system *s* and its functional parts *a* and *b* are classified as *FunctionalObject*. A functional object could be a tag, or an object identified in the design process at a point before tags are introduced.

Furthermore, the functional objects *a* and *b* are also classified as *PhysicalObject*; *a* is classified as instance of *Driver* and *b* as an instance of *PowerSource*; these are both quite high-level characterizations. For *a*, we assert that it has output power of at least 850 W. This is illustrated in Figure 4 below:

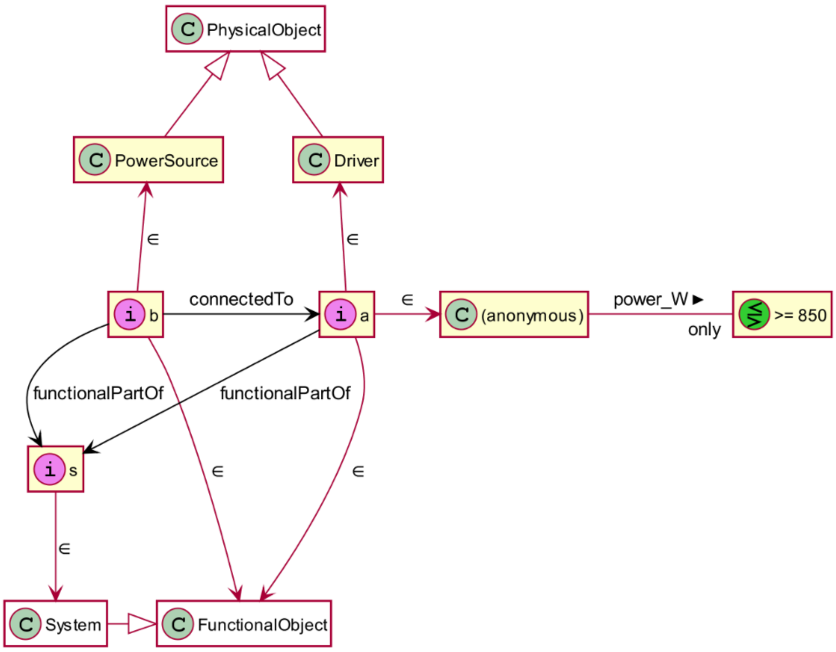


Figure 7 – System, functional objects and classifications as physical objects

An important point is that the breakdown structure of the physical objects may be very different from the system breakdown structure captured by the object property *functionalPartOf*. The physical breakdown structure is not illustrated, but could be captured by part/whole properties, e.g., *arrangedPartOf*.

E.9 Developing a design

Working from the functional description of the system, the designers develop a detailed type description, exploiting the available reference data library (RDL). It turns out their RDL contains the class *ElectricMotor*. What they need is a class that, among other things, carries the power restriction. Besides this the class could restrict with respect to a number of other factors like size, vibration and noise, but these are, for the sake of example, neglected in the following.

The designer search in the RDL for a class called “DriverABC850” without any hit. But before they add the class that they want to the RDL they provide its formal definition to the reasoner to check if the RDL contains a class with an equivalent definition. The check was positive, the RDL contains a class called *DriverABC* which is defined exactly as they wanted, although with an unexpected. Note that this check prevented them from creating a duplicate class in the RDL. The design can now be accurately captured by the class *El.MotorABC850*, which happened to extend the RDL and was hence added.

Figure 5 illustrates the new modeling of the design.

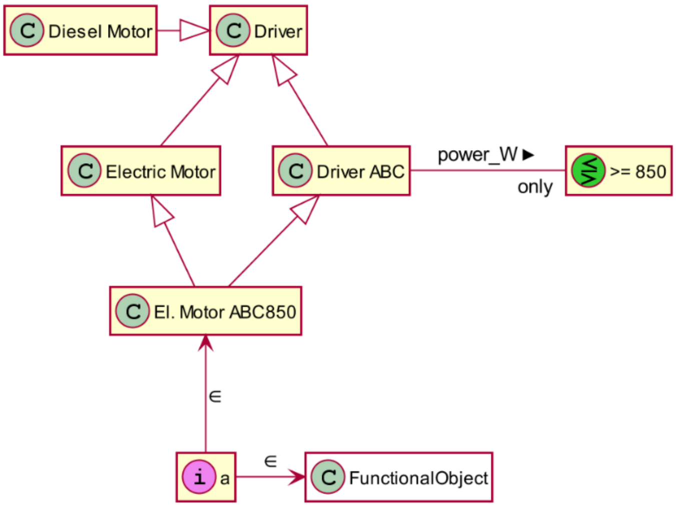


Figure 8 – Design classes for the motor

Note that the functional individual *a* will be constrained according to all superclasses of *El. Motor ABC850*; in particular, the output power restriction is now inherited by that class.

E.10 Design and replaceable parts: Adding product individuals

The RDL contains a product model which contains two product types as illustrated in Figure 6.

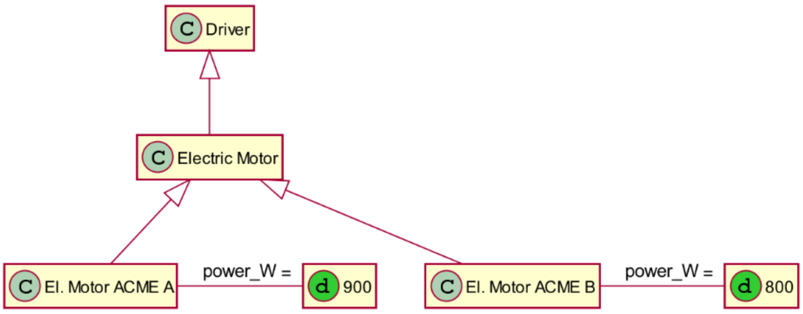


Figure 9 – Product types

Before handing over to procurement the engineers want to test the effect of using these product types and introduce three product individuals, see Figure 7. The product individuals are related to the functional object a via the *installedAs* property. In the present model no information is captured about the nature of the installation and, e.g., at what time the product is installed at the functional (tag) position. Such information can, though, easily be added.

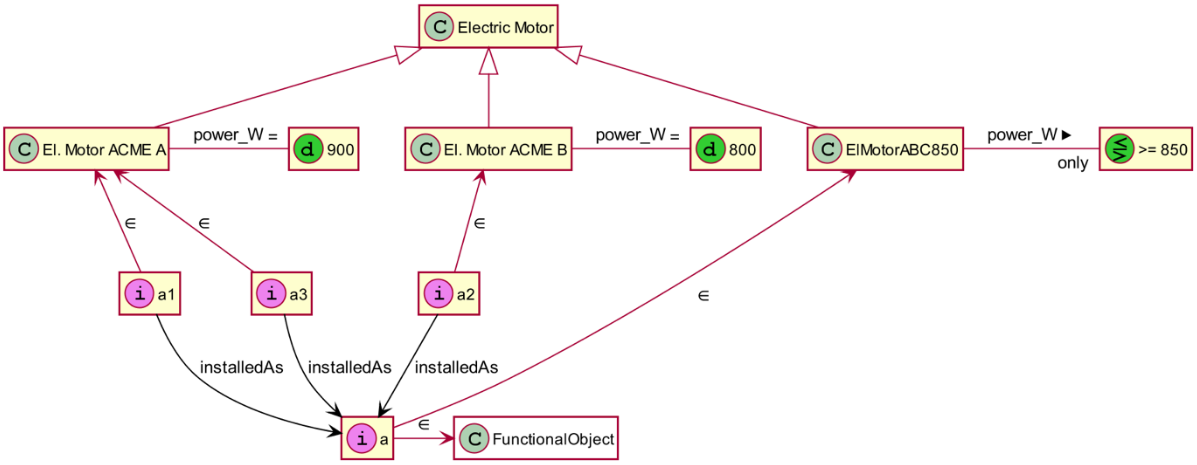


Figure 10 – Installing product individuals at the position of the functional object

E.11 Detecting the inconsistency

Using automated reasoning we can check whether the requirements laid down in a design are satisfied by the installed parts. 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. There are different solutions to check conformance requirements.

First, we can *check emptiness or disjointness* between the component specification class and the classes describing the concrete specifications of a model. For example, the intersection between *ElMotorABC850* and *ElMotorACME_A* is non empty since the *ACME A model* satisfies the requirements (delivers 900 watts) while the intersection between *ElMotorABC850* and *ElMotorACME*_B is empty since *ACME B model* does not meet the requirements (delivers only 800 watts).

Second, we can use *individual substitution* as illustrated in Figure 8. This can be done by selecting "concrete" individuals of a model and substituting them by the targeted design objects. For the example given, we substitute the replaceable parts *a1*, *a2* and *a3* for the design object a. The effect of substitution is that we combine all the requirements of the design with all the characteristics of the product specimens. Substitution can be simulated by adding statements of the type:

```
Individual: a
        SameAs: a2
```

As there is a conflict, the reasoner will discover an inconsistency. The difference with respect to the previous solution is that the concrete individuals of a model may bring additional characteristics to meet the design requirements.

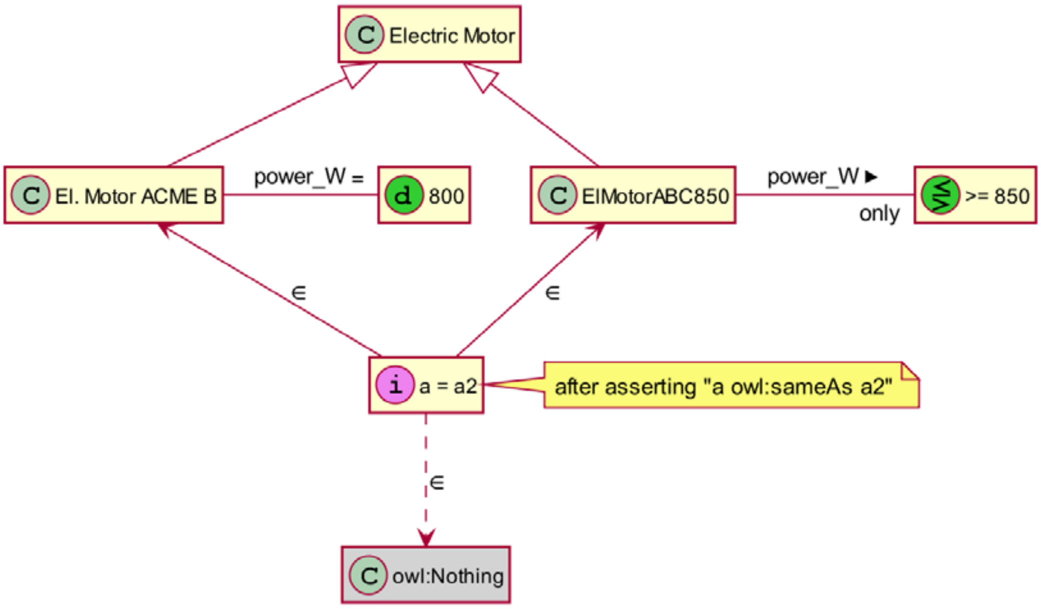


Figure 11 – Inconsistency detected by reasoning

Alternatively, instead of finding conflicts between the requirements and the concrete products, one could try to classify the concrete product individuals and model specifications under the component requirement specification. To this end, as in the example in Section 6, sufficient conditions are required. Adding the following sufficient condition to our example would classify *ElMotorACME_A* under *ElMotorABC850* and thus the replaceable parts a1 and a3 will also be members of *ElMotorABC850*.

```
Class: ElMotor and power_watts some float[>= 850]
       SubClassOf: ElMotorABC850
```

Annex F
(informative)

Lifecycle information

F.1 Lifecycle scenario of an object

This section illustrates the approach for modelling lifecycle information. The approach captures the lifecycle model as suggested in ISO 81346-1. Three different aspects of lifecycle information are being accounted for: evolving specifications, instantiations and resources.

Consider the lifecycle scenario of an object in Annex B.2 of ISO 81346-1. The scenario describes a concrete lifecycle story taking, without loss of generality, the occurrence of pump motor in an industrial process as an example. The story begins with the creation of a functional object that fulfills the need for pumping. The object is then associated with Function Requirements (FR) specifying such details as rated power, etc. Next, the location for the object is specified as well as the Component Type (CT) and a Product Specification (PS) is given. Once the specifications are in place, an actual motor, i.e. a physical object, is installed and, later, replaced by another motor as part of a maintenance policy.

This lifecycle story is illustrated in Figure 12 below (which is a modification of Figure B.2 in ISO 81346-1). The figure only shows Situation N of the story. The *aspect cube* (top) stands for a stable reference to the functional object. The coloured sides of the cube represent the different aspects related to function, location and product specification as distinguished in ISO 81346-1. Representing these aspects using the ISO 15926-14 ontology is illustrated in G.6.

Editor's note: The following are candidates for being promoted from the present, informative, annex to be included in the ISO 15926-14 ontology itself, because they are essential for life-cycle information.

- Object property *specifiedIn,*
- Class *Specification*, from which subclasses would be introduced to cover a use case; here, the use case is ISO/IEC 81346, motivating classes *Function Requirement*, etc.
- Annotation property *partOfSpecification*, from which subproperties would be introduced to cover the variants of specification in scope for a use case.

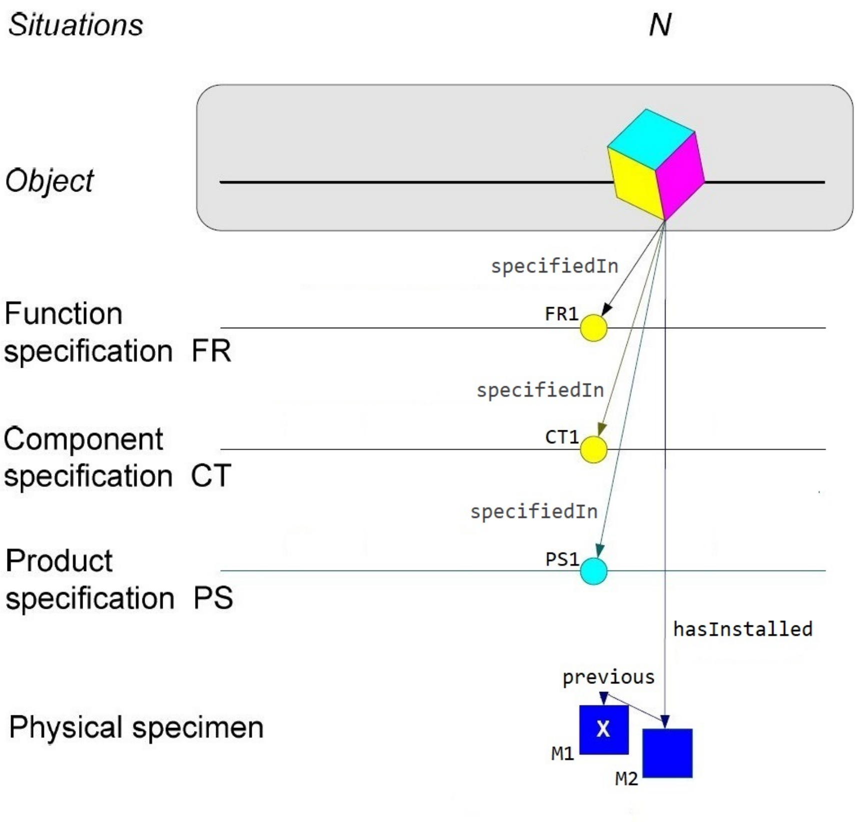


Figure 12 - Situation N of the object's life cycle in ISO 81346-1

The accounts for representing the three aspects of lifecycle information using the ISO 15926-14 ontology are described in the following.

F.1.1 Evolving specifications

The aspect cube is represented in Figure 13 by the individual *driver1*, which is classified as *FunctionalObject* in the ISO 15926-14 ontology. Note that *driver1* is a stable IRI, i.e., it does not change over time.

The function requirement FR1, the component specification CT1 and the product specification PS1 are represented by the individuals *FR1*, *CT1* and *PS1*, respectively. The association between the cube and the function requirement, shown as an arrow from the cube to FR1 (yellow circle) in Figure 12, is modelled using the object property *specifiedIn*. Similarly, the associations between the cube and the component and product specifications, shown as arrows from the cube to CT1 (yellow circle) and PS1 (blue circle), are modelled using *specifiedIn* as well.

Suppose the requirement FR1 states that the consumption of the motor should be at most 5.0 kW. The restriction on the power consumption is modelled using the shortcut data property *consumption_kW* from *driver1* to the value 5.0 of type *xsd:float*. The data property in turn is annotated using the annotation property *partOfFuncSpec* with *FR1*; see Figure 13. A convention for naming shortcut data properties is presented in D.1.

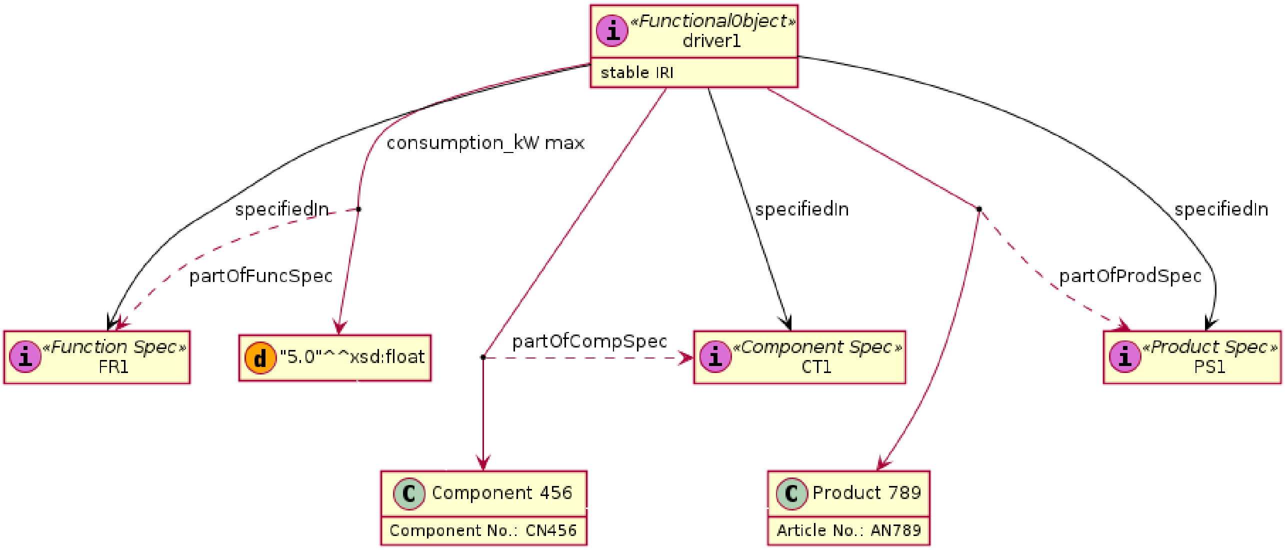


Figure 13 - Associating specifications to a design object

Suppose further that component specification CT1 states that a component with number CN456 is to be used for the motor. This is modelled by asserting that *driver1* is of the class "*Component 456*", whose members have the component number "CN456" as an attribute. This association is annotated with *CT1* using the annotation property *partOfCompSpec*.

Finally, suppose that the product specification PS1 states that the concrete product with number AN789 is to be used for the motor. This is modelled by asserting that *driver1* is of the class "*Product 789*", whose members have the product number "AN789" as an attribute. This association is annotated with *PS1* using the annotation property *partOfProdSpec*.

F.1.2 Evolving instantiation

Once the specifications have been made, the functional object is to be instantiated by a physical object; see Figure 14. This is shown by an arrow from the cube to the motor M2 (blue rectangle) in Figure 12. This is modelled using the object property *installedAs* (blue arrow) that connects *driver1* with the individuals *M1* and *M2.* For more details on how to use the *installedAs* property to detect inconsistencies between specifications and implementation, see E.11.

Suppose that M2 is already the second motor that replaces the first motor, M1 in Figure 12, as part of a maintenance policy. This is modelled by connecting *M2* with the individual *M1*, which represents motor M1, via an annotation property, *previous* (black arrow). The annotation property is application specific and, therefore, it is not specified in the ISO 15926-14 ontology. This is illustrated in Figure 14.

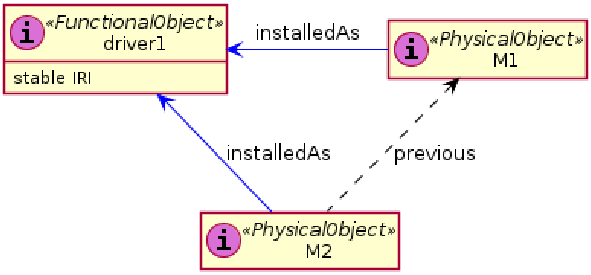


Figure 14 - Evolution of instantiations

F.1.3 Embedded specifications and instantiations

Figure 15 shows the current state of the specifications and instantiations embedded into the same model. The connections to the ISO 15926-14 upper ontology are emphasised. In particular, the classes *FunctionalObject*, *InformationObject*, *PhysicalObject* and *Site* (classes with white background) and the object properties *residesIn* and *installedAs* (blue arrows) from the ISO 15926-14 ontology have been employed. The individual representing the cube, *driver1*, is classified as a *FunctionalObject*, the individuals representing the specifications FR1, CT1 and PS1 are each classified as *InformationObject*, and the individuals representing the motors M1 and M2 are each classified as *PhysicalObject*. Additionally, there an individual, *H-7B*, representing the location, where the driver is to be installed. This individual is classified as *Site*.

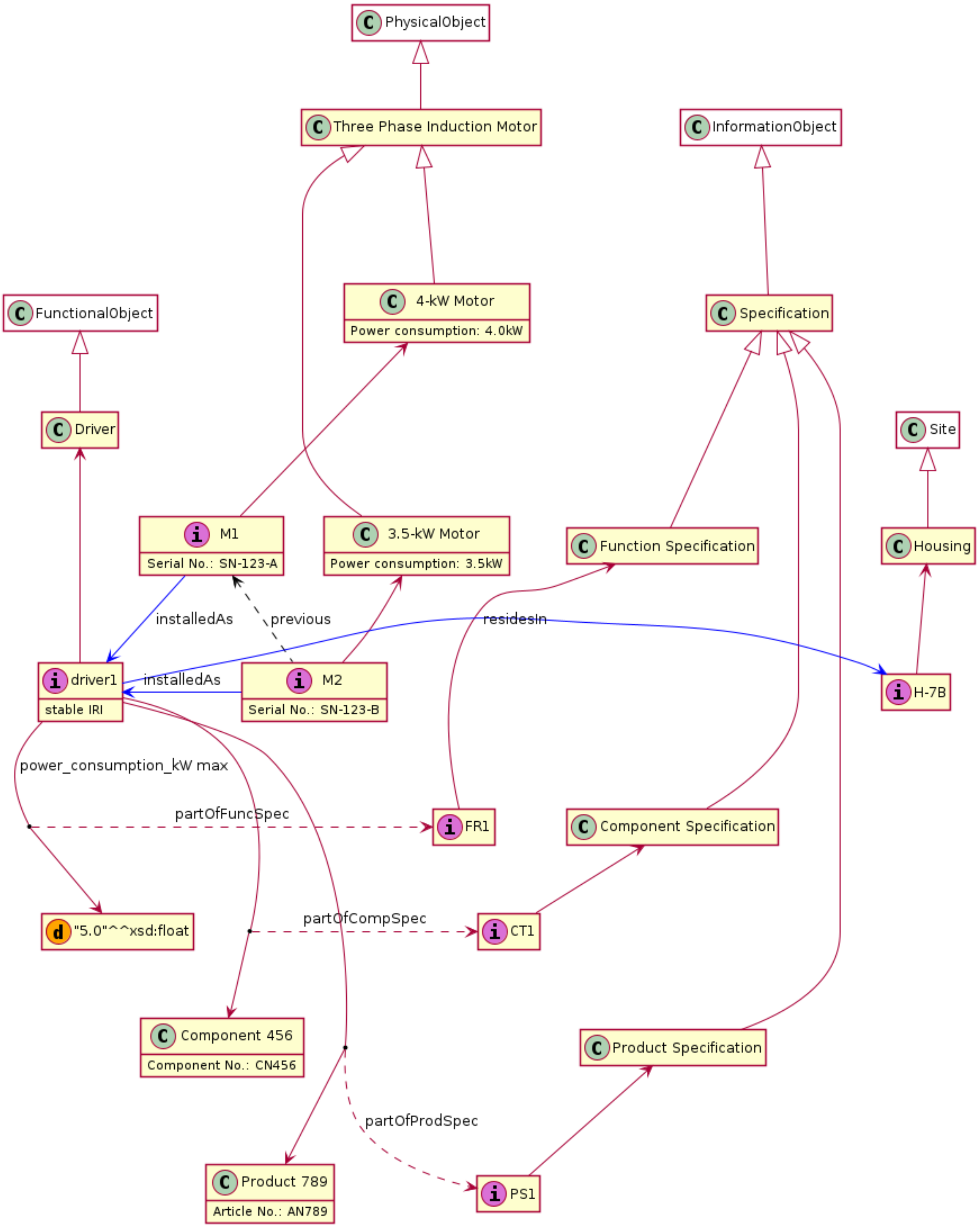


Figure 15 - Specifications and instantiations embedded into one model

F.2 Evolving resources

Industrial processes affect changes to resources by which they are transformed, merged or split apart.

Transforming a resource leads to changes of its properties. The changes may be so significant that the identifier used to refer to the resource before the change may no longer be used to refer to the resource after the change. Instead a new identifier is introduced. For instance, consider the manufacturing process of injection moulding plastic pellets into plastic cups. The process transforms pellets into cups by heating up the pellets and injecting the molten plastic into a cup-shaped mould under high pressure. The material the pellets and the cups are made of is the same, but it appears in a different form. The identifier used to refer to the pellets is no longer used to refer to the moulded cups.

The process of merging resources combines the resources in a specific way. The result is a new resource that receives its own identifier. While merging physical resources will usually consume the original resources, merging digital resources can leave the original resources intact. Therefore, consuming and non-consuming merge processes are distinguished. For instance, consider the process of dissolving a gas into a liquid, e.g., the process of carbonating water. Carbon dioxide gas is dissolved in water to produce carbonated (sparkling) water. The resulting carbonated water receives a new identifier and the identifiers for referring to the gas and the water are no longer used.

The process of splitting a resource leads to several new resources which consist of parts of the original resource. The resulting resources receive new identifiers. The splitting of a physical resource is a destructive process. On the other hand, the splitting of digital resources is not necessarily destructive. For instance, consider the manufacturing process of dividing raw material, e.g., the process of punching out work pieces from sheet metal. In such a process, a metal sheet is being divided into work pieces and scrap, all of which may receive their own identifiers. The reference to the metal sheet no longer points to an existing resource.

F.2.1 Modelling

The model focusses on the evolution of identifiers. The notion of change over time is modelled as a *revision history of ontology snapshots* instead of being modelled within an ontology. Each ontology snapshot represents a timeless state of the model. Links between snapshots are established using dedicated OWL annotation properties. ISO 15926-14 provides three OWL annotation properties for the change operations transform, merge and split, which are called *transformedFrom*, *mergedFrom* and *splitFrom, respectively*. The resources resulting from a change operation are annotated, pointing back to the resources they originated from. The ontology snapshot describing the state of the model after the change operation receives annotations that point back to its preceding snapshot.

F.2.1.1 Transform

A diagram is to be added here.

We introduce a vocabulary of machinery, materials, and processes for describing the states of affairs. This vocabulary is the same both before and after transformation.

```
Prefix: : <http://example.org/>

Class: lis:Activity
Class: lis:Site
Class: Polystyrene
Class: InjectionMouldingMachine_A
Class: Cup_A_Batch

ObjectProperty: lis:residesIn
ObjectProperty: hasInput
ObjectProperty: hasOutput
ObjectProperty: hasTool

DataProperty: grainSize_mm
DataProperty: mass_kg
DataProperty: piece_count

AnnotationProperty: lis:transformedFrom

Individual: injection_moulding_machine_1
    Types: InjectionMouldingMachine_A
Individual: silo_1
    Types: lis:Site
Individual: shelf_1
    Types: lis:Site
```

In the *before* scenario, a moulding process takes a batch of plastic pellets, the individual *plastic_granulate_1*, as input material.

```
Individual: plastic_granulate_1
    Types: PolystyrenePelletBatch, grainSize_mm only xsd:integer[>=4, <8],
      mass_kg value 10
    Facts: lis:residesIn silo_1

Individual: moulding_process_1
    Types: lis:Activity
    Facts: hasTool injection_moulding_machine_1, hasInput plastic_granulate_1
```

In the *after* scenario, the batch of plastic pellets is no longer an individual in the ontology: it has been transformed into a batch of plastic cups. The *transformedFrom* annotation on the batch of plastic cups records its origin in the batch of pellets.

```
Individual: plastic_cups_1
    Annotations: lis:transformedFrom <http://example.org/plastic_granulate_1>
    Types: Polystyrene, Cup_A_Batch
    Facts: piece_count 550

Individual: moulding_process_1
    Types: lis:Activity
    Facts: hasTool injection_moulding_machine_1, hasOutput plastic_cups_1
```

F.2.1.2 **Merge**

A diagram is to be added here.

We introduce a vocabulary of machinery, compounds, and processes for describing the states of affairs. This vocabulary is the same both before and after transformation.

```
Class: lis:Activity
Class: lis:Site
Class: CO2
Class: H2O
Class: CarbonatedWater
Class: CarbonatingMachine_A

ObjectProperty: lis:residesIn
ObjectProperty: hasInput
ObjectProperty: hasOutput
ObjectProperty: hasTool

DataProperty: mass_kg
DataProperty: volume_l

AnnotationProperty: mergedFrom

Individual: carbonator_1
    Types: CarbonatingMachine_A
Individual: pressure_gas_tank_1
    Types: lis:Site
Individual: liquid_tank_1
    Types: lis:Site
```

In the *before* scenario, a volume of gas, *carbon_dioxide_1*, is injected into a volume of water, *water_1*.

```
Individual: water_1
    Types: H2O
    Facts: lis:residesIn liquid_tank_1, volume_l 1000
Individual: carbon_dioxide_1
    Types: CO2
    Facts: lis:residesIn pressure_gas_tank_1, mass_kg 3

Individual: carbonation_process_1
    Types: lis:Activity
    Facts: hasTool carbonator_1, hasInput water_1, hasInput carbon_dioxide_1
```

In the *after* scenario, the volumes of gas and water are no longer individuals in the ontology: they have been merged into a volume of carbonated water. The *mergedFrom* annotation on the merged volume records its origin in the volumes of gas and water.

```
Individual: carbonated_water_1
    Annotations: lis:mergedFrom <http://example.org/water_1>,
      lis:mergedFrom <http://example.org/carbon_dioxide_1>
    Types: CarbonatedWater
    Facts: lis:residesIn liquid_tank_1, volume_l 1000

Individual: carbonation_process_1
    Types: lis:Activity
    Facts: hasTool carbonator_1, hasOutput carbonated_water_1
```

F.2.1.3 Split

A diagram is to be added here.

We introduce a vocabulary of machinery, materials, and processes for describing the states of affairs. This vocabulary is the same both before and after transformation.

```
Class: lis:Activity
Class: SteelPlate
Class: WorkPiece
Class: SheetMetalPress

ObjectProperty: lis:residesIn
ObjectProperty: hasInput
ObjectProperty: hasOutput
ObjectProperty: hasTool

DataProperty: thickness_mm

AnnotationProperty: lis:splitFrom

Individual: press_1
    Types: SheetMetalPress
```

In the *before* scenario, work pieces are punched from a sheet of steel.

```
Individual: steel_sheet_1
    Types: SteelPlate
    Facts: thickness_mm 4

Individual: blanking_process_1
    Types: lis:Activity
    Facts: hasTool press_1, hasInput steel_sheet_1
```

In the *after* scenario, the sheet of steel is no longer an individual in the ontology. Three blanks have been produced.

```
Individual: work_piece_1
    Types: WorkPiece
    Annotations: lis:splitFrom <http://example.org/steel_sheet_1>
Individual: work_piece_2
    Types: WorkPiece
    Annotations: lis:splitFrom <http://example.org/steel_sheet_1>
Individual: work_piece_3
    Types: WorkPiece
    Annotations: lis:splitFrom <http://example.org/steel_sheet_1>

Individual: blanking_process_1
    Types: lis:Activity
    Facts: hasTool press_1, hasOutput work_piece_1, hasOutput work_piece_2,
        hasOutput work_piece_3
```

Annex G
(informative)

Use cases

G.1 Events and alarms

This use case discusses how to represent machine events and alarms. This is relevant in particular for the representation and integration of data produced by sensors, which represent a major source of data available on the web today.

G.1.1 Status

Note from the Editors: This use case requires further development. In particular:

- Improve analysis of WoT TD, SOSA/SSN and iot.schema.org
- Refine current examples and add new examples to cover the notion of Event
- Support the analysis with examples using real data
- Provide an OWL ontology that represents the content of the modelling diagram
- Provide OTTR templates, and accompanying tables of example data

G.1.2 Relevant references

Relevant sources of information considered during the preparation of this use case are the ontologies SOSA/SSN[6], WoT Thing Description[7] and iot.schema.org[8]. The first two are W3C recommendations and Figures G.1.1 and G.1.2 have been extracted from these recommendations and therefore, they are protected under a license agreement[9].

W3C Web of Things activities aim to improve data interoperability for the Internet of Things (IoT). But in order to achieve this goal, it requires common semantic vocabularies. *WoT Thing Description* (WoT TD) [20] is a W3C recommendation that provides an *Information Model* with relevant terms and data structures for IoT. In particular WoT TD focuses on *things* and their *interaction affordances*. A *thing* is an abstraction of a physical or a virtual entity whose metadata and interfaces are described using WoT TD. These metadata and interfaces, known as *interaction affordances*, define how to interact with a thing. WoT TD defines three types of interaction affordances: *Properties*, *Actions* and *Events*. The format of the data used in WoT TD is described using a *data schema vocabulary*, which is based on *JSON Schema*[10]. Figure G.1.1 provides an overview of the core classes and properties of the WoT TD Information Model.

W3C Semantic Sensor Network Ontology [21] is a recommendation that aims to provide a coherent representation of entities, relations, and activities involved in sensing, sampling, and actuation. It specifies two ontologies: Semantic Sensor Network (SSN) and Sensor, Observation, Sample, and Actuator (SOSA) ontologies, where SOSA provides a lightweight core and SSN is an extension with a broader scope. In SOSA, the activities of observing, sampling, and actuating, each targets some feature of interest by either changing its state or revealing its properties, each follows some procedure, and each is carried out by some object or agent (namely a sensor, an actuator or a sampler, respectively). Figure G.1.2 provides

[6] https://www.w3.org/TR/vocab-ssn/

[7] https://www.w3.org/TR/wot-thing-description/

[8] http://iotschema.org/

[9] https://www.w3.org/Consortium/Legal/2015/doc-license

[10] https://tools.ietf.org/html/draft-handrews-json-schema-validation-01

an overview of the core classes and properties that are specifically related to modeling observations in SOSA/SSN. SOSA axioms are shown in green, while SSN-only axioms are shown in blue.

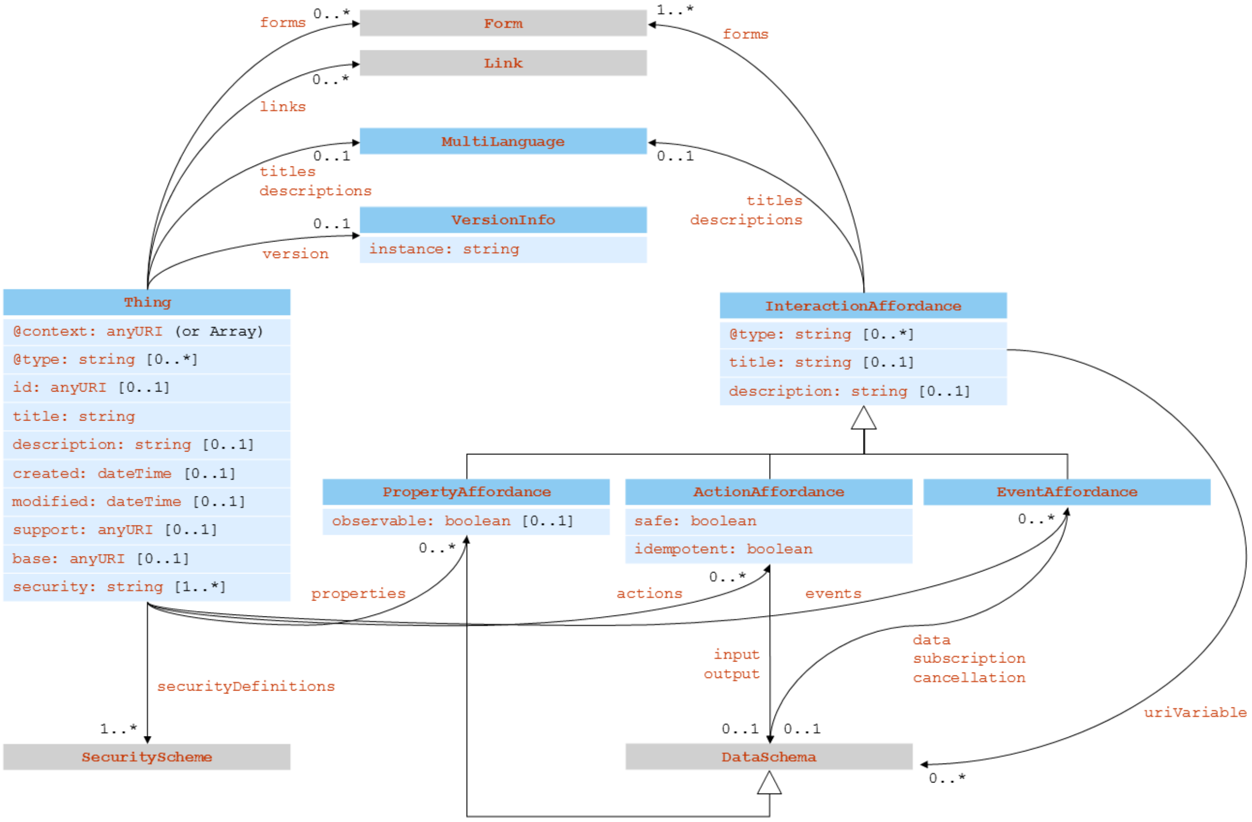


Figure G.1.1: WoT core vocabulary (Copyright © 2020 W3C ® (MIT, ERCIM, Keio), All Rights Reserved)[11]

[11] https://www.w3.org/TR/wot-thing-description/#overview

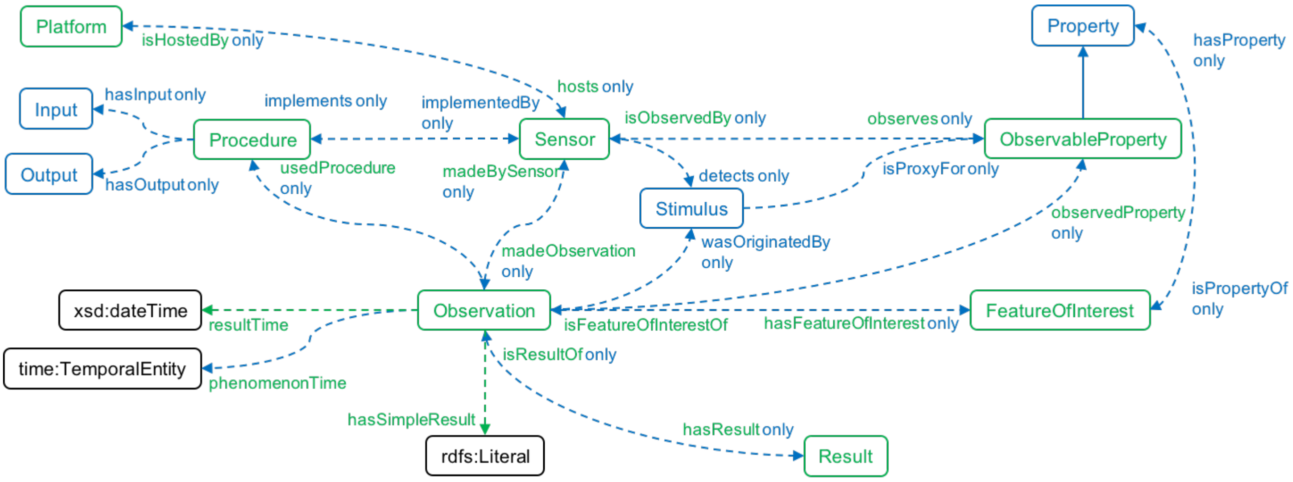


Figure G.1.2: Main classes and properties related to Observation (Copyright © 2017 W3C ® (MIT, ERCIM, Keio), All Rights Reserved)[12]

iot.schema.org extends *schema.org* to provide specific semantic vocabularies for WoT applications. The model of iot.schema.org is defined around the notion of *thing's capabilities* such as measuring pressure or turning on and off a device. Capabilities are related to *interaction patterns*, which describe an affordance to a capability. Interactions can be of type *Property*, *Event* or *Action*. Similar to WoT TD, iot.schema.org includes *data schemas* based on J-SON Schema to describe the data that interactions exchange. Things are represented by the notion of *Feature of Interest* and being related to a *physical location* of being part of a *system* or *equipment*. Figure G.1.3 shows the main classes and properties of the iot.schema.org specification.

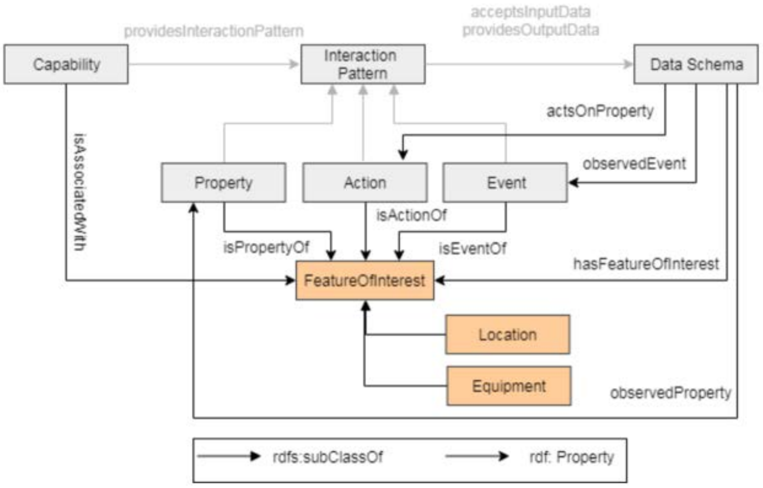


Figure G.1.3: Main classes and properties defined by iot.schema.org [22]

G.1.3 Examples

Notes from the Editors:

[12] https://www.w3.org/TR/vocab-ssn/#Observations-overview

- *hd12_room123*, representing an object of type *HeatDetector* installed in *room123*, has function *hd12_room123_hag_f* but it is not of type *FunctionalObject* because it is not part of a system
- The class *Alarm* is defined as a subclass of *Activity* and not a subclass of *Event* because an event has *zero-time extension*, following *ISO 15926-2*.

G.1.3.1 Example of a simple heat detector and alarm

Figure G.1.4 illustrates a modelling example of a heat alarm, *hd12_room123_alarm19,* of type *Alarm* generated by the heat detector *hd12_room123*, which is of type *InanimatePhysicalObject*. *hd12_room123* is installed (*residesIn*) in room *room123*. The heat detector *hd12_room123* has a function *hd12_room123_hag_f* of type *HeatAlarmGeneration*. The alarm *hd12_room123_alarm19* mimics the notion of *observation* in *SOSA/SSN* and it has a timestamp and a quantity datum (*hd12_room123_alarm19_qd1*) indicating the temperature that was recorded in a particular time. The class *TemperatureTooHigh* is a a subclass of *Temperature* and it represents a physical quantity, such as the related temperature measurements are larger than *60 degrees Celsius*.

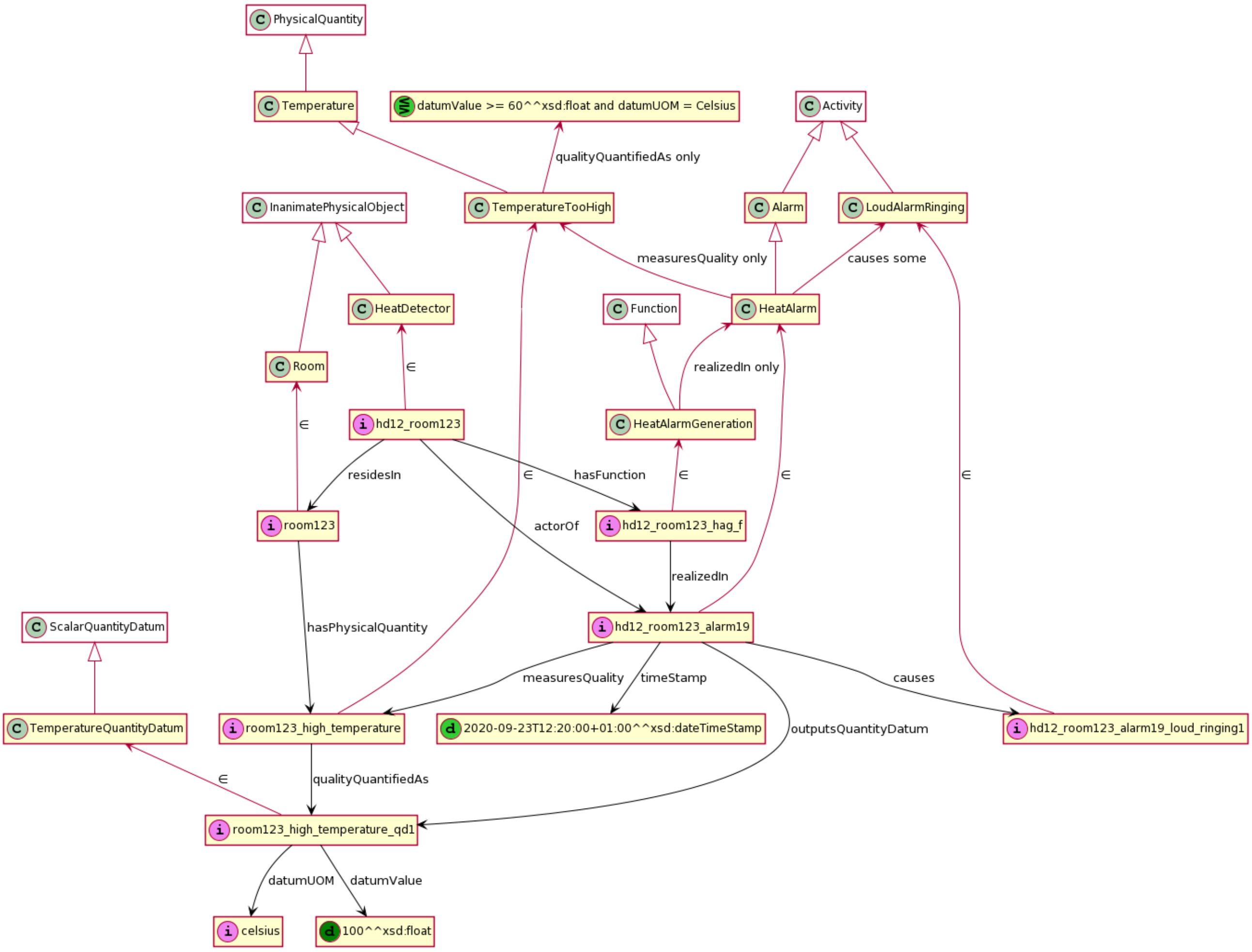


Figure G.1.4: Example of modelling a heat detector that produces heat alarms

G.1.3.2 Example of a heat detector and alarm suitable for more complex scenarios

For more complex scenarios with multiple alarms, devices, processes and types of events, it might be recommended to produce simpler models. This can be done by creating hierarchies of specialized *data properties* or defining *magic codes*, i.e. predefined values with specific meaning. The next two figures illustrate these modelling approaches.

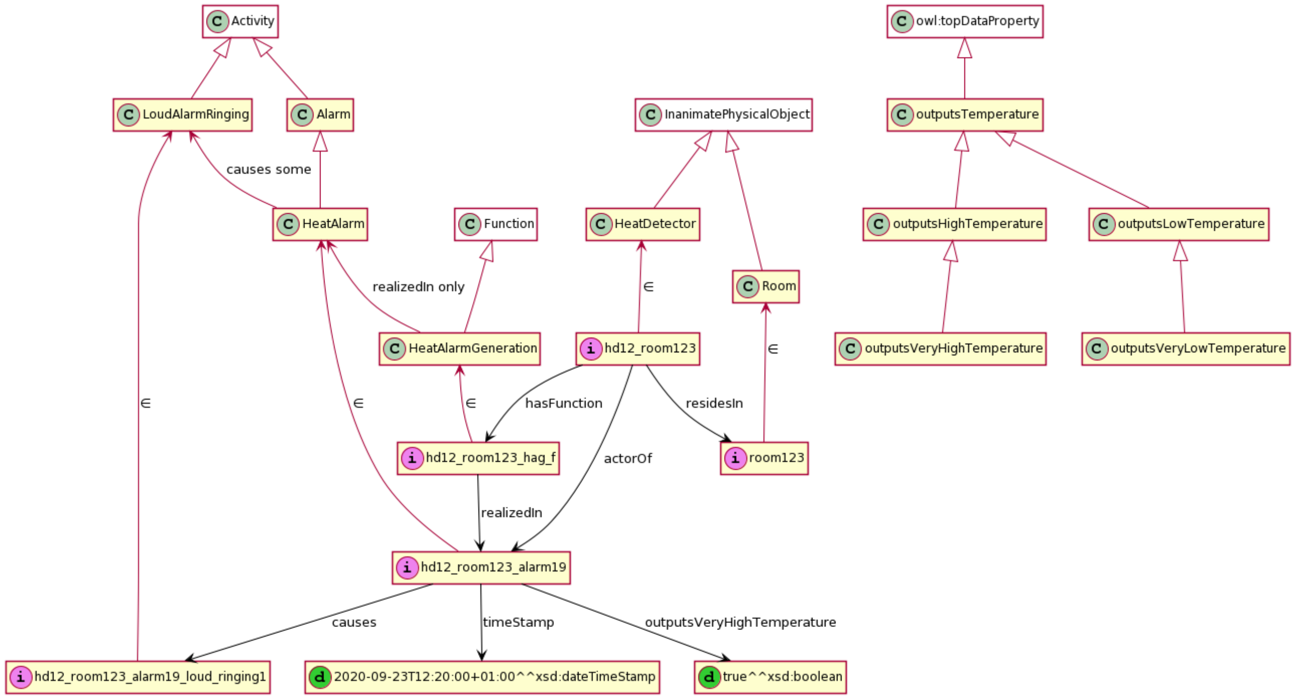


Figure G.1.5: Example of modelling alarms using data properties

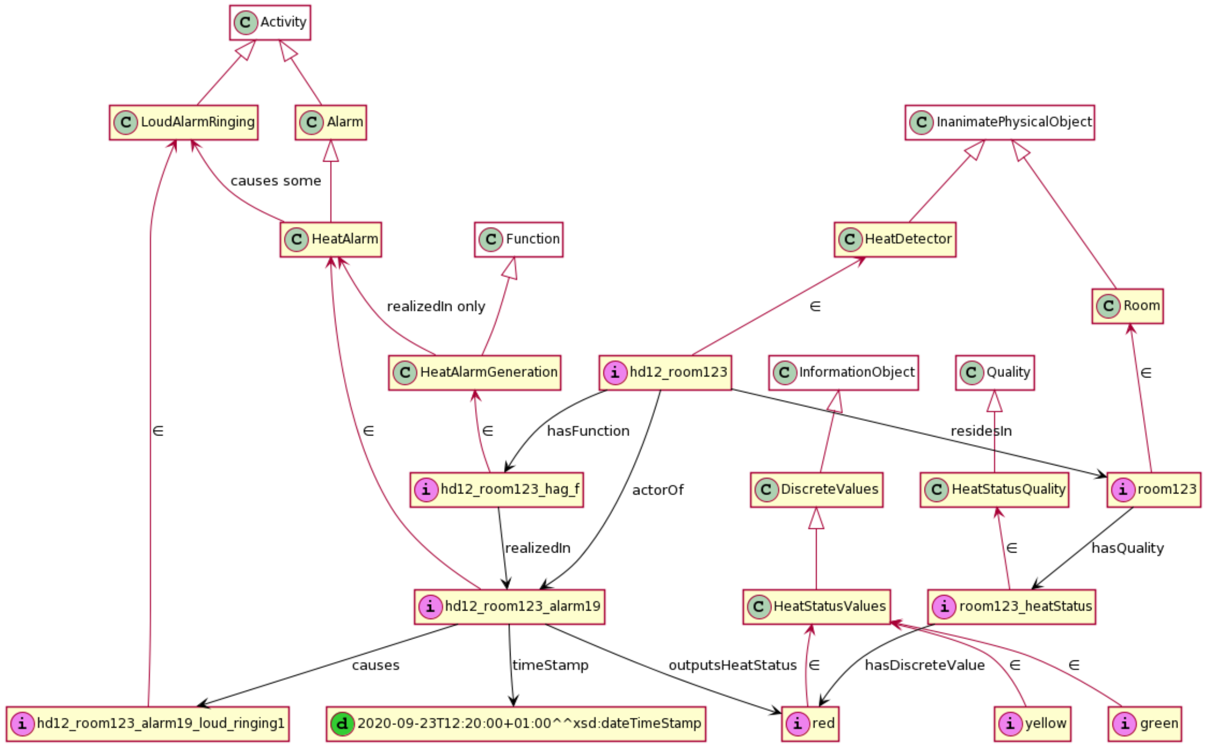


Figure G.1.6: Example of modelling alarms using *magic codes*

G.2 Physical-spatial

Models of building structures include building topology (decomposition of buildings into storeys, storeys into spaces/rooms, rooms into segments). Some of the objects are physical in their nature (building, room), while some are purely spatial (segment). Some spaces are purely spatial objects, while others may correspond to the actual rooms in the buildings. Various devices can be located in different parts of a building. In particular:

- The physical representation of a building structure with the elements which are part of the building topology (such as floors and rooms)
- The spatial location of a building structure, and the spatial location of a buildings' floors and rooms
- The location of a specific object (device) which is in a building
- The relationship between the physical and the spatial representation of the elements which constate the building topology

G.2.1 Status

Notes from the Editors: This use case requires further development.

- Make detailed references to BIM and IFC vocabulary sources
- Provide an OWL ontology that represents the content of the modelling diagram
- Provide OTTR templates, and accompanying tables of example data

G.2.2 Source data schemas and systems

Revit, Building Information Model (BIM)[13] and IFC[14].

G.2.3 Modelling Example

The following modelling assumptions have been made:

- A physical building, floor, and a room is represented as an instance of the class *PhysicalObject*. Building has floors as parts, and floors have rooms as parts.
- The location of a building is represented as an instance of the class *Site*, which is a subclass of *Location*. Additionally, the spatial representations of a buildings' floors and rooms are also represented as instances of the class *Site* (e.g., *RoomSite*, *BuildingSite*).
- When we want to specify the location of a specific device, we refer to the spatial representation of the location (e.g., *RoomSite*, *BuildingSite*). Building site has floor sites as sublocations, and floor sites have room sites as sublocations. Room sites have segments as sublocations.
- The relation between the physical representation of a building, a floor or a room and their corresponding location is represented with the object property *residesIn*.
- The object property *hasLocation* with domain *PhysicalObject* and range *Location* is used as an abbreviation for *residesIn* followed by *subLocationOf*; i.e.,

  $residesIn \circ subLocationOf \sqsubseteq hasLocation$

  E.g. we may state that a thermostat has a location room_site, which is short for saying that thermostat's site (in which it resides) is a sublocation of room_site.

[13] https://ec.europa.eu/jrc/en/publication/building-information-modelling-bim-standardization

[14] https://www.buildingsmart.org/standards/bsi-standards/industry-foundation-classes/

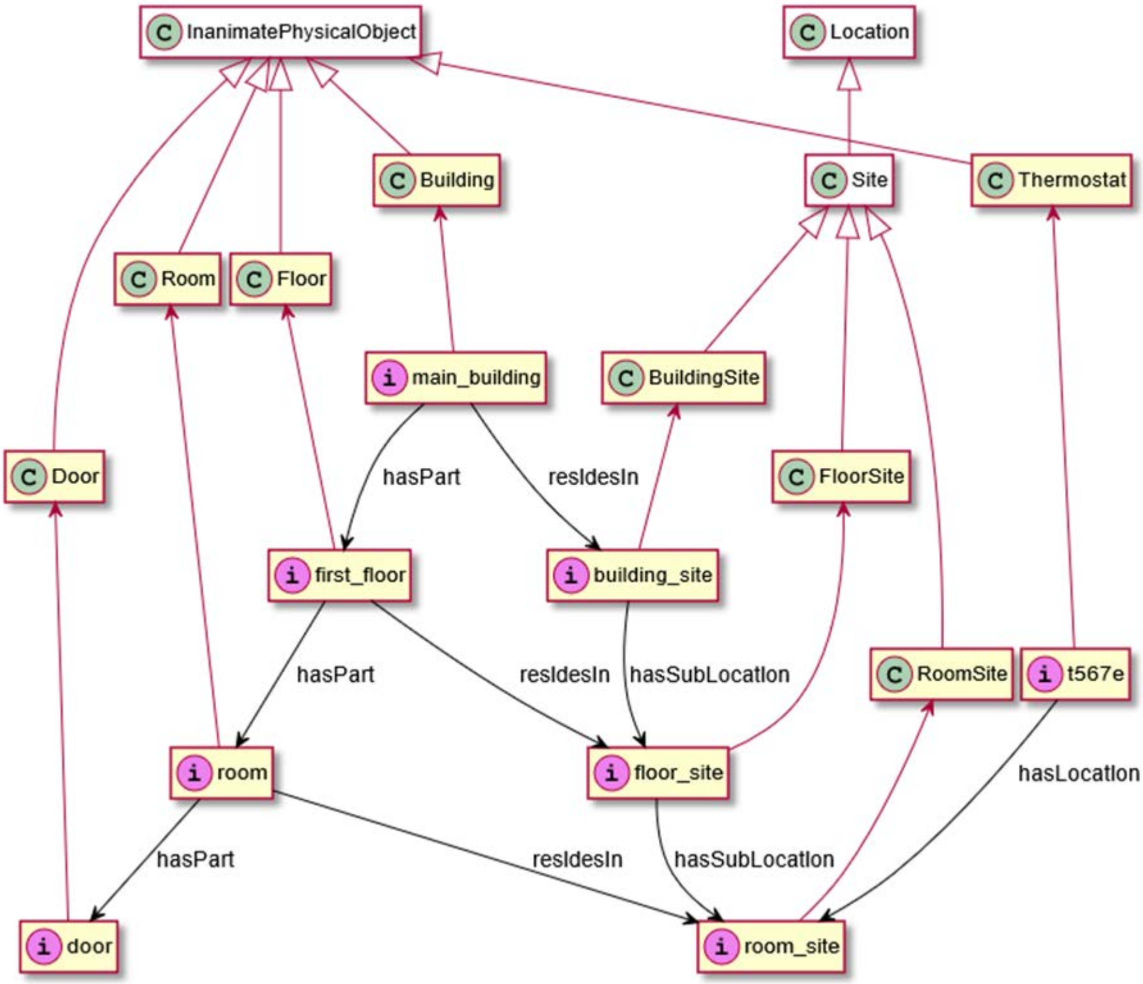


Figure G.2.1: Modelling example of a building structure

G.3 Piping and Instrumentation Diagram (P&ID)

A Piping and Instrumentation Drawing (P&ID) is a key design artefact for building process controls on a facility. The primary purpose of a P&ID is to present the topology of the plant from a process control perspective. Therefore, it includes the following objects and the exposes the links between them in a 2D diagram:

1) Equipment (showing the identifiers of the individual equipment/instrumentation items as well as an icon to show the class it belongs to).

2) Lines (including process flow lines, as well as instrumentation/control lines showing how the instrumentation forms an instrument/control loop)

3) Mereological composition links (e.g. if a motor is part of a compressor)

4) Topological connection links (e.g. if a line is connected to the compressor inlet)

5) Containment information (e.g. the flow stream within a process line)

6) Spatial information (e.g. how a particular vent is above the knockout drum or below)

For the purpose of this use case, we will only take into account only items 1 to 4 from the P&ID information types, however, the general methodology explained here can be extended to cover the remaining information as well.

G.3.1 Status

Notes from the Editors: This use case requires further development.

- A terse account of the use of OTTR shall be provided in a separate section, serving all use cases for which templates are provided. As it stands, the present use case can not be understood without reference to the OTTR documentation.
- Provide an OWL ontology, as a digital attachment, that represents the content of the modelling diagram
- Extend the P&ID content with streams, and containment relationships to process lines

G.3.2 Source data

Source data was secured from *https://15926.org/topics/mapping-pid/index.htm.* The input dataset was stored in a set of tables, capturing

1) Types: This contains the list of equipment types (e.g. Compressors, Process lines, instrumentation equipment),
2) Individuals with Types: this contains the list of individual P&ID objects (Equipment and Lines) along with the type instance link to the class,
3) Relationships: this includes mereological and topological links between the Individuals.

The source data was stored in comma-separated format, with the following schema.

1) Classes are stored in the file *types.csv*, with subtype—supertype pairs:

TYPES	SUPER_TYPES
COMPRESSOR SYSTEM	SYSTEM
CENTRIFUGAL COMPRESSOR	COMPRESSOR

2) Individuals are stored in the file *types-instances.csv*, containing instance—type pairs:

INSTANCES	TYPES
B14-KS-101	COMPRESSOR SYSTEM
B14-K-101	CENTRIFUGAL COMPRESSOR

3) Mereological relations are stored in *wholes-parts.csv*, containing whole—part pairs:

WHOLES	PARTS
B14-RW17801	B14-RW17801-S1
B14-RW17801	B14-RW17801-S2

4) Topological relations are stored in *connections.csv*, containing side_1,side_2 and connection type triples as indicated below:

SIDE_1S	SIDE_2S	CONNECTION_TYPES
B14-K-101	B14-KM-101	DRIVE-TO-DRIVEN CONNECTION
B14-E-101	B14-EM-101	DRIVE-TO-DRIVEN CONNECTION
B14-RW17801-S1	B14-V-101	FLANGED CONNECTION WITH SPECTACLE BLIND

G.3.3 Modelling

G.3.3.1 ISO 15926-14 Context

The following mapping can be used to relate P&ID contents to the core ISO 15926-14 concepts:

- Equipment classes are subclasses of *Equipment*, a subclass of *InanimatePhysicalObject*.
- Equipment/Instrument Items: individuals are modelled as instances of *FunctionalObject*, as the Tags represent designed equipment, where the manufactured equipment (*PhysicalObject*) will be installed at the *FunctionalObject*.
- Topological relationships are modelled using *connectedTo* for remote connections and *directlyConnectedTo* for proximal connections.
- Mereological relationships: parthood relations between the equipment (e.g. between a compressor and compressor motor) can be modelled with *partOf*.

G.3.3.2 Example

The following model diagram explains how a compressor (K-101) is *directlyConnectedTo* a line (B14-RZ1780) and *connectedTo* a vessel (V-101).

The ontology that results from translating the tables of data, via OTTR templates, will be provided as a digital attachment to this document. The following is an excerpt:

```
Individual: pnid:B14_K_101
    Types:
        pnid:CENTRIFUGAL_COMPRESSOR
    Facts:
     lis:connectedTo  pnid:B14_KM_101

Individual: pnid:B14_RZ17801
    Types:
        pnid:PIPING_SYSTEM

Individual: pnid:B14_RZ17801_S1
    Types:
        pnid:PIPING_NETWORK_SEGMENT
    Facts:
     lis:arrangedPartOf  pnid:B14_RZ17801,
     lis:connectedTo  pnid:B14_FE_101,
     lis:connectedTo  pnid:B14_RZ17801_S3,
     lis:connectedTo  pnid:B14_V_101

Individual: pnid:B14_RZ17801_S2
    Types:
        pnid:PIPING_NETWORK_SEGMENT
    Facts:
     lis:arrangedPartOf  pnid:B14_RZ17801,
     lis:connectedTo  pnid:B14_FE_101,
     lis:connectedTo  pnid:B14_K_101
```

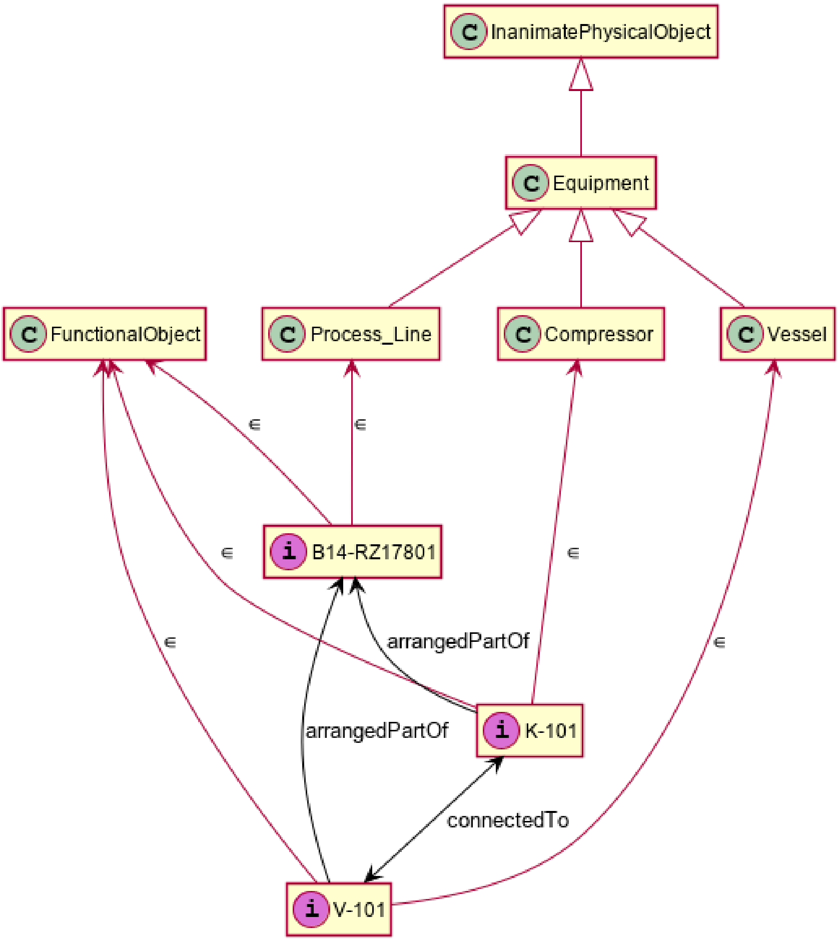


Figure B3.1: Example of modelling a compressor (K-101)

G.3.4 OTTR Templates

Further to the model patterns above, we describe OTTR templates that can be used to ingest the P&ID data into an ontology.

G.3.4.1 Prefixes

The following prefixes extend the list provided in section A.1.1.

OTTR template library prefixes:

```
@prefix :     <http://ns.ottr.xyz/0.4/> .
@prefix ax:  <http://tpl.ottr.xyz/owl/axiom/0.1/>.
@prefix dec:  <http://tpl.ottr.xyz/owl/declaration/0.1/>.
@prefix o-rdf: <http://tpl.ottr.xyz/rdf/0.1/>.
@prefix ont: <http://tpl.ottr.xyz/p/asset-maintenance/owl/ontology/0.1/> .
```

Prefixes for templates introduced in this use case:

@prefix ex-temp: <http://example.org/template/>.

```
@prefix pnid: <http://example.org/pnid/>.
```

G.3.4.2 Interface Templates

The following templates are in the *stOTTR* format. They provide a vocabulary to closely match the data of this particular use case. Each template will use one or more generic templates, assumed to be provided by a shared library of templates.

```
ex-temp:EquipmentTypesInterface [owl:Class ?sub, owl:Class ?super] :: {
    ex-temp:SubEquipmentType(?sub, ?super)
}.

ex-temp:EquipmentInstanceInterface [owl:NamedIndividual ?namedindividual, owl:Class ?class] :: {
    ex-temp:EquipmentInstance(?namedindividual, ?class)
}.

ex-temp:MereologyInterface [lis:PhysicalObject ?whole , lis:PhysicalObject ?part] :: {
    ex-temp:ArrangedWholePart(?whole, ?part)
}.

ex-temp:TopologyInterface [lis:PhysicalObject ?side1, lis:PhysicalObject ?side2] :: {
    ex-temp:Connection(?side1, ?side2)
}.
```

G.3.4.3 Modelling Templates

The following templates are in the *stOTTR* format. Their definitions use generic templates provided in the generic OTTR library.

```
ex-temp:SubEquipmentType[owl:Class ?subequipment, owl:Class ?super] :: {
    ax:SubClassOf(?subequipment, ?super)
}.

ex-temp:EquipmentInstance[owl:NamedIndividual ?namedindividual, owl:Class ?class] :: {
    o-rdf:Type(?namedindividual, ?class)
}.

ex-temp:ArrangedWholePart[lis:PhysicalObject ?whole, lis:PhysicalObject ?part] :: {
    :Triple(?whole, lis:arrangedPartOf, ?part)
}.

ex-temp:Connection[lis:PhysicalObject ?side1, lis:PhysicalObject ?side2] :: {
    :Triple(?side1, lis:connectedTo, ?side2)
}.
```

G.3.4.4 Mapping tables into templates

The above data is ingested from data files, via templates and into ontology, according to the following mappings, provided in *bOTTR* format. The embedded SQL queries are in the H2 database dialect, reading from comma-separated tables.

```
[] a :InstanceMap ; :source [ a :H2Source ] ; :template ex-temp:EquipmentTypesInterface ;
  :query """
SELECT CONCAT('pnid:',REGEXP_REPLACE(TYPES,'\\W', '_')),
CONCAT('pnid:',REGEXP_REPLACE(SUPER_TYPES,'\\W', '_'))
 FROM CSVREAD('./data/types.csv');
""";
  :argumentMaps( [ :type :IRI ] [ :type :IRI ]) .
```

```
[] a :InstanceMap ; :source [ a :H2Source ] ; :template ex-temp:EquipmentInstanceInterface ;
  :query """
SELECT CONCAT('pnid:',REGEXP_REPLACE(instances,'\\W', '_')),
CONCAT('pnid:',REGEXP_REPLACE(types,'\\W', '_'))
 FROM CSVREAD('./data/types-instances.csv');
""";
  :argumentMaps( [ :type :IRI ] [ :type :IRI ]) .
```

```
[] a :InstanceMap ; :source [ a :H2Source ] ; :template ex-temp:MereologyInterface ;
  :query """
SELECT CONCAT('pnid:',REGEXP_REPLACE(parts,'\\W', '_')),
CONCAT('pnid:',REGEXP_REPLACE(wholes,'\\W', '_'))
 FROM CSVREAD('./data/wholes-parts.csv');
""";
  :argumentMaps( [ :type :IRI ] [ :type :IRI ]) .
```

```
[] a :InstanceMap ; :source [ a :H2Source ] ; :template ex-temp:TopologyInterface ;
  :query """
SELECT CONCAT('pnid:',REGEXP_REPLACE(side_1s,'\\W', '_')),
CONCAT('pnid:',REGEXP_REPLACE(side_2s,'\\W', '_'))
 FROM CSVREAD('./data/connections.csv');
""";
   :argumentMaps( [ :type :IRI ] [ :type :IRI ]) .
```

Finally, the OTTR library template *Ontology* is used to declare the resulting ontology with a SQL query that returns static values.

```
:OntologyImports a :InstanceMap ; :source [a :H2Source ] ; :template ont:Ontology ;
  :query """
SELECT 'pnid:ontology', '(pnid:base, http://standards.iso.org/iso/15926/part14)';
""";
 :argumentMaps( [ :type :IRI ] [ :type (rdf:List :IRI) ]) .
```

G.4 Software

The goal of this use case is to represent software artefacts and their relationship to the hardware component in which the software is installed. We would like to model the following:

- A software program and its constituents, such as subroutines, statements, etc.
- Deployment and storage of a software program on hardware.
- Different configurations of a software program.
- Program runs.

G.4.1 Status

Notes from the Editors:

- The details of the type of entity that concretizes a software artefact needs review (object feature, versus a quality of the object)
- Details of the modelling using FunctionalObject needs review

G.4.2 Source data schemas and systems

Siemens Totally Integrated Automation Portal (TIA Portal)[15]

G.4.3 Modelling

G.4.3.1 ISO 15926-14 Context

- Software (SW) artefacts are primarily represented as information objects. This is in line with definitions from the BFO-compliant *Information Artifact Ontology (IAO)*:
    - An algorithm is a plan specification which describes the inputs and output of mathematical functions as well as workflow of execution for achieving a predefined objective. Algorithms are realized usually by means of implementation as computer programs for execution by automata.
    - A software artefact is a plan specification composed of a series of instructions that can be interpreted by or directly executed by a processing unit.
- Concrete installations of SW artefacts are represented as physical/functional objects and they are linked to the software artefacts (which are information objects) by means of the relation *concretizes*[16].
- Any SW core ontology needs to introduce a relation *installedOn* to directly link information-level SW artefacts to HW locations of their concretizations; this can be seen as a shortcut for the path *concretizedBy* o *partOf*; this implies that the installation-relation is not pure partonomy, as it has an implicit concretization-link in it for bridging between information and physical/functional objects.
- For information-level partonomies, any SW core ontology can specialize *hasPart*, here e.g. into *consistsOf*, a specialization of *hasPart* with *InformationObject* as *domain* and *range*.
- Concretizations of SW artefacts (installed SW artefacts) can be mereologically linked by means of more specific *hasPart* relations for physical/functional objects among each other.
- Hardware assets that host installations of SW artefacts are considered both *InanimatePhysicalObject*, for their physical character, and *FunctionalObject*, for their role in a system.
- Variables used in the software program are represented as instances of the class *Aspect*. More precisely, they are instances of a to-be-introduced Information Quality Entity (see Information Artifact Ontology), with different values of a variable (as in, over time, at configuration time, etc.) represented as *QuantityDatum* instances.
- A program run is an instance of the class *Activity*. An installed SW program is linked to its run via relationship *participatesIn*.

G.4.3.2 Example

Automation Software Configuration: Consider an automation engineering environment that allows you to program PLC code and to configure your automation setup, for example in terms of device configuration and network communication. The *Siemens TIA Portal* is a prominent example of such an environment. To give the automation engineer advice and assistance, the current automation solution is to be instantiated in a semantic model so that domain knowledge rules can be applied to it for providing validation and recommendation services to the user. Advice could be given as, 'you connected an input channel of a device to another input instead of an output', or 'you should include a self-testing functionality in the initialization of your code because the type of device used requires it'. The semantic model is populated from parts of the automation software code and configuration (e.g., PLC configuration

[15] https://new.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal.html

[16] The pattern of "concretization" can be applied to information objects other than software artefacts. The relation *concretizes* has been included into ISO15926 Part14, following the pattern provided by the IAO.

values) but possibly also from other additional sources that instantiate, e.g., the target plant layout with information about the physical devices controlled by the automation software.

Examples illustrating design decisions from the previous section are shown in Figure G.4.1 (software configuration example with a focus on the *installedOn* relationship), Figure G.4.2 (installed software asset as a physical concretization of a software artefact), and Figure G.4.3 (program run).

In Figure G.4.1, a partonomy of software artefacts (PLC Configuration -> PLC Program -> Procedure) is modeled by means of *consistsOf* relationship between the respective instances. The software artefacts are information objects. On the other hand, a PLC is a HardwareAsset (an *InanimatePhysicalObject*) and it is an assembled part (and also a functional part) of a station, which also also a HardwareAsset. By means of *installedOn* relationship, the PLC Program (software) is related to the PLC (hardware) it is installed on.

In Figure G.4.2, instances of installed software assets (installed PLC Program, Functional Block) are shown. The installed PLC Program is a part (both physical and functional) of the PLC hardware and it concertizes the PLC Program that is installed.

In Figure G.4.3, an installed PLC program (an *InanimatePhysicalObject*) is shown to be a participant in the program run (an *Activity*).

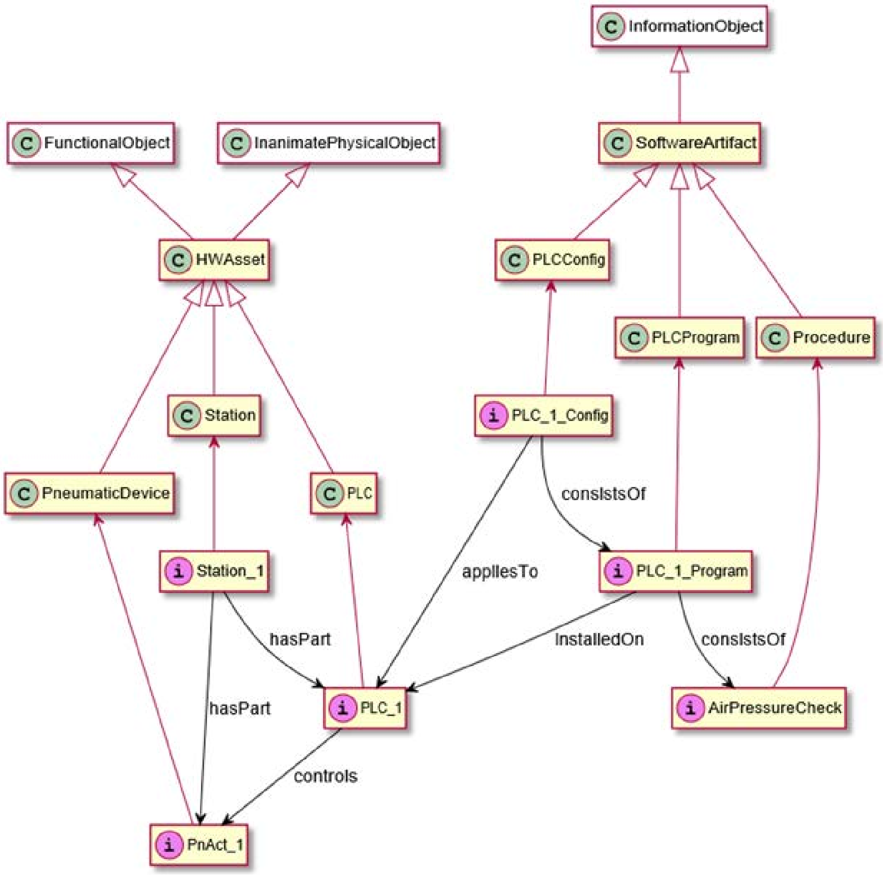


Figure G.4.1: Software configuration example

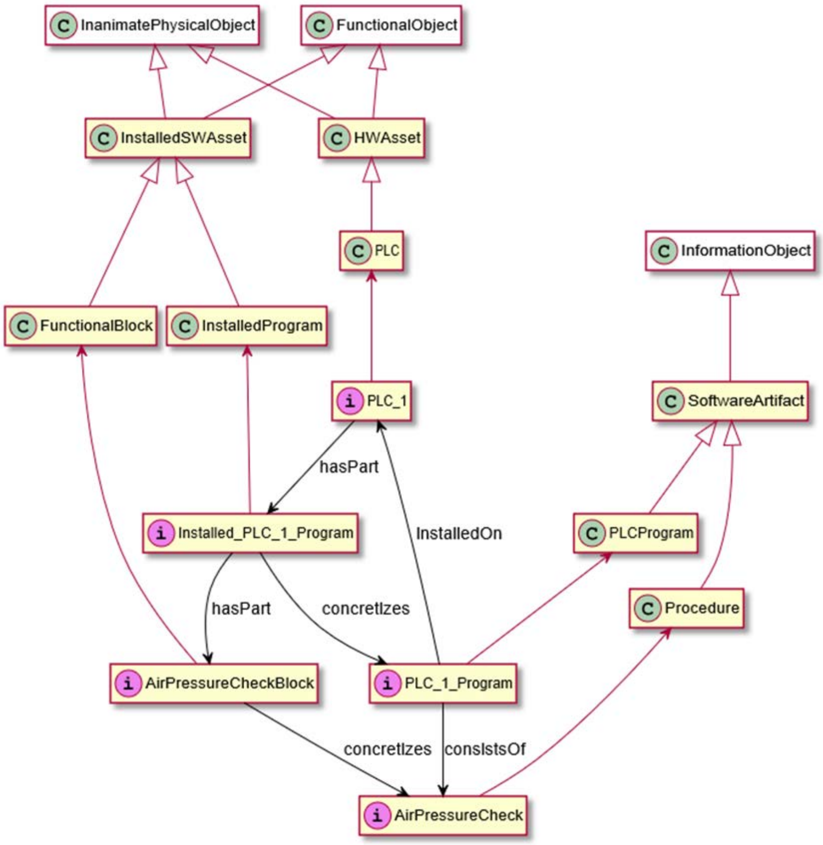


Figure G4.2: Installed Software Asset Example. It illustrates the concretizes relationship between a PLC program and its installation on a hardware.

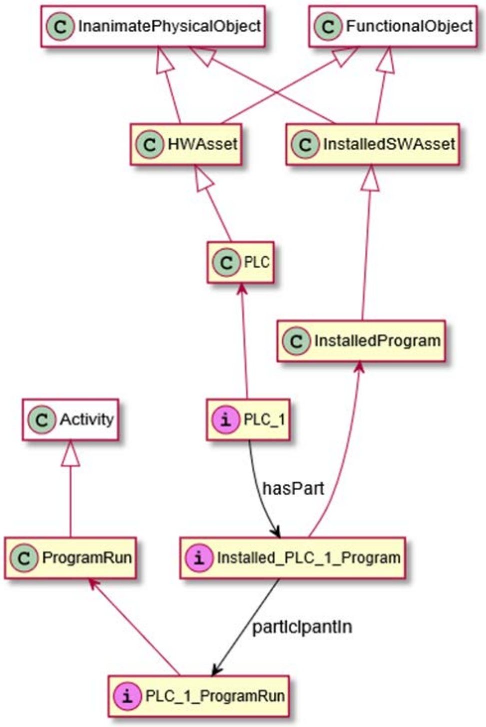


Figure G.4.3: Program run example: An installed PLC program (a PhysicalObject) is a participant in the program run (an Activity)

G.5 Product catalog

Product catalogues contain descriptions of hardware and software products, their classification, variants, components and properties. They may also include servicing offers, but this is of of scope of this use case. More specifically, the following needs to be considered when modelling product properties:

- Some properties are physical quantities, to be measured in specified units (e.g. Voltage is always measured in kV). They may have numeric ranges.
- Some properties have value ranges other than numeric (e.g. Boolean or enumerations).

Another important feature to be described is connection points (ports). For example, a network device may have ports that allows it to be wired with other devices within a system.

Product knowledge includes constraints (rules) on admissible product configurations, leading to valid product variants. These usually include dependencies between the component properties.

Product knowledge also includes information on product versions as well as their compatibility and successor relationships that is relevant for product maintenance.

Finally, it is important to relate maintained (installed) products to (abstract) products from the catalogue as well as to their spare parts list.

G.5.1 Status

Notes from the Editors:

- Add a discussion of how a piece of equipment may be validated against a prototype.
- Provide detailed, worked-out ontology examples

G.5.2 Source data schemas and systems

There are several standards for product classifications and descriptions, including *eCl@ss*[17] and *ETIM*.[18] Many companies use their own proprietary schemas (e.g., extensions of eCl@ss) and systems to capture product knowledge.

G.5.3 Modelling

G.5.3.1 ISO 15926-14 Context

While in terms of OWL, catalogue products and their classification are most naturally described via OWL classes, the detailed level of description required for particular products and their relationships to other products cannot be captured via OWL axioms. For example, a detailed Bills of Material (BoM) of a product that specifies that two sub-components of a product are physically connected to each other requires a non-tree-shaped model that cannot be enforced in OWL without resorting to individuals. For this reason, the possibility for representation on both class and individual level is necessary.

When it comes to the representation of products on the individual level, these are considered to be *prototypical* products and are treated as physical objects (hardware) or information objects (software). If a configured product requires software to be installed (e.g., a laptop with Windows 10), then the

[17] https://www.eclasscontent.com/

[18] https://www.etim-international.com/

concrete Windows 10 installation on the configured (prototypical) laptop is also considered a physical object (a physical *Feature*) – see the Software use case for more details.

G.5.3.2 Examples

Figure G.5.1 shows the class and subclass relations of a product. A product can have many different configurations and therefore many different prototypes. Because the number of such different prototypes would be exponential to the number of different possible configurations, it is often the case that not all configurations are explicitly shown with separate classes. In the diagram below, we can see an example of an instance of a prototype for a switchgear which is of a type *8DSHStandard*, it has a single bus bar and it is a wall standing switchgear. In the case when not all different configurations are represented with separate classes (such as *SingleBusBar_Switchgear* or *WallStanding_Switchgear*), the configuration of the instance prototype should be inferred from its properties. In addition to modelling the prototype to be an instance of the most specific product class, it is also represented as an instance of a specially designated class *Prototype*. We can observe this in the example, where the instance *prototype_8DSH_singlebus_wall_standing_switchgear* is an instance of the specific product class *8DSH_Singlebus_WallStanding_Switchgear* and it is also an instance of the class *Prototype*.

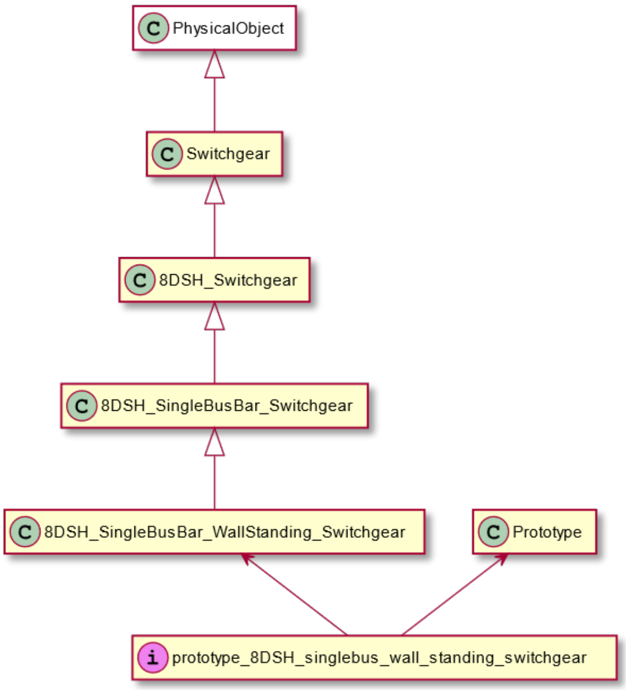

C PhysicalObject
C Switchgear
C 8DSH_Switchgear
C 8DSH_SingleBusBar_Switchgear
C 8DSH_SingleBusBar_WallStanding_Switchgear
C Prototype
i prototype_8DSH_singlebus_wall_standing_switchgear

Figure G.5.1: Example of product classification and a prototype instance

Figure G.5.2 illustrates how product knowledge such as product versions and successor relationships are modelled between prototype instances of the same specific product class. Here we observe that the two prototype instances are both instances of the same specific product class (*8DSH_SingleBusBar_WallStanding_Switchgear*) and of the class *Prototype*. The difference is in their version number, which is represented as a data property. The two instances are related via the relationship *hasSuccessor*.

Figure G.5.3 shows how to relate a concrete maintained product to its corresponding prototype from the catalogue. The catalogue prototype has a unique manufacturer's article number, while the concrete product has a unique serial number. The concrete product is related to the corresponding prototype via relationship *hasDesign*. For a functional location at which the concrete product is installed, permissible design alternatives are modelled by making the functional location an instance of the complex class *hasDesign only* {*prototype*$_1$, *prototype*$_2$, ..., *prototype*$_n$}.

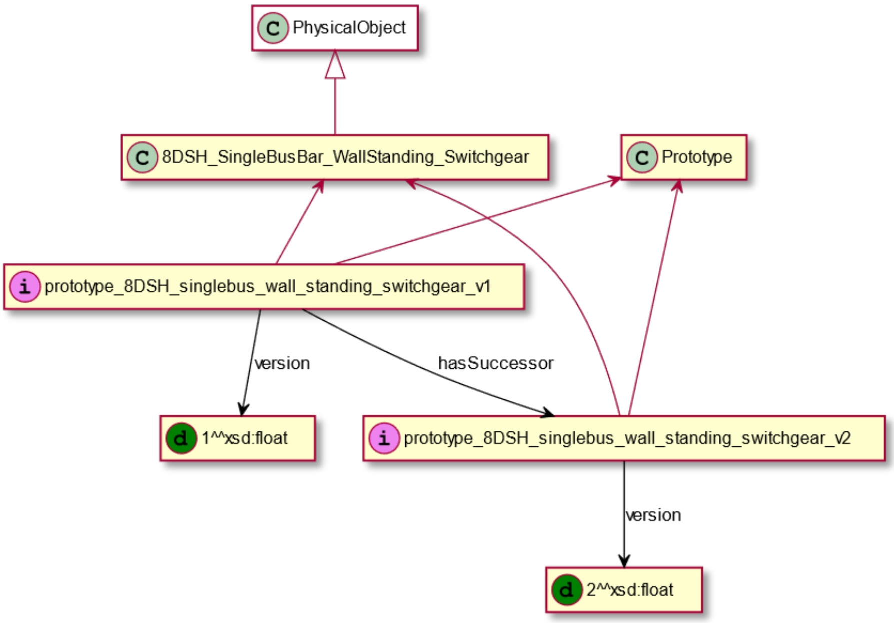

PhysicalObject
8DSH_SingleBusBar_WallStanding_Switchgear
Prototype
prototype_8DSH_singlebus_wall_standing_switchgear_v1
version
hasSuccessor
1^^xsd:float
prototype_8DSH_singlebus_wall_standing_switchgear_v2
version
2^^xsd:float

Figure G.5.2: Product versions and product successor

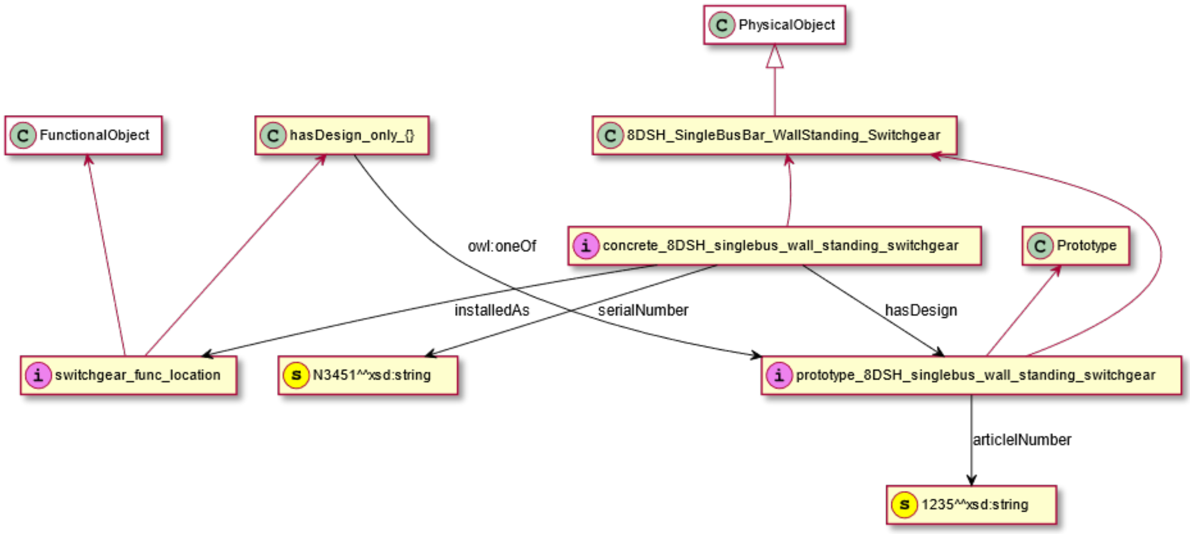
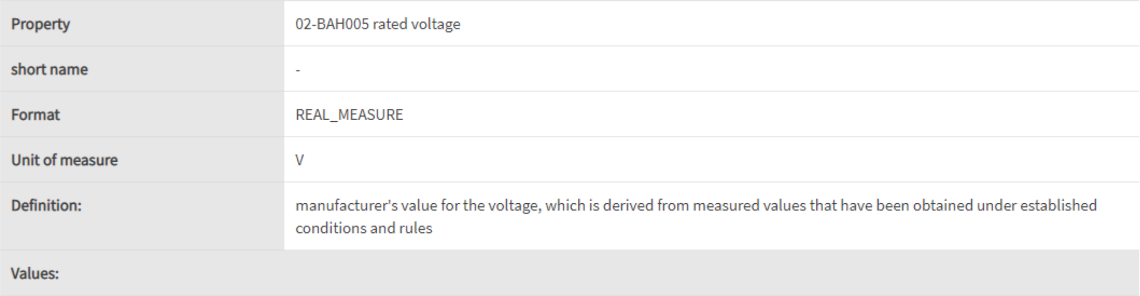


Figure G.5.3: Relationships among a concrete product, prototype product, and a functional location

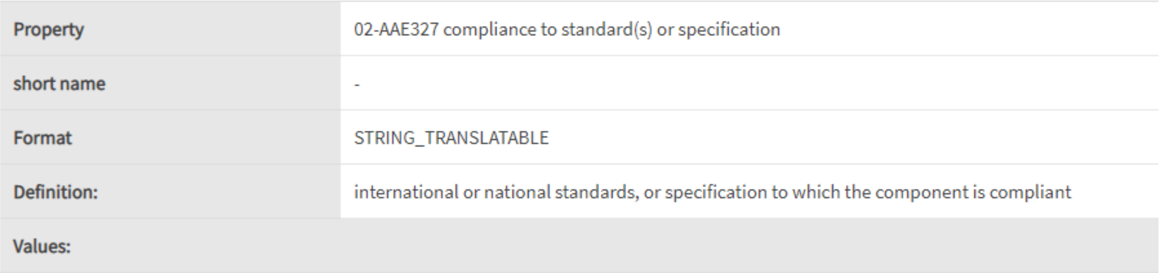
In the last two examples we show how the properties, as well as the configuration of a product are expressed on instance level. In Figure G.5.6 we observe how the different properties of a product are represented. Here, the properties of the switchgear instance are instances of *RatedVoltage* and *Standard*. Such product properties often come from standards such as *ecl@ss*; see Figure G.5.4 and Figure G.5.5 for respective *ecl@ss* definitions of our example properties. In terms of Part 14, *RatedVoltage* is a subclass of *PhysicalQuantity*, while *Standard* is a subclass of *Quality*.

Please note that a proper way of modelling standards would require to introduce classes such *IECConformantProduct*; however, real-life use cases (including representation of ecl@ss properties) requires its representation as a property of the product. Also note that in many use cases, a more compact representation of properties as OWL object or data properties is suitable.



Property	02-BAH005 rated voltage
short name	-
Format	REAL_MEASURE
Unit of measure	V
Definition:	manufacturer's value for the voltage, which is derived from measured values that have been obtained under established conditions and rules
Values:	

Figure G.5.4: *Rated Voltage* property definition in ecl@ss



Property	02-AAE327 compliance to standard(s) or specification
short name	-
Format	STRING_TRANSLATABLE
Definition:	international or national standards, or specification to which the component is compliant
Values:	

Figure G.5.5: *Compliant Standard* property definition in ecl@ss

The fifth example (Figure G.5.7) shows how prototype configuration rules are defined.

Simple example interlocking (configuration) rule:

```
qualityQuantifiedAs(switchgear_standard, GOST) ->

qualityQuantifiedAs(switchgear_ratedVoltage, 12-kV) or
qualityQuantifiedAs(switchgear_ratedVoltage, 24-kV)
```

Such rules are used e.g. in product configuration user interfaces. Please note that these rules tend to be very complex, also interlocking properties of switchgear assembled components (*hasAssembledPart* can be used to relate a prototype switchgear to its components) and their interfaces (connection ports).

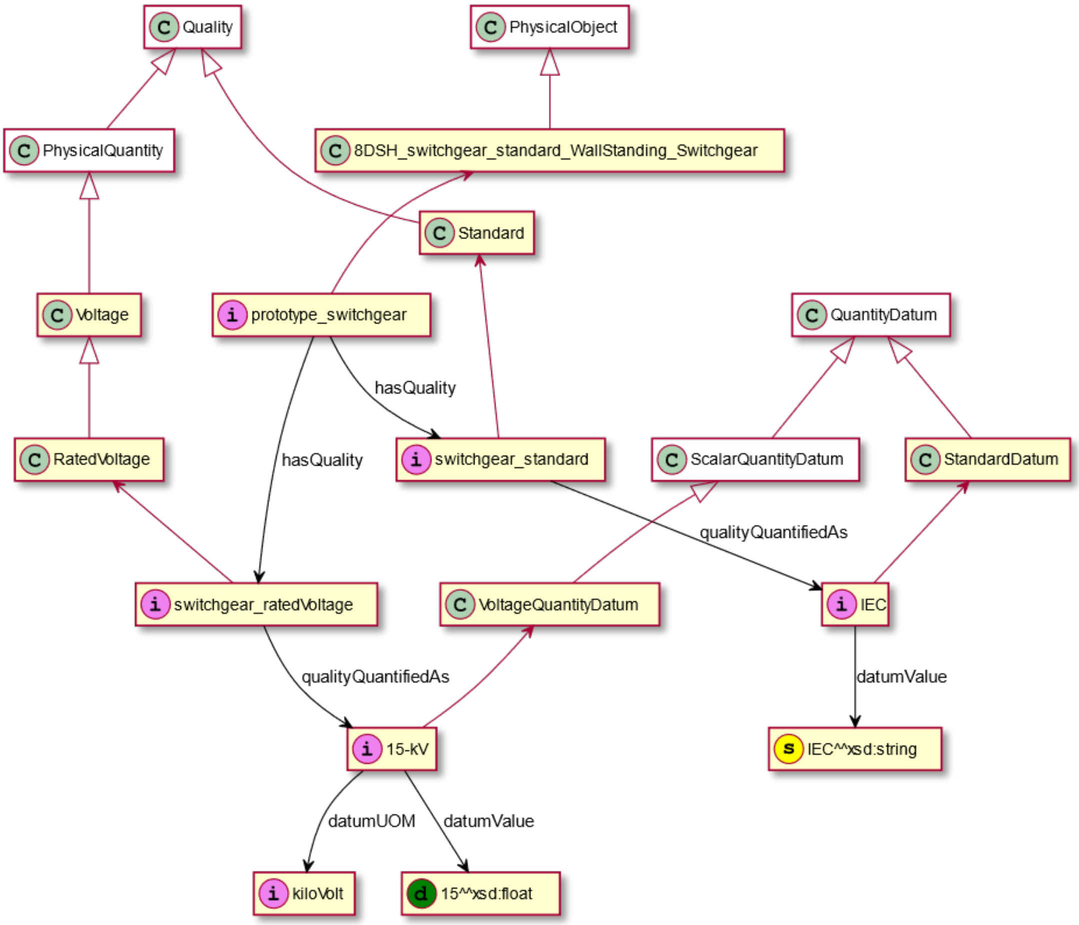


Figure G.5.6: Example of switchgear's properties

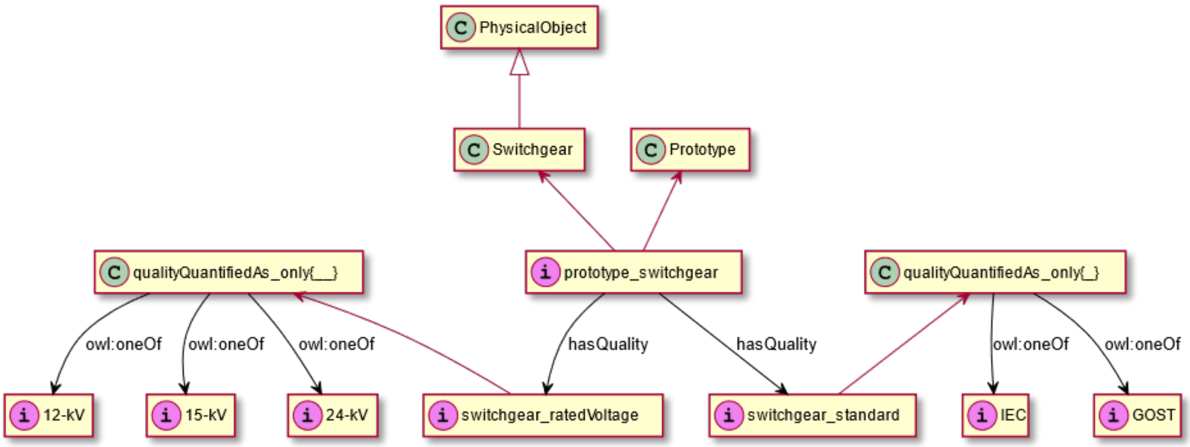


Figure G.5.7: Product configuration example

G.6 Aspect-based reference designation system

ISO/IEC 81346 (RDS) is a standard from systems engineering for codification of system breakdown structures and reference designation systems. In particular, it provides a means of structuring system information into *aspects*, providing views onto the same asset from distinct perspectives. This allows differentiating between information relating to an assets function, its location, and its components.

RDS focusses on assets as *specified*. It provides a top-down view by structuring information in the design stage of an asset's life cycle. These specifications describe the properties that the physical products are intended to have. As such it is a desirable candidate for integrating with ISO 15926-14 ontologies for checking whether an asset satisfies its design specifications.

The goal of this use case is to demonstrate how a design-level view such as that which ISO/IEC 81346 provides can be integrated with ISO 15926-14 ontologies. It allows for both a top-down view on the asset (determining specifications based on complex part-of structures, their relationships, as well as which physical asset must comply with these) as well as a bottom-up view from the data (grouping related specifications on a physical asset and associating these with a tag).

G.6.1 Status

Notes from the Editors:

- Review modelling details of the example
- Provide a self-contained ontology example
- Provide templates for modelling patterns

G.6.2 Modelling

The model is naturally divided into three models corresponding to distinct levels of abstraction, as depicted in Figure G.6.1.

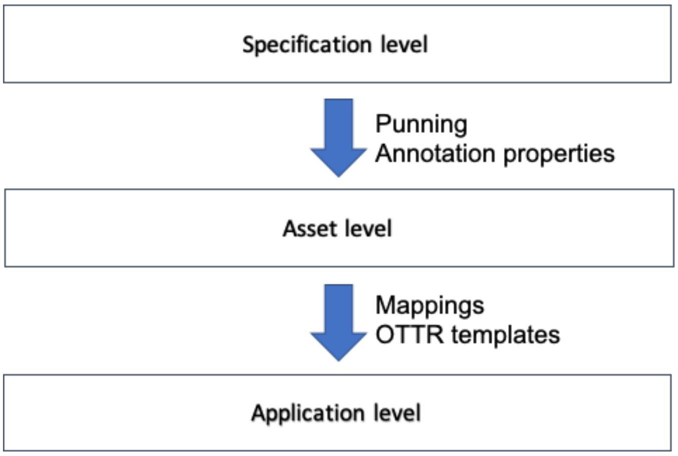


Figure G.6.1: Levels of Abstraction

This division is a natural interpretation of the RDS. Figure C.5 in the ISO/IEC 81346-1 document describes how the various aspects can be represented either as a group of objects (one for each aspect) or merged into a single object. In this use case, the left-hand side of the figure (where each aspect is a distinct object) is represented at the Specification level, whereas the right-hand side (where the objects are merged into one) is modelled at the Asset level.

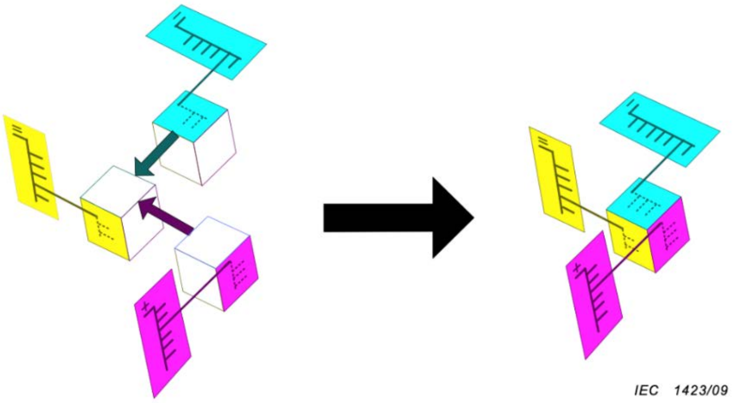


Figure C.5 – The three objects are merged into one

G.6.2.1 Specification level

The *Specification level* models specifications made during the design of an asset. It organizes these specifications into various aspects or part-of breakdown structures according to ISO/IEC 81346. At this level, design specifications are represented in a reified form, that is, as OWL individuals. In this manner design specifications related to different aspects can be referenced, as they each have a unique IRI, and organized into functional, location, and product part-of structures. This corresponds to the left-hand side of Figure C.5, where each aspect is its own object. Integrating specifications across aspects is handled at the Asset level.

G.6.2.2 Asset level

This level models functional objects and how they relate to the specifications represented in the Specification level. Here, the specifications are interpreted as axioms, which are used to verify whether a given functional object satisfies the restrictions posed to it by its design specifications. These objects correspond to the *cube* view on the right-hand side of Figure C.5, where the different aspects are viewed as a single object.

G.6.2.3 Application level

This level corresponds to the data and properties related to existing physical assets. This data is typically stored in a variety of structured formats, such as data management systems and spreadsheets.

G.6.2.4 Linking the Levels

Linking these levels is not trivial. The individuals in the topmost model are reified specifications. These provide a restricted view (per aspect) on the design specifications made to a functional object. This functional object (which we refer to as the system individual) is introduced at the Asset. However, in the context of the Asset level, the specifications serve as restrictions on the system individual's properties. Thus, the reified specification individuals from the top layer are interpreted as axioms in the Asset level. The connection between the axioms in the Asset level and the reified specifications in the Specification level is achieved via annotation properties and punning.

The actual data and property values lie at the Application level. To populate the Asset level with this data, it is gathered and translated into *OWL assertions* via mappings and *OTTR templates* [23].

G.6.2.5 Relation to ISO/IEC 81346 Tags

One of the great benefits to the aspect structure within ISO/IEC 81346 is the manner in which it facilitates a formalized method of generating tags. Part 2 of the standard provides a syntax for identifying system individuals through the aspects' breakdown structures. Tags describe a way of traversing the breakdown structures and are a concatenation of strings of the form:

[PREFIX][IDENTIFIER]

where PREFIX can be "-", "+", or "=", representing the product, location, and function aspects respectively. Thus, *=B1-A1* is a tag for asset with identifier A1 which is a subproduct of the asset with functional identifier B1.

The modelling described in this use case is highly compatible with these tags. On the one hand, given a set of tags the respective breakdown structures can be generated at the Specification level. On the other hand, tags for systems at the Asset level can be generated using the breakdown structures at the Specification level.

G.6.3 Example

To illustrate the modelling, we consider the functional object *pump1* of type *Electric Motor*. For this particular functional object, the following specifications were made during the design phase:

- *pump1* shall reside in a non-explosive atmosphere
- *pump1* shall have a capacity greater than 100 m^3/h
- *pump1* shall weigh less than 200 kg

All design specifications within the project are modelled at the Specification level. These are illustrated in Figure G.6.2 (Function aspect), Figure G.6.3 (Location aspect) and Figure G.6.4 (Product aspect).

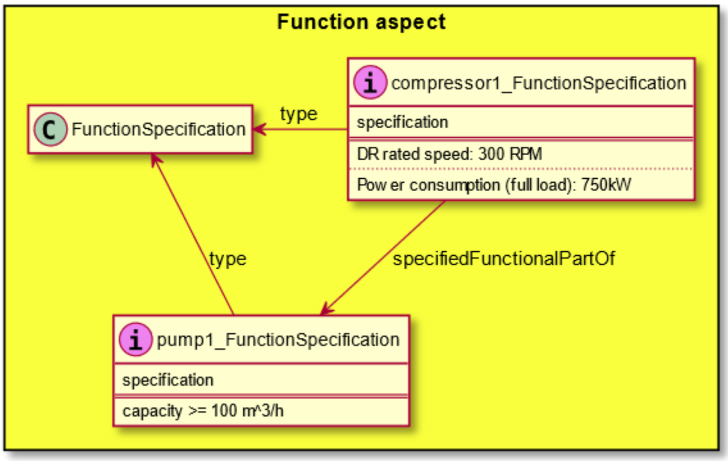

Function aspect
compressor1_FunctionSpecification
specification
DR rated speed: 300 RPM
Pow er consumption (full load): 750kW
FunctionSpecification
type
type
specifiedFunctionalPartOf
pump1_FunctionSpecification
specification
capacity >= 100 m^3/h

Figure G.6.2: Function aspect of functional object *pump1*

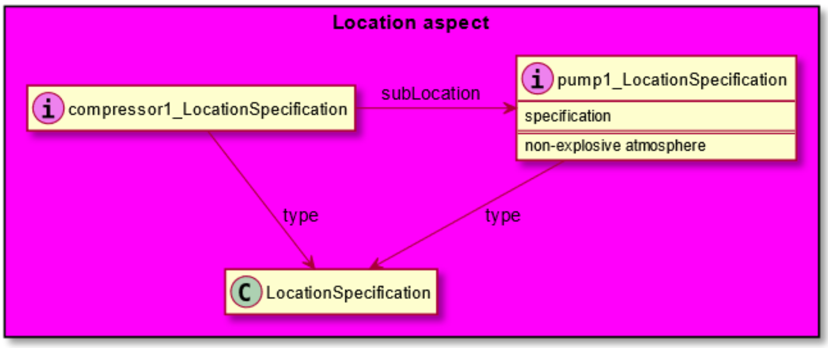


Figure G.6.3: Location aspect of functional object *pump1*

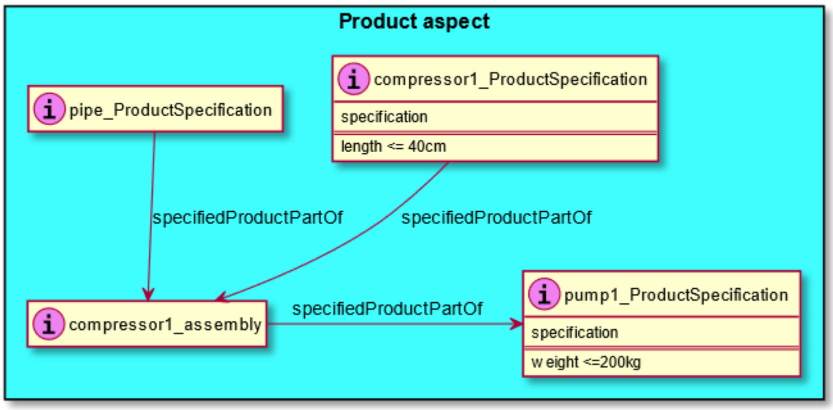


Figure G.6.4: Product aspect of functional object *pump1*

The specification individuals for each aspect are distinct objects with distinct IRI's. The integration of the three aspects (cf. Figure C.5 of the ISO/IEC 81346-1 standard) happens at the Asset level. Here a *system* individual is introduced as a FunctionalObject, which relates to the specification individuals at the Specification level. This system individual represents the *cube* view provided by ISO/IEC 81346-1 and corresponds to the right-hand side of Figure C.5. The result is depicted in Figure G.6.5.

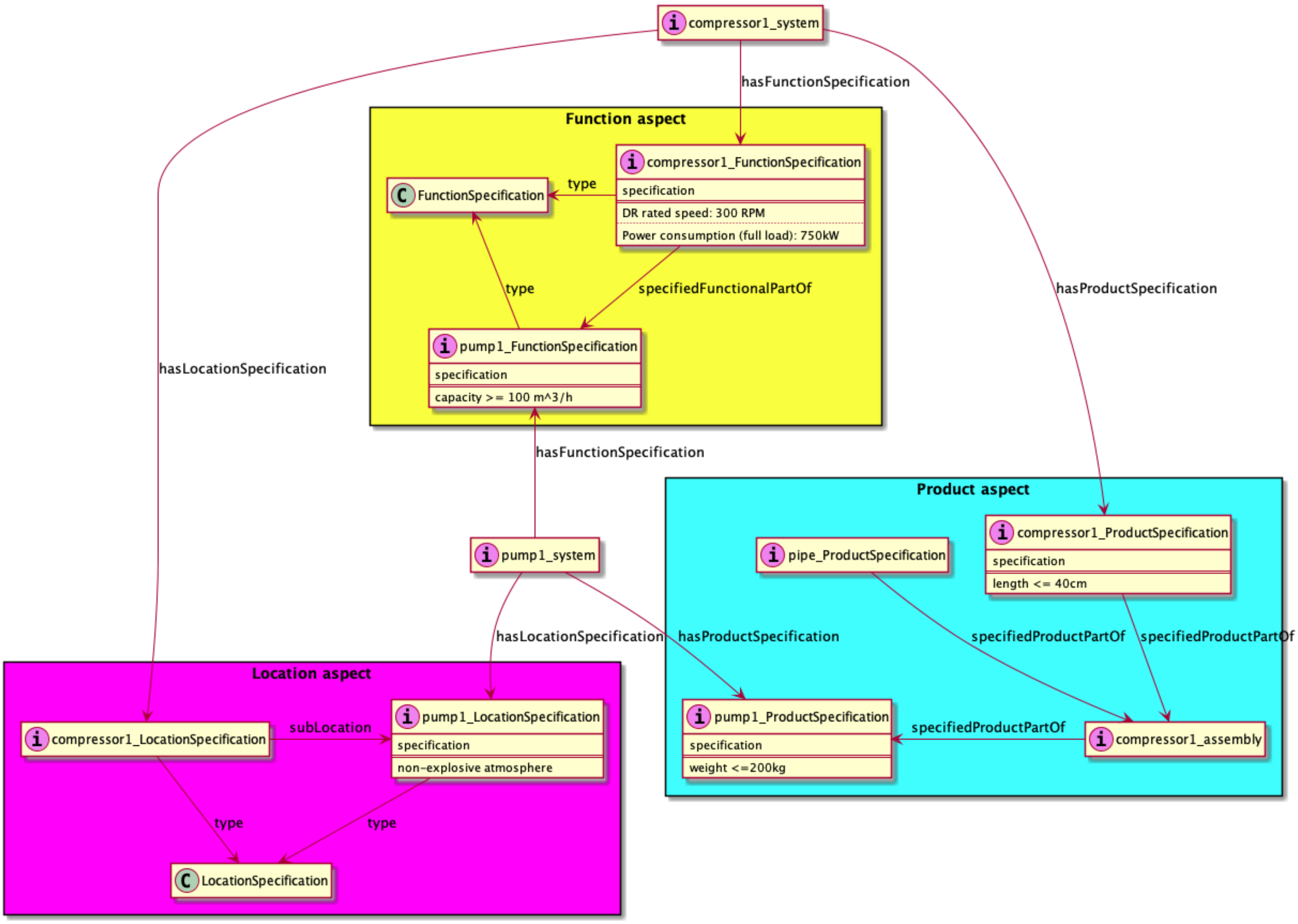


Figure G.6.5: Integration of three aspects using the system *pump1_system*

The properties of an asset are assigned to the system individual. In order to verify that these properties satisfy the restrictions posed by the specifications, the individuals at the Specification level are translated to restriction axioms at the Asset level:

```
Individual: :pump1
    Types:
      lis:residesIn only :Non-explosive_atmosphere,
      :hasCapacity only xsd:integer[>= 100],
      :hasWeight only xsd:integer[<= 200]
```

These restrictions are linked to the specification individuals on the Specification level. This is achieved by annotating the restriction axioms as follows:

```
Individual: :pump1

    Types:

 Annotations: partOfLocationSpecification <http://example.org/pump1_LocationSpecification>
        lis:residesIn only Non-explosive_atmosphere,

 Annotations: partOfLocationSpecification <http://example.org/pump1_FunctionSpecification>
        hasCapacity only xsd:integer[>= 100],

 Annotations: partOfLocationSpecification <http://example.org/pump1_ProductSpecification>
        hasWeight only xsd:integer[<= 200]
```

In this manner, the restrictions can be grouped according to their aspect into, respectively, the OWL individuals *pump1_LocationSpecification*, *pump1_FunctionSpecification*, and *pump1_Product*Specification.

G.7 Requirements

Requirements are posed to assets, such as oil platforms, process plants, compressors or pumps, at each stage in their life cycles. This use case discusses how to represent requirements in the context of asset life-cycle information, following the methods described in the *Reified Requirements Ontology* [24]. Additionally, multiple approaches to how *OWL reasoning* can be used for requirement verification are discussed.

G.7.1 Status

Notes from the Editors:

- The Terminology section should be made clearer, to emphasize the form Scope(x) & Condition(x) -> Demand(x) – currently, it's not obvious that the demand is also a class
- Use of description logic symbols can probably be avoided, in favour of natural language analogues
- Providing templates and ontology is probably best left to the Requirements Ontology; refer to that resource if they become available

G.7.2 Terminology

In this document, we adopt the following terminology used in the *Reified Requirements Ontology* [24]:

- *Requirement*: a proposition with the modality of *it is required that …*:
  - a requirement contains a *scope*, and *demand*, and may contain a *condition*;
  - this is to be read intuitively as: *all elements of the scope that satisfy the condition must meet the demand*.
- *Scope*: a class of subjects to which a requirement should *apply*.
- *Condition*: a condition the scope of a requirement must *satisfy*.
- *Demand*: the expected *outcome* or documentation that all elements of Scope that satisfy the Condition are obligated to have.

G.7.3 Source data schemas and systems

Industrial requirements are available both in an *unstructured manner*, e.g., in PDF's and other natural language documents, as well as in *structured formats*, such as Excel spreadsheets or digital requirements catalogues [25].

G.7.4 Modelling

As opposed to the *declarative statements* in an OWL ontology, i.e., statements that something is true, requirements are imperative statements, i.e., statements that something *should hold*. Thus the logical framework underlying OWL is not suitable to fully capture the semantics of requirements [3]. For this reason, we divide the modelling of requirements into two tasks: (1) the *representation of requirements*, e.g., for the purposes of documentation; and (2) *verification of requirements*. The former follows the modelling practices outlined in the *Reified Requirements Ontology* [21] and is described in Section G.7.4.1. In Section G.7.4.2 we propose various *encodings of requirements* as OWL axioms in order to use reasoning to verify whether (certain types of) requirements are met by specific assets. Note that the two approaches discussed in Sections G.7.4.2.1 and G.7.4.2.2 have different advantages and, as such, complement each other well in practical use.

The connection between reified requirements and the OWL axioms used for their verification is achieved via OWL *punning*,[19] i.e., interpreting IRI's as both an OWL individual and an OWL class. Examples of this approach are provided below.

G.7.4.1 Reified Requirements

For the purposes of documenting requirements, we introduce referrable objects called *reified requirements* as OWL individuals to the model. Each reified requirement has an *SCD Clause* associated to it via the *has SCD Clause* object property. The SCD Clause bears information regarding which class of individuals a requirement applies to (defined by the scope and condition of the SCD clause) as well as what one demands of such individuals. It is important to note that the SCD Clause does not indicate the normative strength of the proposition. This is rather indicated by the requirement's placement in the *Proposition > Advice > Recommendation/Requirement* class hierarchy provided by the *Reified Requirements Ontology* [1].

It is conceivable that multiple organizations pose requirements with the same SCD clause; that is, effectively, that they require the same demand of the same assets. To differentiate requirements posed by distinct entities, we use the *posited by* object property from the *Reified Requirements Ontology*, linking the requirement to its source.

Modelling example: Let us consider the requirement *Every RDS Electrical System in an Explosive Gas Zone should have an EX Certificate*, which is posited by the *Organization org_1*. This is modelled as depicted in Figure G.7.1.

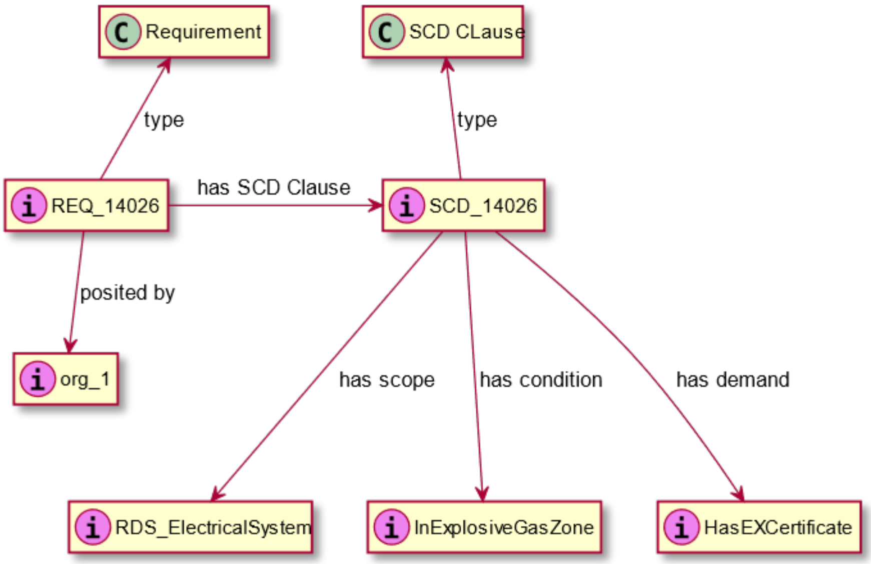


Figure G.7.1: Modelling example of a reified requirement

G.7.4.2 Verifying Requirements

OWL 2 semantics are not capable of capturing the *defeasible nature* of requirements [26]. Nevertheless, one can construct OWL axioms from the reified requirements that *approximate* the intended semantics well enough to provide valuable insight. In the following we present multiple such approaches.

The connection between the axioms described in this section and the reified requirements from the previous section is achieved via *punning*, which is a feature in OWL that allows using the same IRI to

[19] https://www.w3.org/TR/owl2-new-features/#F12:_Punning

denote classes, properties, and individuals. In the case of axiomatizing requirements, the *Scope*, *Condition*, and *Demand* individuals of a requirements' SCD Clause are punned as classes in OWL axioms.

We discuss the various axiomatizations for requirement verification based on our previous example requirement: *Every RDS Electrical System in an Explosive Gas Zone has an EX Certificate*. Furthermore, we illustrate the differences of the approaches by discussing how the semantics of the axioms act in the following scenarios:

- *Scenario 1*: *ES1* is an *RDS Electrical System* in an *Explosive Gas Zone* and we do not know whether it has an *EX Certificate*.
- *Scenario 2*: *ES2* is an *RDS Electrical System* in an *Explosive Gas Zone* that does not have an *EX Certificate*.
- *Scenario 3*: *ES3* is an *RDS Electrical System* in an *Explosive Gas Zone* that has an *EX Certificate*.

G.7.4.2.1Naïve Axiomatization

The Scope, Condition, and Demand of a requirement are *punned* and *interpreted* as OWL classes. The requirement is then represented by axioms of the following schema

$$Scope \sqcap Condition \sqsubseteq Demand$$

Thus, our example requirement would be represented by the following axiom:

$$RDS\_ElectricalSystem\ \sqcap InExplosiveGasZone \sqsubseteq HasEXCertificate$$

- *Scenario 1*: *ES1* is a member of *RDS_ElectricalSystem* and *InExplosiveGasZone*. Thus, it is inferred to be in the class *HasEXCertificate*.
- *Scenario 2*: *ES2* is a member of *RDS_ElectricalSystem* and *InExplosiveGasZone*, and is inferred to not be a member of *HasEXCertificate*. Hence the ontology is inconsistent.
- *Scenario 3*: ES3 is a member of *RDS_ElectricalSystem*, *InExplosiveGasZone*, and *HasEXCertificate*, thus satisfying the axiom.

Scenario 1 demonstrates that, with this axiomatization, a lack of information is not detected; *ES1* is assumed to have an *EX Certificate* unless we specifically know it does not.

The benefit of this approach is most clearly exemplified by *Scenario 2*. Whenever a requirement is broken, the ontology becomes *inconsistent*. Then justification tools built into OWL reasoners can provide information about the offending assets as well as the requirements they break.

However, this approach's benefit can also be a disadvantage: if assets often do not satisfy requirements, the ontology will often be *inconsistent*. This severely limits the usability of the ontology for purposes other than using the inconsistency justification tools for requirement verification: due to inconsistency, basic functionalities of the reasoner, .e.g. classification and query answering, no longer function.

G.7.4.2.2 Axiom with Exceptions

As before, the Scope, Condition, and Demand are punned and interpreted as OWL classes and requirement is represented according to the following schema:

$$Scope \sqcap Condition\ \sqsubseteq Demand\ \sqcup' NonCompliant\ Subject'.$$

Our example is then represented by the following axiom:

$$RDS\_ElectricalSystem \sqcap InExplosiveGasZone \sqsubseteq HasEXCertificate \sqcup 'NonCompliant\ Subject'.$$

- *Scenario 1*: *ES1* is a member of *RDS_ElectricalSystem* and *InExplosiveGasZone*. It is inferred to be in the *union* of *HasEXCertificate* and *Non-Compliant Subject*, however the reasoner cannot state which of the classes it is in without further information.
- *Scenario 2*: *ES2* is a member of *RDS_ElectricalSystem* and *InExplosiveGasZone*. Since it is inferred to not a member of *HasEXCertificate*, it is inferred to be a member of *Non-Compliant Subject*.
- *Scenario 3*: *ES3* is a member of *RDS_ElectricalSystem*, *InExplosiveGasZone*, and *HasEXCertificate*. Thus, the axiom is *satisfied* by *ES3*.

By adding the class *Non-Compliant Subject* to the axiom, the lack of information in *Scenario 1* is no longer assumed to satisfy the requirement. In this case, more information is needed for the reasoner to determine whether the requirement is *broken* (e.g., the non-existence of an *EX Certificate* must be known) or whether the requirement is *satisfied* (e.g., the existence of an *EX Certificate* must be known).

The broken requirement in *Scenario 2* is handled by inferring that *ES2* is a member of *Non-Compliant Subject*. In this way, the ontology does not become *inconsistent* whenever a requirement is *broken*, and justification tools can be used to explain class membership in *Non-Compliant Subject*.

In *Scenario 3*, the axiom does not infer that *ES3* is a *Non-Compliant Subject*. Note that a different requirement could infer *ES3* to be a member of *Non-Compliant Subject*, as *HasEXCertificate* and *Non-Compliant Subject* are not *disjoint* classes.

G.7.4.2.3 Other approaches

There is ongoing work on using formalisms that extend OWL with the *defeasible semantics* necessary for reasoning over requirements, e.g., *rule-based formalisms* such as *SWRL* and *Datalog*, and *theorem provers* for *modal logic*.

G.8 BoMs and BoPs

Typically, the key elements of a process plant will be defined in terms of the following dimensions

- Plant breakdown structure
- Bill of Material (BoM) information (part structure breakdown of a product)
- Bill of Process (BoP) information (processual breakdown of a production process)

The scope of this use case is to cover the latter two. They are strongly related, indeed often intertwined, and have a bias towards discrete manufacturing industries, while in process industries you often have recipes with ingredients.

G.8.1 Status

Notes from the Editors:

- The discussion of functional objects can be greatly simplified, partly by reference to other use cases
- Details of the modelling diagrams, with “variants”, need review
- Provide a worked example as ontology, and OTTR templates to populate the recommended patterns

G.8.2 Source data schemas and systems

An experimental setup of a *quality station* built from typical Siemens automation components was used as an example. The station can measure the *height* of objects for quality control as part of an overall production process (e.g., an object milled from a raw block of metal can be checked to determine whether its physical dimensions are within a certain tolerance interval).

Figure G.8.1 shows a functional breakdown of the system together with a photo:

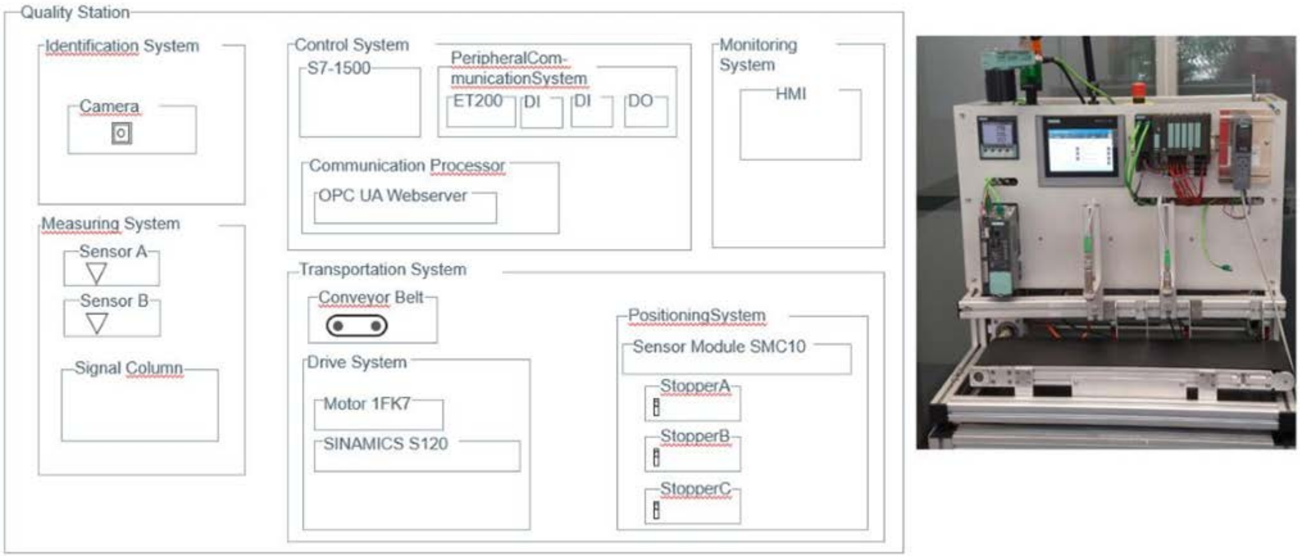


Figure G.8.1: Example of a quality station

The basic representation question in this area is how to distinguish objects that make a commitment to be *functional* vs *physical*. The as-built setup will have attributes related to the functional requirements as well as physical-specimen attributes, like serial numbers, that need to be captured.

All components are considered production equipment in this sense, but the respective equipment class cannot make any commitment as to whether an equipment is physical or functional. For each system component there is a choice of being either physical or functional; and the following arguments can be forwarded to support the choice. Traditionally the argument for functional vs physical criteria of identity are as follows:

- for a *functiona*l (possibly virtual) subsystem like the transportation system, its identity is rather defined by its formal/function of transportation of objects within the station, while its replaceable parts (such as specific sensors etc.) could be exchanged e.g. in the course of maintenance.
- for a *physical* component like the drive controller, its identity is rather defined by its material extent and characteristics, as in this use case, the specific controller that is replaced as a whole by another controller with different serial number during maintenance.

This use case does not cover the additional issue of the software PLC program included as part of the system; this is covered in Section G4.

G.8.3 Modelling

G.8.3.1 ISO 15926-14 Context

For the Part 14 representation here, each element of the system as designed is a *FunctionalObject*. Installed, physical parts (the ones with serial numbers) are related to the functional objects by the *installedAs* relation.

ProductionEquipment is defined as a subclass of *FunctionalObject*, and also of *PhysicalObject*:

```
Class: WorkCell
       SubClassOf: lis:FunctionalObject

Class: QualityStation
       SubClassOf:  WorkCell

Individual: myQualityStation
       Types: QualityStation
```

We represent all system parts as *FunctionalObject* individuals*. hasInstalled* (*installedAs*) links point to physical individuals for any entities that have serial numbers; noting that for replaceable parts, there will typically be more than one thing that is *installedAs* any of the functional objects over time. This is central to the way one can handle lifecycle of systems and components in part 14 (cf. Annex F). Time-stamps may be added to manage which items were installed at different periods of time.

G.8.3.2 Example

Two possible modelling variants are considered in this use case, represented by Figure G.8.2 and Figure G.8.3.

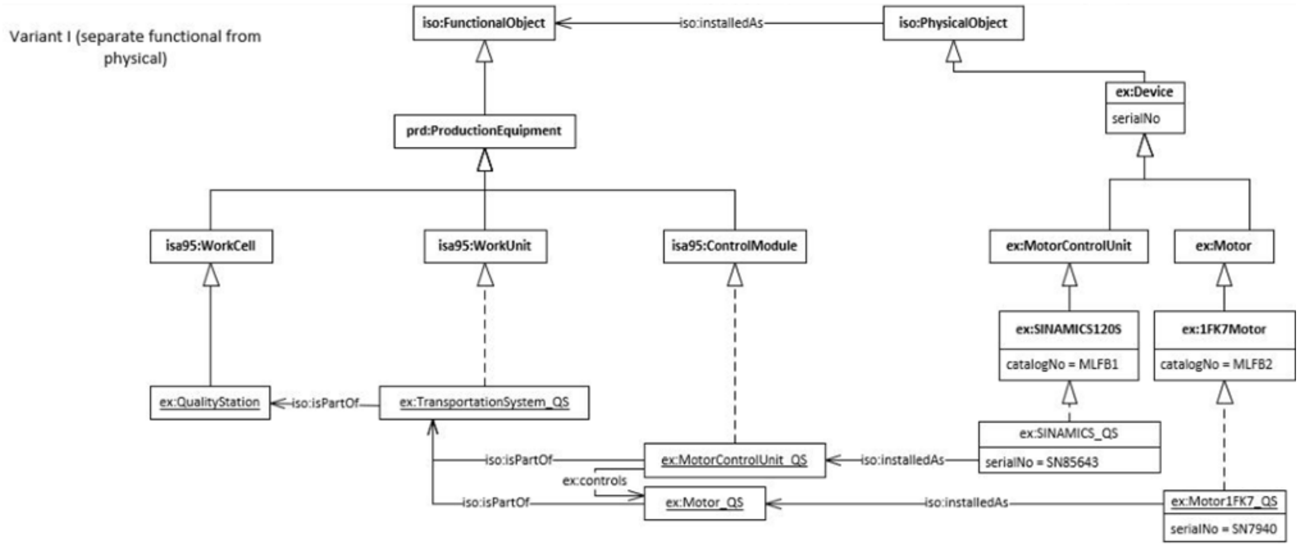
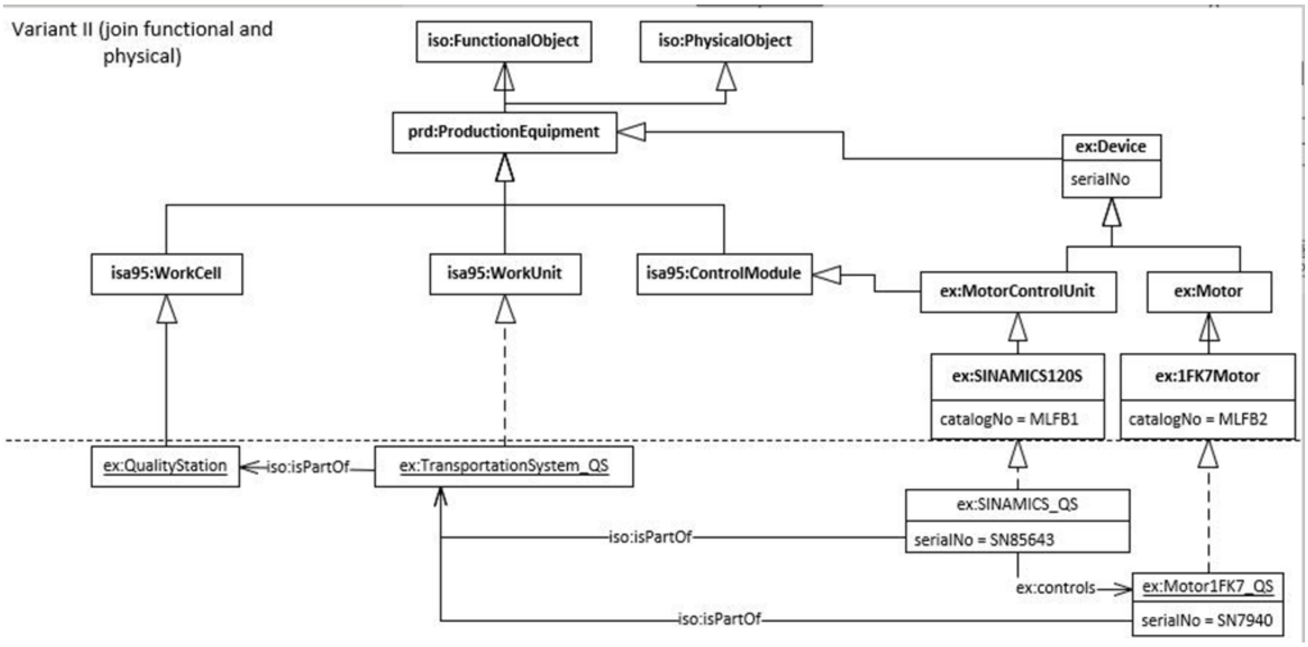


Figure G.8.2: Tracking installed components (variant I)



Variant I) in which tracking the installed components over time matters; and Variant II) where it does not matter, and a more parsimonious model is desirable for practical purposes that excludes any extra individuals in a plant structure representation.

G.8.3.3 **Explanatory notes**

It is important to be clear on what commitment a class like *ProductionEquipment*, extending from Part 14 in a core ontology for the production domain, should make with respect to an object being either functional or physical or both. A clear separation between functional component descriptions and their installations via *installedAs* would suggest to primarily focus on the functional hierarchy when describing a plant structure, as in *Variant I*. On the other hand, joining the physical and functional views would suit the view of simple UCs as in *Variant II.*

One may consider a functional object as a kind of variable, to be instantiated with *constants* that are the installed objects. This helps build the intuition on how the *hasInstalled* (*installedAs*) relation is used for representing that the *FunctionalObject* (the variable) is occupied by an artefact (other things than artefacts also possible). The intention is that one will have a part—whole break-down of the plant, using functional objects, and then point to installed artefacts where needed.

The advantage of this approach is that we obtain a straightforward way to check the specification that's carried by the functional object -- which is typically expressed by range constraints on data properties -- with the fixed attributes of the product individual: this can be achieved by transforming the ontology by declaring the *FunctionalObject* as the installed object, and executing a reasoner to identify inconsistencies. (Additional meta-data specifying which product individuals were installed in which periods of time would be required to make this possible.)

Variant II is indeed an example of such a transformed ontology. For *ProductionEquipment*, we have two alternatives. If *ProductionEquipment* can be the class of a piece of equipment in storage (i.e., equipment that hasn't been installed in a system, that doesn't contribute to a plant function), then it will mean: A member of *ProductionEquipment* is an artefact designed to perform a function that is usable in a production facility.

If *ProductionEquipment* is defined to be a class of equipment that participates in a plant system, it will not be appropriate to use it for something that's on the shelf in storage. In such a case, we still want

PhysicalObject to be a superclass, but more importantly, it will be a subclass of *System* -- carrying the restriction that any member is part of a system and contributes to the function of a system.

G.9 Physical quantities and units of measurement

The use case discusses how to represent physical quantities (also known as quantity kinds, which includes mass, length or temperature) and units of measures (such as kilogram, meter or kelvin).

G.9.1 Status

Notes from the Editors: This use case requires further development.

General comments:

- Re-focus the use case to provide more practical advice on the use of ISO 15926-14 (and, cutting down on detail about other ontologies)
- Improve analysis of *WoT TD, SOSA/SSN* and *iot.schema.org*
- Extend the discussion about *value (or measurement) scales* and how to include them into *ISO 15926-14*
- Discuss the notion of *uncertainty of a measurement* (including standard and relative standard uncertainty)
- Discuss the notion of *vector quantities* (e.g. velocity, acceleration and force) and how to represent them in an ontology, in particular, *direction*. Notice that the ontologies *OM* and *QUDT* do not address this problem
- Examples must be revised. In particular Figure 9.3 is incomplete and relevant definitions are missing.

Comments about *quantity values*:

- Quantity values (u observations) are a central notion for these ontologies. In fact, iot.schema.org and OM only explicitly define relations between quantity values (or observations) to a phenomenon but not the other way around.

- The use and benefits of short-cuts should be further investigated, and the conclusions of this analysis should be reflected in the document presenting ISO 15926-14

- Following QUDT, it is possible to introduce specific properties to represent the uncertainty of a quantity value (or measurement)

- The notion of *observation* may be introduced, as subclass of *lis:Activity*, as a mechanism to relate quantity values to phenomena, observable properties (physical quantities), processes and sensors

- The notion of observations also solves the problem of associating timestamps to quantity values, which is not addressed by the current version ISO 15926-14

- It is important to ensure interoperability when quantity values are represented using OM, QUDT (SOSA/SSN), WoT and iot.schema (to be further investigated)

Some comments about *system of quantities*:

- The classes, one for representing systems of quantities and one for representing systems of units, are not defined in the current version of ISO 15926-14. Similar to the class *lis:UnitOfMeasure*, these classes can be defined as subclasses of the class *lis:InformationObject*.

- Together with these classes, the necessary object properties may be defined to relate base and derived quantities (or units) to a system of quantities (or units)

Some comments about *dimensions*:

- The class representing dimensions is not defined in the current version of ISO 15926-14. Similar to the class *lis:UnitOfMeasure*, this class can be defined as subclasses of the class *lis:InformationObject*

- Together with the class for dimensions, seven data properties (one for each base dimension) must be defined

Some comments about *quantity kinds*:

- It follows the design pattern quantity kinds as classes

- It does not provide any class restriction for the class *lis:PhysicalQuantity* that states how individuals of this class are related to quantity values (*lis:ScalarQuantityDatum*), types of units (*lis:UnitOfMeasure*) and dimensions (not defined)

- It does not clarify the relation with existent systems of quantities (ISQ) and units (SI, Imperial, USCS, etc.)

- The distinction between base and derived quantity kinds is also missing

Some comments about *units of measurement*:

- The class *lis:UnitOfMeasure* must define (using class restrictions) relevant properties for individuals representing units of measurement including:

    - conversion multiplier and offset together with the unit used as reference for these conversion values,

    - dimension vector,

    - related quantity kinds,

    - related base (or reference derived) unit (for unit conversion) and

    - constituent units (if it is a composed unit)

- A hierarchy of classes of units of measurement based on relevant classes defined by OM and QUDT should also be considered. In particular:

    - Classes of units of measurement related to classes of quantity kinds (as in OM), including the class dimensionless unit

    - Classes of units of measurement related to systems of measurement such as SI or Imperial (as in QUDT)

    - Classes of compound units (as in OM)

- In general, ISO 15926-14 should provide more definitions related to quantity kinds and units of measurement which are very important topics in engineering. This should also facilitate the integration of ISO 15926-14 with OM and QUDT but also to a future units of measurement ontologies

G.9.2 Terminology

According to the Oxford dictionary, *quality* is a distinctive attribute or characteristic possessed by an entity (known as the bearer of the quality). In order for a quality to exist some other entity (or entities) must also exist [17].

Following the terminology from [16], the object or event that has a quality is referred to as a *phenomenon*. Similarly, a *(physical) quantity* is an observable property of a phenomenon that can be measured numerically, and it is determined by two main attributes: a *quantity kind* and a *quantity value*. A quantity kind is the aspect of phenomenon being measured such as length or mass. A quantity value as the magnitude of the quantity expressed as the product of a number and a unit. Other sources, such as Wikipedia, defines the notions of *concrete number*, which is a number associated to the things being counted and a denominate number is a type of concrete number where a unit of measure is related to it.

A *system of quantities (or measurement)* is a collection of units and quantities and rules related to each of them. Relevant examples of this type of systems are the *International System of Quantities (ISQ)* [19], the *Imperial system* and the *US customary measurement system*. Quantity kinds (similarly for units) are usually categorized in these systems as *base* and *derived quantities*. A base quantity is a physical quantity in a subset of a given system of quantities that is chosen by convention, where no quantity in the set can be expressed in terms of the others. A derived quantity is a quantity in a system of quantities that is a defined in terms of the base quantities of that system. The International System of Quantities (ISQ) [19], for instance, defines seven base quantities and units, and also many derived quantities and units. ISQ base quantities include *Length* (l), *Mass* (m), *Time* (t), *Electric Current* (I), *Thermodynamic Temperature* (T), *Amount of Substance* (n), and *Luminous Intensity* (Iv). Furthermore, ISQ base units are *Meter* (m), *Kilogram* (kg), *Second* (s), *Ampere* (A), *Kelvin* (K), *Mole* (mol) and *Candela* (cd), where each base units has its correspondent base quantity.

A *(physical) dimension* is defined in [16] as an abstraction of quantity ignoring magnitude and units. The notion of dimension was introduced to facilitate the characterization of quantities of the same kind (also known as commensurable quantities), without referring to specific units. Quantities of the same kind can be compared and therefore, conversion mechanisms can be defined for these quantities and units. This is summarized by the *principle of dimensional homogeneity*: only commensurable quantities (physical quantities having the same dimension) may be compared, equated, added, or subtracted. In particular, ISQ defines seven base dimensions (one for each base quantity) and they are also known by the same name but different symbol: *Length* (L), *Mass* (M), *Time* (T), *Electric Current* (I), *Thermodynamic Temperature* (Θ), *Amount of Substance* (N) and *Luminous Intensity* (J). *Derived quantities* are defined as the product of powers of the base dimensions. For instance, the derived quantity kind area is defined as L^2 (two times the dimension Length). Similarly, the derived quantity acceleration is defined as LT^{-2} (one time dimension Length divided by two times dimension Time). It is easy to verify that the quantities area and acceleration are not commensurable. Notice also that it may nevertheless be meaningless to compare or add two physical quantities with identical dimensions. For instance, torque and energy share the dimension L^2MT^{-2} but they are fundamentally different physical quantities.

Quality (and quantity) kinds can have more than one *value scale* (or measurement scale). In general, a value scale might have some of the following properties: identity, magnitude, equal intervals and absolute (or true) zero. The property *identity* means that each value has a particular meaning. For instance, the color red and the color yellow have different meanings. The property *magnitude* implies that values have an inherent order from smaller to larger. For instance, the value 8 is larger than the value 3 in the Richter scale. The property *equal intervals* enforces that differences between values anywhere on the scale is the same. For instance, the increase in temperature between 20 and 30°C is the same as the increase between 90 and 100°C. Finally, the property *absolute* (or true) *zero* means that the scale has an absolute (or true) zero value, below which no values exist. For instance, there is no duration or weight lower than 0.

[18] defines four types of value (measurement) scales: nominal, ordinal, interval and ratio. *Nominal scale* has discrete values and they do not overlap. This type of value scale only fulfills the property identity. For instance, the quality gender might have a nominal scale with the values male and female. Similarly, the quality color might have the nominal scale basic colors that might include the values red, blue or black. *Ordinal scale* has also categories and these categories are ranked in a certain order. This type of value scale has the properties identity and magnitude. The Mohs scale of mineral hardness and the Richter scale for earthquake sizes are examples of ordinal scales. *Interval scale* has ordered values with meaningful divisions, the magnitude between consecutive intervals are equal. This type of scale includes the properties identity, magnitude and equal intervals. Celsius and Fahrenheit scales are examples of interval scales. The last type of value scale defined by [18] is *ratio scale* which extends the type interval scale with the property absolute (or true) zero. Many quantity kinds in physical sciences such as mass, length and duration fit with this type of scale. In contrast to interval scales, ratios are now meaningful, and it is possible to say, for instance, that diameter 20 meter is twice diameter 10 meter. Both type of scales, interval and ratio, commonly express magnitude using numerical values in combination with units of measure.

G.9.3 Relevant references

This use case is based on the work described in the following references: [15, 16]. Other relevant sources of information considered during the preparation of this use case are the ontologies of units of measurement OM[20] and QUDT[21], the W3C recommendations SOSA/SSN[22] and WoT[23], and the IoT ontology iot.schema.org[24].

G.9.4 Scope

In the current version of the use case, we only discuss the following notions: *phenomenon* (or feature of interest), *quantity*, *quantity kind*, *quantity value*, *unit of measurement* and *physical dimension*. Other notions like *uncertainty of a measurement*, *types of value scales* and *vector quantities* are not part of the current version, but they might be studied in future versions.

G.9.5 Discussion

G.9.5.1 Prefixes

When referring to terms defined by *OM*, we use the prefix *om*. For the case of *QUDT*, we use the following prefixes: *qudt* (core terms), *quantity* (types of quantities), *quantitykind* (quantity kinds), *qkdv* (dimensions). The prefixes *sosa* and *ssn* refer to terms in *SOSA/SSN*. For *WoT*, we use the prefix *wot*, and for *iot.schema.org*, we use the prefix *iot*. For terms defined in *ISO 15926-14*, we use the prefix *lis* and for

[20] https://github.com/HajoRijgersberg/OM

[21] https://github.com/qudt/qudt-public-repo

[22] https://www.w3.org/TR/vocab-ssn/

[23] https://www.w3.org/TR/wot-thing-description/

[24] http://iotschema.org/

terms defined in the ontologies *Basic Formal Ontology* (BFO) and *Information Artifact Ontology* (IAO), we use the prefixes *bfo* and *iao*, respectively.

G.9.5.2 Phenomenon (or feature of interest)

The notion of *phenomenon* (or *feature of interest*) is not explicitly defined by QUDT and OM. OM includes the object property *om:hasPhenomenon* to relate an individual representing an observable property (*om:Quantity*) of a phenomenon to the individual representing this phenomenon. ISO 15926-14 defines the classes *lis:Activity* and *lis:Object* for representing a phenomenon. ISO 15926-14 also includes the object properties *lis:hasQuality* and *lis:hasPhysicalQuantity* to relate an individual representing a phenomenon to its quality (or physical quantity), with inverse relations *lis:qualityOf* and *lis:physicalQuantityOf*.

SOSA/SSN defines the class *sosa:FeatureOfInterest* to represent the notion of feature of interest (or phenomenon). Individuals of this class can be associated to individuals of the class *ssn:Property*, which represent qualities of a feature of interest, using the object property *ssn:hasProperty*. In addition, it is possible to relate a feature of interest to an individual of the class *sosa:Observation*, which represents a measurement, using the object property *sosa:isFeatureOfInterestOf*. The inverse relation is also possible using the property *sosa:hasFeatureOfInterest*.

WoT defines the class *wot:Thing* as an abstraction of a physical or virtual entity. An individual of this class can be related to an individual of the class *wot:PropertyAffordance* using the object property *wot:properties*. A property affordance represents a state of an individual of the class *wot:Thing*.

Similar to SOSA/SSN, *iot.schema.org* defines the class *iot:FeatureOfInterest* as the thing whose property is being estimated or calculated in the course of an observation. Individuals of the class *iot:Property* can be related to an individual of the class *iot:FeatureOfInterest* using the object property *iot:isPropertyOf*. The inverse of this property has not been defined (to be investigated).

G.9.5.3 Quantity values

A *quantity value* is defined as the magnitude of a quantity (kind) expressed as the product of a *number* and a *unit*. The ontologies we are considering in this discussion (QUDT, OM, ISO 15926-14, SOSA/SSN, WoT and iot.schema.org) coincide to define any quantity value as an *individual*, which represents the combination of a numerical value and a unit.

ISO 15926-14 defines the class *lis:ScalarQuantityDatum* to represent quantity values. This class is defined similar as the class *iao:IAO_0000032* (scalar measurement datum) in the *Information Artifact Ontology* (IAO). The following properties are also available:

- *lis:datumUOM*: relates a quantity value to its unit
- *lis:datumValue*: relates a quantity value to its numerical value

QUDT defines the class *qudt:QuantityValue* to represent quantity values as individuals of this class. The following properties can be used:

- *qudt:hasQuantityKind*: relates a quantity value to the appropriate quantity kind
- *qudt:unit*: relates a quantity value to its unit
- *qudt:value*: relates a quantity value to its numerical value

- *qudt:standardUncertainty*: relates a quantity value to a numerical value representing the standard uncertainty of the measurement

- *qudt:relativeStandardUncertainty*: relates a quantity value to a numerical value representing the relative standard uncertainty of the measurement

OM defines the class *om:Measure* to represent quantity values. This class does not include any class restriction or subclasses. In fact, the definitions of the individuals of this class partially rely on the definition of the subclasses of the class *om:Quantity*. The following properties can be used:

- *om:hasUnit*: relates a quantity value to its unit

- *om:hasNumericalValue*: relates a quantity value to its numerical value

SOSA/SSN indicates that the result, defined by the class *sosa:Result*, of a sosa:Observation or a sosa:Actuation can be a quantity value. However, SOSA/SSN does not recommend any particular approach for modelling quantity values. It is suggested the possibility of using specific ontologies of units of measurement such as OM and QUDT. For instance, the class *sosa:Result* could be considered equivalent to the class *om:Measure* (or *om:Point*) defined by *OM* or the class *qudt:QuantityValue* defined by *QUDT*[25]. Therefore, quantity values will be represented as individuals of the class sosa:Result and linked to at least one individual of the classes sosa:Observation or sosa:Actuation using the object property *sosa:isResultOf*, which is the inverse property of *sosa:hasResult*. Notice that an individual of the class *sosa:Result* does not carry out information about the time the measurement was taken. This is information that belongs to the individuals of the class *sosa:Observation* or *sosa:Actuation*. In addition, it is possible to simplify the representation of quantity values using the data property *sosa:hasSimpleResult*. This is similar to the notion of *short-cuts* for quantity values in *ISO 15926-14*.

WoT defines the class *wot:PropertyAffordance* to represent observable properties of an individual of the class *wot:Thing*, which represents and abstraction of a physical or virtual entity. Each individual of the class wot:Thing is related to one or more individuals of the class *wot:PropertyAffordance* using the object property *wot:properties*. The class wot:PropertyAffordance is defined as subclass of the classes *wot:InteractionAffordance* and *wot:DataSchema*. The later provides a collection of properties to describes data formats base on *JSON-Schema*[26]. The properties *wot:type*, *wot:unit*, and *wot:format* are relevant for the representation of quantity values. It seems, however, that WoT does not provide a specific class for quantity values (to be further investigated).

iot.schema.org refers to the class *schema:QuantitativeValue* defined by schema.org to represent physical quantities and their quantity values. This class is a subclass of *schema:StructureValue* that allows the definition of complex schemas (similar to wot:DataSchema). The following properties are relevant when representing quantity values: *schema:value* (any type of simple or complex value including schema:StructureValue), *schema:unitCode* (based on UN/CEFACT Common Code), *schema:unitText* (units represented as strings). The class schema:QuantitativeValue and how it can be used to represent quantity values should be further investigated.

ISO 15926-14 recommends the use of *short-cuts* (cf. Section E.5) to simplify modelling and improve reasoning performances by leaving out some classes and properties. Consider the example of the definition of the class *ex:BigHammer* (cf. Section E.1):

```
Class: ex:BigHammer
```

[25] https://www.w3.org/TR/vocab-ssn/#quantity-values-and-unit-of-measures

[26] https://json-schema.org/

```
        EquivalentTo: ex:hasMass some
            (lis:qualityQuantifiedAs some
                (lis:datumUOM value ex:kilogram and lis:datumValue some float[> 1]))
```

The definition of the class ex:BigHammer can be simplified as follows:

```
Class: ex:BigHammer

        EquivalentTo: ex:has_mass_in_kilogram some float[> 1]
```

Some comments about *short-cuts*:

- Notice that the simplified definition of the class *ex:BigHammer* in the previous example does not include additional annotation property assertions to indicate the unit of measurement *ex:kilogram* and the quantity *ex:Mass*. Moreover, ISO 15926-14 does not define these annotation properties for units of measure and quantities.

- The suitability and potential benefits of defining short-cuts might depend of each use case and it should be considered carefully. For instance, the use of subclass relations instead of equivalent relations in class restrictions might alleviate the impact on reasoning performances and the need of implementing short-cuts. Similarly, the requirement of representing many possible combinations of units and physical quantities might also discourage this practice

G.9.5.4 **Systems of quantities and units**

Units of measurement are organised on *systems of units* that they also define how units are related to each other. Popular systems of units include the *International System of Units* (SI), the *Imperial System of Units* (IS or Imperial) and the *US Customary System of Units* (USCS). Despite the wide adoption of SI, other system of units such as Imperial and USCS are still popular. This implies that mechanisms for conversion between units of different systems of units must be considered.

OM assumes ISQ as the only system of quantity kinds. Quantity kinds are defined as subclasses of the class *om:Quantity*. OM also defines the class *om:SystemOfUnits*, where several system of units such as IS (*om:InternationalSystemOfUnits*) are defined as named individuals. Despite OM including some units defined by Imperial or USCS, these systems of units are not explicitly defined in OM.

QUDT includes the class *qudt:SystemOfQuantityKinds* which defines named individuals representing *systems of quantity kinds* (e.g. *soqk:SOQK_ISQ* , *soqk:SOQK_IMPERIAL*, *soqk:SOQK_USCustomary*). Because only ISQ seems to be defined as a system of quantities, it might not be correct to define additional system of quantities (to be further investigated). Each of the individuals representing a system of quantities are associated to individuals representing suitable base and derived quantity kinds, and system of units. This is done using the properties *qudt:hasBaseQuantityKind*, *qudt:systemDerivedQuantityKind* and *qudt:hasUnitSystem*. Notice that these properties are more specific that the properties used when defining the class qudt:SystemOfQuantityKinds (to be further investigated).

In addition to system of quantities, *QUDT* also defines *systems of units* represented as named individuals (e.g. *sou:SOU_IMPERIAL*, *sou:SOU_SI* and *sou:SOU_USCS*) of the class *qudt:SystemOfUnits*. The following properties (not an exhaustive list) can be used to define a relation between a system of units and relevant units: *qudt:hasBaseUnit* (e.g. *unit:KiloGM*), *qudt:hasDefinedUnit* (e.g. *unit:KiloGM-PER-SEC2*), *qudt:hasDerivedCoherentUnit* (e.g. *unit:KiloGM-PER-SEC2*), *qudt:hasPrefixUnit* (e.g. *unit:Centi*). Notice that the distinction between base and derived units makes more sense in SI but QUDT also considers this distinction for other system of units (e.g. *sou:SOU_CGS*).

ISO 15926-14 does not provide predefined classes or properties to define systems of quantity or units.

SOSA/SSN relies on OM and QUDT ontologies for relevant information about systems of quantities and units. *WoT* and *iot.schema.org* seems to do not make any reference to this topic in their specification (to be further investigated).

G.9.5.5 **Dimensions**

A *(physical) dimension* is defined as an abstraction of a quantity ignoring magnitude and unit. The notion of dimension was introduced to facilitate the characterization of quantities of the same kind (also known as commensurable) without referring to specific units. *ISQ* defines *seven base dimensions* (one for each base quantity) and they are also known by the same name but different symbol: *Length* (L), *Mass* (M), *Time* (T), *Electric Current* (I), *Thermodynamic Temperature* (Θ), *Amount of Substance* (N) and *Luminous Intensity* (J). *Derived quantity kinds* are defined as the product of powers of the base dimensions. For instance, the derived quantity kind *area* is defined as L^2 (two times the dimension Length). Similarly, the derived quantity kind *acceleration* is defined as LT^{-2} (one time dimension Length divided by two times dimension Time). It is easy to verify that the quantities area and acceleration are not commensurable. Notice also that it may nevertheless be meaningless to compare or add two physical quantities with identical dimensions. For instance, *torque* and *energy* share the dimension L^2MT^{-2} but they are fundamentally different physical quantity kinds.

When building hierarchies of classes (or properties) of quantity kinds, dimensions play a key role because the subclasses (or subproperties) of a class (or property) representing a quantity kind must have the same dimension. For instance, the class *ex:DryMass* (or the property *ex:hasDryMass*) can be defined as a subclass of *ex:Mass* (or the property *ex:hasMass*) because both have the same dimension (Mass, M).

ISO 15926-14 does not define a classes and the required data properties for dimensions.

OM and *QUDT* follow a very similar approach when defining dimensions as dimension vectors, a symbolic expression that define a dimension as the product of the powers of the base dimensions. Each dimension vector is represented using a named individual and the symbolic expression is modelled using seven data property assertions, one for each base dimension. If an individual is asserted using the number zero, it means that the base dimension is not relevant for the characterization of a quantity kind or units of measure. For instance, the individual representing the base dimension mass is only asserted to the integer zero except for the data property that refers to the power of the mass dimension, which is the integer one. Noticed that negative numbers are also possible (e.g. the dimension acceleration).

OM defines the class *om:Dimension* for representing dimensions. The definition of this class does not include any class restriction, but OM also defines seven data properties for each base dimension (e.g. *om:hasSIMassDimensionExponent)*.

QUDT defines the class *qudt:QuantityKindDimensionVector* for representing dimensions. It includes the class restrictions to indicate which data properties can be used to represent the power of each base dimension (e.g. *qudt:dimensionExponentForMass*).

SOSA/SSN relies on *OM* and *QUDT* ontologies for relevant information about dimensions. *WoT* and *iot.schema.org* seems to do not make any reference to this topic in their specification (to be further investigated).

G.9.5.6 **Quantities and quantity kinds**

A (physical) *quantity* is an observable property of an object or activity that can be measured numerically. Quantities are determined by two main attributes: kind and magnitude. The first attribute, a *quantity kind*, is the aspect of phenomenon being measured such as length or mass. The second attribute, a *quantity value*, is the magnitude of the quantity expressed as the product of a number and a unit. *ISQ* defines seven base quantities and multiple derived quantities. ISQ base quantities include *Length* (l), *Mass* (m), *Time* (t), *Electric Current* (I), *Thermodynamic Temperature* (T), *Amount of Substance* (n), and *Luminous Intensity* (I_v).

There are at least three well-known design patterns to represent quantity kinds: *as individuals, as classes* and *as properties* [16]. In the first design pattern, quantity kinds are represented as named individuals and asserted to a class that represents all quantity kinds. To specify that a quantity kind is more general or more specific than other quantity kind (e.g. the individual representing the quantity kind *dry mass* is more specific than the individual representing the quantity kind *mass*), a specific object property is used. In the second design pattern, quantity kinds as classes, quantity kinds are defined as classes and generalization or specialization relations are specified using subclass relations (e.g. the class representing the quantity kind *dry mass* is defined as a subclass of the class representing the quantity kind *mass*). In the third design pattern, quantity kinds as properties, quantity kinds are defined as properties and generalization or specialization relations are specified using subproperty relations (e.g. the property representing the quantity kind *has dry mass* is defined as a subproperty of the property representing the quantity kind *has mass*). Notice that in all design patterns, the definition of generalization and specialization relations are related to the notion of dimensions. Only quantity kinds of the same dimension can be part of this type of relations.

ISO 15926-14 defines quantity kinds as subclasses of the class *lis:PhysicalQuantity*, following the design pattern *quantity kind as classes*. Generalization or specialization relations between quantity kinds are defined using subclass relations. This design pattern is also applied by *BFO-IAO*. The definition of the class lis:PhysicalQuantity does not include any class restriction to state relations with dimensions (not defined by ISO 15926-14) and types of units. In addition, the capability of supporting different systems units is not addressed by the current specification.

QUDT defines quantity kinds as named individuals of the class *qudt:QuantityKind*, following the design pattern *quantity kind as individuals*. Generalization or specialization relations between quantity kinds are defined using the object properties *qudt:generalization* (or *skos:broader*) and *qudt:specialization* (or *skos:narrower*). In the latest releases of QUDT, we observed that SKOS semantic properties are replacing QUDT properties for defining generalization of specialization relations between quantity kinds. In addition to these object properties, there are other relevant properties worth mentioning:

- *qudt:dimensionVectorForSI* relates an individual of type quantity kind with an individual of type dimension vector (*qudt:QuantityKindDimensionVector_SI*) according to ISQ

- *qudt:symbol* is a data property that relates an individual of type quantity kind with a string representing a symbol for the quantity kind

- *qudt:qkdvNumerator* and *qudt:qkdvDenominator* are relevant properties when defining dimensionless quantities obtained as ratios of quantities that are not dimensionless (related to dimensional analysis)

- *qudt:applicableUnit* relates an individual of type quantity kind with one or more individuals representing applicable units of measure for this quantity kind

In addition to the class *qudt:QuantityKind*, QUDT also defines the class *qudt:Quantity*, which represents a relation between quantity values of the same kind and related to the same entity. Thus, individuals of this class have a similar role as individuals of the class *om:Quantity* and *lis:ScalarPhysicalQuantity*.

OM defines quantity kinds as subclasses of the class *om:Quantity*, following the design pattern *quantity kind as classes*. Generalization or specialization relations between quantity kinds are defined using subclass relations. The individuals of a class representing a quantity kind defines a relation between a phenomenon (or feature of interest) and the related quantity values. This is possible using the object properties *om:hasValue* and *om:hasPhenomenon*. In addition, the definition of each quantity kind restricts the dimension and the type of units of measure that can be use in related quantity values. For instance, the class *om:Mass* includes the following class restrictions (in Manchester syntax):

```
Class: om:Mass

    SubClassOf: om:Quantity    SubClassOf: om:hasDimension value om:mass-Dimension

    SubClassOf: om:hasValue only (om:hasUnit only om:MassUnit)
```

OM also defines quantity kinds as individuals (using *punning*) but only to explicitly relate each quantity kind to relevant applicable units using the object property *om:commonlyHasUnit*. For instance, the individual *om:Mass* is related to the unit *om:kilogram* as follows (in Manchester syntax):

```
Individual: om:Mass
     Facts: om:commonlyHasUnit om:kilogram, …
```

One main difference between OM and QUDT is that the former is heavily focused on ISQ (quantities and dimensions) and SI (units of measure), whereas QUDT tries to accommodate other systems of units including Imperial and USCS. In industry areas like oil & gas, it is still very common to use different systems of units. Accommodating different systems of units, however, increase modelling complexity as more specialized classes and properties are required.

SOSA/SSN defines the class <u>*ssn:Property*</u> to refer to the properties of a feature of interest. This class is defined as a subclass of *dul:Quality* and it has as subclasses: *sosa:ObservableProperty* and *sosa:ActuableProperty*. SOSA/SSN does not stablish a preferable design pattern for modelling quantity kinds. However, it is suggested the possibility of using specific ontologies of units of measurement such as OM and QUDT that include definitions of quantity kinds[27].

WoT provides a mechanism to extend *Thing Descriptions* with additional *Vocabulary Terms*, named *TD Context Extension*. Using this mechanism is possible to refer to quantity kinds defined by specialised ontologies such as OM and QUDT (to be further investigated). In addition, the following mappings with SOSA/SSN has been identified (in Manchester syntax):

```
Class: wot:Thing
  SubClassOf: ssn:System or sosa:Platform or sosa:FeatureOfInterest

Class: wot:InteractionAffordance
  SubClassOf: ssn:forProperty some ssn:Property
```

iot.schema.org defines the class *iot:Property*, which is a subclass of *iot:InteractionPattern*, to refer to the properties of a feature of interest. Subclasses of iot:Property such as *iot:Temperature* or *iot:Humidity* are examples of quantity kinds. Therefore, iot.schema.org adopted the design pattern *quantity kinds as classes*. The following properties are relevant in the definition of the class *iot:Property*:

- *iot:isPropertyOf* stablishes relations with individuals of type *iot:FeatureOfInterest*
- *iot:isObservedBy* stablishes relations with individuals of type *iot:Sensor*
- *iot:providedOutputData* stablishes relations with individuals of type *schema:PropertyValue* (a property value-pair) or *schema:PropertyValueSpecification*
- *iot:acceptsInputData* stablishes relations with individuals of type *schema:PropertyValue* (a property value-pair) or *schema:PropertyValueSpecification*

[27] https://www.w3.org/TR/vocab-ssn/#quantity-values-and-unit-of-measures

G.9.5.7 Units of measurement

A *unit of measurement* is a definite magnitude of a quantity, defined and adopted by convention or by law, that is used as a standard for measurement of the same kind of quantity. For instance, the unit of measurement *kilogram* is defined in terms of the mass of the *International Kilogram Prototype*, a platinum cylinder stored at the *International Bureau of Weights and Measures* in France.

Units of measurement are organised in systems of measurement (or units) that define rules between units. Popular systems of units include the *International System of Units* (SI), the *Imperial System of Units* (IS or Imperial) and the *US Customary System of Units* (USCS). SI is the only *coherent system of measurement* among them, meaning that all its units are either *base units*, which are units that cannot be expressed in terms of other units, or are *derived* from the base units without using any numerical factors other than one. SI defines seven base units, which are the *second* (time, s), *metre* (length, m), *kilogram* (mass, kg), *ampere* (electric current, A), *kelvin* (thermodynamic temperature, K), *mole* (amount of substance, mol), and *candela* (luminous intensity, cd). SI also defines twenty-two derived units including the *Pascal* (Pa), which is a used to quantify (internal) pressure, and it can be expressed using SI base units as follows: kg x m^{-1} x s^{-2}. In addition, SI defines twenty *prefixes* that precede a unit of measurement to indicate a power-of-ten (i.e. decimal) multiple or sub-multiple of the unit. Notice that the use of prefixes with coherent SI units produce units that are no longer coherent, because the prefix introduces a numerical factor other than one. However, there is one exception in SI, for historical reasons. This is the kilogram, the only coherent SI unit which includes a prefix.

Quantity values of the same kind are often represented using different units of measurement. Therefore, *unit conversion* becomes crucial to compare these quantities or to operate with them. This is a multi-step process that commonly involves multiplication or division by a numerical factor, selection of the correct number of significant digits, and rounding. For quantities that do not fit into a ratio scale but into an interval scale, such as temperature, unit conversion involves also subtraction or addition of a numerical offset. For instance, the conversion from Celsius to Kelvin is done by adding the numerical offset 273.15 to the numerical value of a quantity value.

ISO 15926-14 includes the class *lis:UnitOfMeasure* for units of measurement, which is defined as a subclass of *lis:InformationObject*, and it has a subclass the class *lis:Scale*. Neither of these classes include any class restrictions that help to better understand their meaning and how their individuals are related with other individuals. The object property *lis:datumUOM*, with domain *lis:QuantityDatum* and range *lis:UnitofMeasure*, is defined to relate a quantity value to a unit of measurement (both represented as individuals).

QUDT defines units as individuals of the class *qudt:Unit*. This class also has several subclasses including:

- *qudt:BaseUnit* refers to base units defined by the systems of measurement considered in QUDT (e.g. *unit:A*, ampere)
- *qudt:DimensionlessUnit* refers to quantities without units (e.g. *unit:MACH*, mach)
    - *qudt:CurrencyUnit* refers to currencies (e.g. *unit:Euro*, euro)
    - *qudt:CountingUnit* refers to counts of things (e.g. *unit:PERCENT*, percent)
- *qudt:LogarithmicUnit* refers to (dimensionless) quantities in logarithmic scale using units (e.g. *unit:DeciB*, decibel)
- *qudt:PrefixUnit* refers to the prefixes of prefixed units
    - *qudt:BinaryPrefixUnit* refers to binary prefixes (e.g. *unit:Kibi*, kilobinary)

    - *qudt:DecimalPrefixUnit* refers to decimal prefixes (e.g. *unit:Giga*, giga)

- *qudt:ScaledUnit* refers to prefixed units
    - *qudt:BinaryScaledUnit* refers to binary prefixed (e.g. *unit:KiloBYTE*, kilobyte)
    - *qudt:DecimalScaledUnit* refers to decimal prefixed units (e.g. *unit:KiloGM*, kilogram)
- *qudt:StandardsUnit* refers to units defined by a system of measurement
    - *qudt:NonSI-Unit* refers to units defined by a system of measurement that it is not SI
        - *qudt:CGS-Unit* refers to units defined by the system of measurement *CGS* (e.g. *unit:GM*, gram)
        - *qudt:ImperialUnit* refers to units defined by the system of measurement *Imperial* (e.g. *unit:HP*, horsepower)
        - *qudt:International-CustomaryUnit* refers to units defined by the system of measurement *International customary units*
        - *qudt:MKS-Unit* refers to units defined by the system of measurement *MKS* (e.g. *unit:M*, metre)
        - *qudt:US-CustomaryUnit* refers to units defined by the system of measurement *USCS* (e.g. *unit:IN*, inch)
    - *qudt:SI-Unit* refers to units defined by the system of measurement *SI* (e.g. *unit:KiloGM*, kilogram)
    - *qudt:US-SurveyUnit* refers to units defined by the system of measurement *US survey units* (e.g. *unit:FT_US*, US Survey Foot)

The following properties represent a selection of all properties defined by QUDT with respect to the class *qudt:Unit*:

- *qudt:hasQuantityKind* is an object property that refers to the related quantity kind
- *qudt:isUnitOfSystem* is an object property that refers to related systems of measurement
- *qudt:iec61360Code* is a data property with range string that refers to unit codes defined by IEC 61360
- *qudt:ucumCode* is a subproperty of the data property *skos:notation* with range typed literal that refers to a suitable UCUM code[28]
- *qudt:conversionMultiplier* represents the conversion multiplier to be applied to the numerical value of any quantity defined using this unit to a base (or a reference derived) unit
- *qudt:conversionOffset* represents the conversion offset to be applied to the numerical value of any quantity defined using this unit to a base (or a reference derived) unit

[28] https://ucum.org/trac

- *qudt:uneceCommonCode* is a data property with range string that indicates the *UN/CEFACT Common Code*[29]. These codes are also supported by iot.schema.org

- *qudt:symbol* is a data property with range string that indicates a recommended symbol to represent the unit according to QUDT

For instance, the conversion multiplier and offset of the unit *kilowatt hour* (*unit:KiloW-HR*) are 3.6e^{6} and 0.0, respectively. These values are defined with respect to the reference derived unit *Joule* (*unit:J*), which is of the same kind, and it has as conversion multiplier 1.0 and offset 0.0. So, 1.0 kilowatt hour is equivalent to 3.6e^{6} joules.

OM defines units as individuals of the class *om:Unit*. This class does not have any class restriction but all individuals representing units of measurement has been asserted to this class. The class *om:Unit* also has many subclasses which nearly replicates the class hierarchy for quantity kinds. This facilitates the definition of class restrictions that state that quantity values of certain kind can only use units of measurement suitable for that kind. For instance, the class *om:MassUnit* represents all units defined by OM that are relevant for the quantity kind *om:Mass* and this class restricts the units for quantity values to be of type *om:MassUnit* as follows (in Manchester syntax):

```
Class: om:Mass
    SubClassOf: om:Quantity
     SubClassOf: om:hasDimension value om:mass-Dimension
     SubClassOf: om:hasValue only (om:hasUnit only om:MassUnit)
```

The units included in the definition of *om:MassUnit* are not only SI units but also units for other systems of measurement including CGS, Imperial or USCS (the latter two are not defined specifically as a system of measurement in OM). The class *om:MassUnit* is defined as follows (in ManchesterSyntax):

```
Class: om:MassUnit
    SubClassOf:   om:Unit
    EquivalentTo: om:PrefixedGram or om:PrefixedTonne or
       om:PrefixedUnifiedAtomicMassUnit or ({om:InternationalUnit,
       om:carat-Mass, om:grain, om:gram, om:hundredweight-British,
       om:hundredweight-US, om:milligramRAE, om:ounceApothecaries,
       om:ounceAvoirdupois, om:pennyweight-Troy, om:poundApothecaries,
       om:poundAvoirdupois, om:slug, om:solarMass, om:ton-Long,
       om:ton-Short, om:ton-ShortAssay, om:tonne, om:unifiedAtomicMassUnit})
```

In addition to the classes for types of units just mentioned earlier, OM defines few more specialised subclasses, some of them are related to compound units. The individuals of these classes have specific properties to define an exponent or to differentiate between two terms in a multiplication or division expression. The classes are the following:

- *om:CompoundUnit*
    - *om:UnitDivision* refers to units defined as the division of two units (e.g. *om:metrePerSecond-Time*)
    - *om:UnitExponentiation* refers to units defined as the power of a reference unit (e.g. *om:cubicMetre*)
    - *om:UnitMultiplication* refers to units defined as the multiplication of two units (e.g. *om:kilowattHour*)

[29] https://www.unece.org/

- *om:PrefixedUnit* refers to units with a prefix (e.g. *om:millimetre*)

- *om:SingularUnit* refers to units that are not compound units (e.g. *om:ampere*)

The specification of the constituent units of a compound unit complements the information given by the related quantity kind (and the related dimension vector) and it helps to better understand the nature of the unit and how it is related to other units. As it was reported in [15], the class restrictions defined for the class *om:PrefixedUnit* and its subclasses have a strong impact on reasoning performances using the state of the art OWL 2 reasoner *HermiT*[30]. For practical applications, the definition of these classes must be revised.

In addition to the issue of reasoning performances, many compound units do not include the necessary information to compute conversion of units. As it is reported in [15], this approach difficult the conversion of units using SPARQL, which is not a problem in QUDT. For instance, the unit *om:kilowattHour* does not include any conversion factor in its definition. Therefore, we need to find the information in the definition of its constituent units starting from the units *om:kilowatt* and *om:hour*. Another problem is that OM units do not have a conversion offset when they refer to quantities that do not fit in a ratio scale but in an interval scale (e.g. Temperature).

SOSA/SSN indicates that the result of a *sosa:Observation* or a *sosa:Actuation* can be a quantity value. However, SOSA/SSN does not recommend any particular approach for modelling quantity values. It is suggested the possibility of using specific ontologies of units of measurement such as OM and QUDT. For instance, the class *sosa:Result* could be considered equivalent to the class *om:Measure* (or *om:Point*) defined by OM or the class *qudt:QuantityValue* defined by QUDT[31].

WoT includes in the definition of the class *wot:DataSchema* a collection of properties to describe data formats base on *JSON-Schema*[32]. A specific property, *wot:unit*, is defined to refer to unit of measurement as strings. WoT also provides a mechanism to extend *Thing Descriptions* with additional *Vocabulary Terms*, named *TD Context Extension* (to be further investigated). It seems this mechanism makes possible to refer to units of measurement defined by external vocabularies such as OM and QUDT ontologies (e.g. https://www.w3.org/TR/wot-thing-description/#example-28).

iot.schema.org defines the data properties *schema:unitCode* and *schema:unitText* to indicate a unit of measurement related to a quantity value (of type *schema:QuantitativeValue*) or a property value (of type *schema:QuantitativeValue*). The range of the data property *schema:unitCode* is an URL or a string of unit code based on UN/CEFACT Common Code[33]. The range of the data property *schema:unitText* is a string and this property allows to name units using a different identifier than the ones defined by UN/CEFACT Common Code. This might allow the use of unit identifiers from OM or QUDT (to be further investigated).

G.9.6 Examples

G.9.6.1 Representation of quantity values

This example illustrates how to represent a quantity value related to a physical quantity (kind) of a physical object. For quantity kinds, the modelling pattern *quantity kinds as classes* has been chosen. In the example, the individual *hbig* represents the physical object *big hammer*, which is of type *Hammer*. The individual *hbig_mass* of type *Mass*, represents that the hammer *hbig* has the physical quantity mass. The individual *hbig_mass_datum* of type *MassQuantityDatum*, represents a measurement of the mass of the

[30] http://www.hermit-reasoner.com/
[31] https://www.w3.org/TR/vocab-ssn/#quantity-values-and-unit-of-measures
[32] https://json-schema.org/
[33] https://github.com/schemaorg/schemaorg/wiki/Using-UN-CEFACT-Codes

hammer. The uncertainty of the measurement is represented using the data properties *standardUncertainty* and *relativeStandardUncertainty* (as it is done in QUDT). In addition, the measurement includes a timestamp indicating when it was taken.

The class *ScalarPhysicalQuantity* represents scalar physical quantities only, where quantity values are of type *ScalarQuantityDatum*, meaning they are defined by a numerical value (cf. data property *datumValue*) and (in many cases) a unit of measurement (cf. object property *datumUOM*). It is expected a specific class for vector physical quantities (i.e. *VectorPhysicalQuantity*) but for the sake of simplicity is not included in the example.

To emphasize the use of reasoning for detecting inconsistent statements, two class restrictions have been added, one for the class *ex:Mass* and one for the class *MassQuantityDatum*. These class restrictions, however, are insufficient for detecting inconsistencies (c.f. Annex E) but simpler enough to facilitate the understanding the diagram.

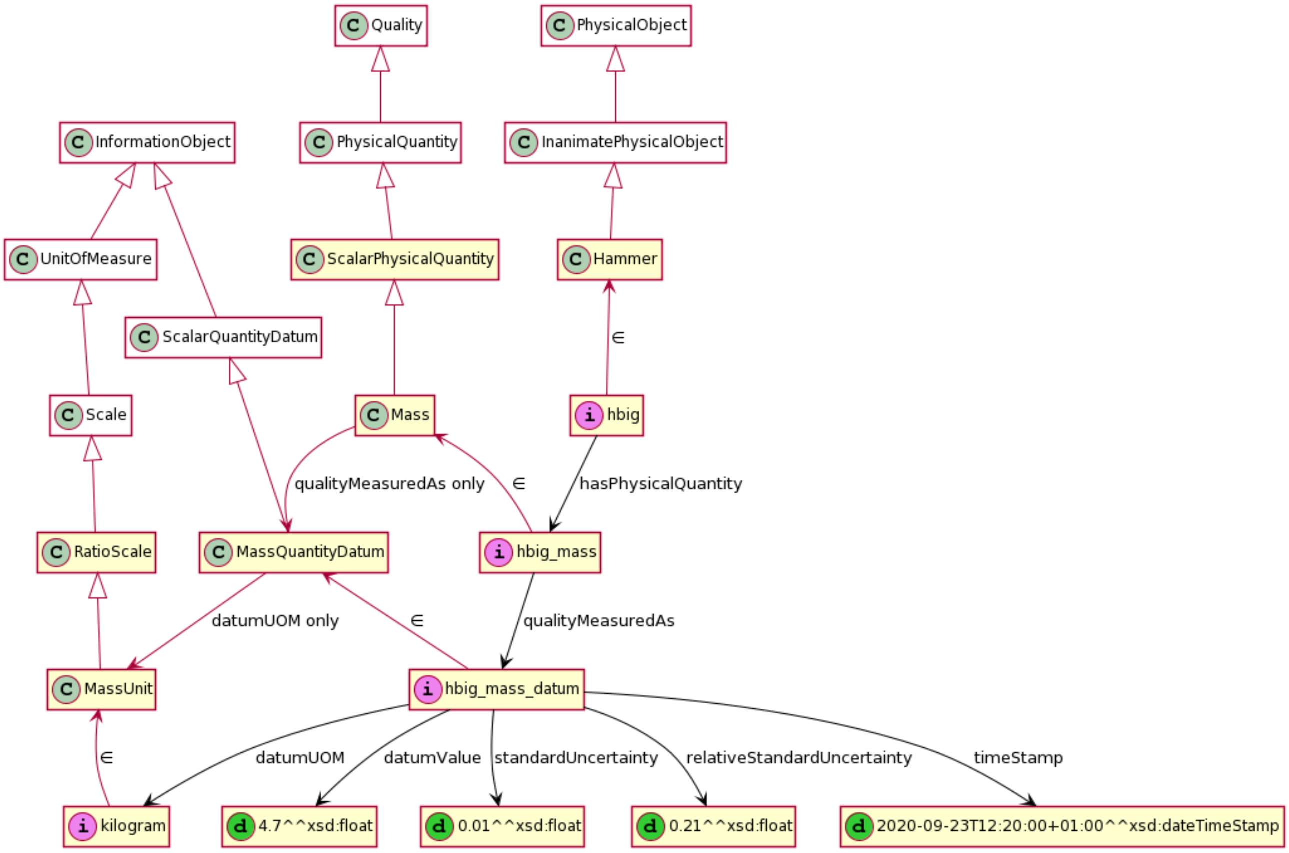


Figure G9.1: Example of a measurement of the mass of a physical object

G.9.6.2 Representation of quantity values using short-cuts

ISO/TR 15926-14 (page 18) recommends the use of *short-cuts* (c.f. Section E.5 in this document) to simplify modelling and improve reasoning performances by leaving out some classes and properties. The previous example can be simplified by replacing the individual *hbig_mass_datum* representing a quantity value with a data property assertion that relates the individual *hbig* (representing the physical object *big hammer*) with the numerical value *4.7^^xsd:float*. This is done in this example using the data property *hasMass_kg*. This data property is annotated using two annotation properties: *hasUnit* and *hasQuantityKind*. The first one indicates the unit of measurement, in this case kilogram, and the second property indicates the quantity kind, which is Mass in the example. It becomes obvious that this approach might not scale well in use cases dealing with many quantity kinds and units of measurement. In addition, the data property assertion *hasMass_kg (hbig, 4.7^^xsd:float)* is also annotated to indicate when the measurement was taken and additional information about the uncertainty of the measurement.

In this example, it can be observed that quantity kinds are represented as individuals of the class *InformationObject* (similar as the unit of measurement kilogram) and not as an individual of the class Aspect like in the previous example. The individual *massQuantityKind* represents the notion of quantity kind *Mass*, whereas in the previous example, the individual *hbig_mass* represents that the object *hbig* has the physical quantity mass. Notice that quantity kinds represented as classes and individuals can coexist in the same ontology.

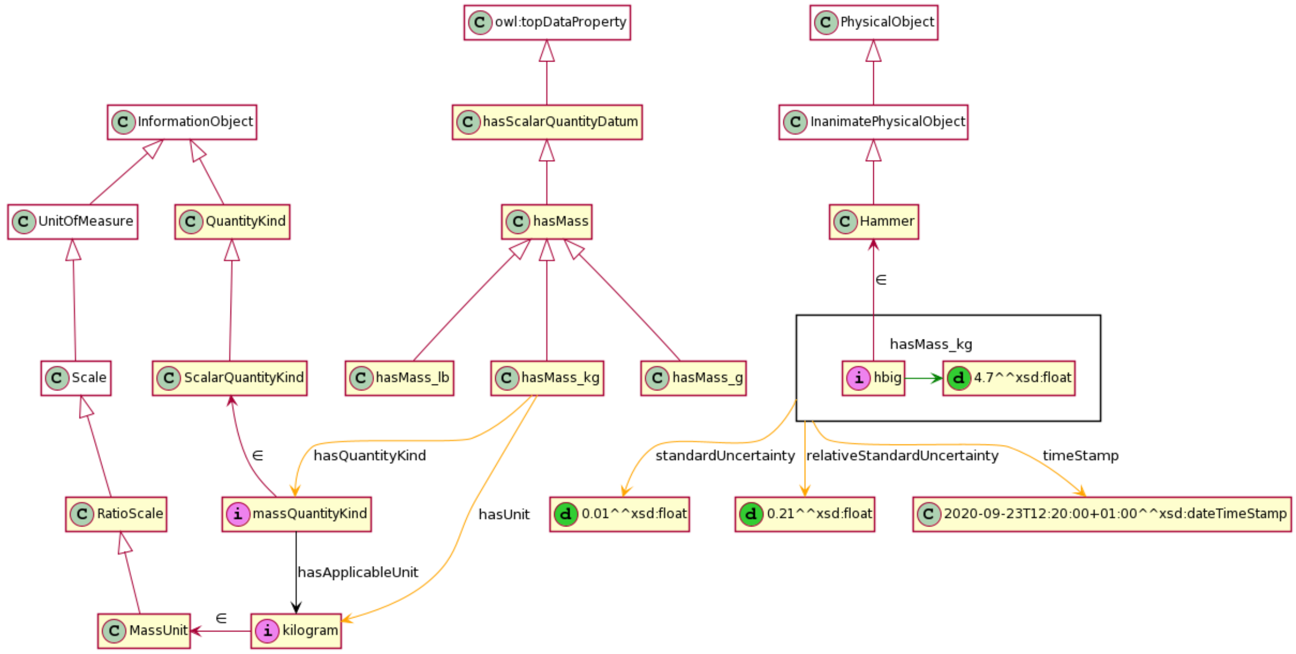


Figure G9.2: Example of a measurement of the mass of a physical object using a short-cut

G.9.6.3 Representation of quantity kinds and units of measurement

This example illustrates some relevant properties and relations of quantity kinds and units of measurement. Notice that the class *PhysicalQuantityKind* describes properties of quantity kinds, whereas the class *PhysicalQuantity* is used to state that a physical object has a physical quantity.

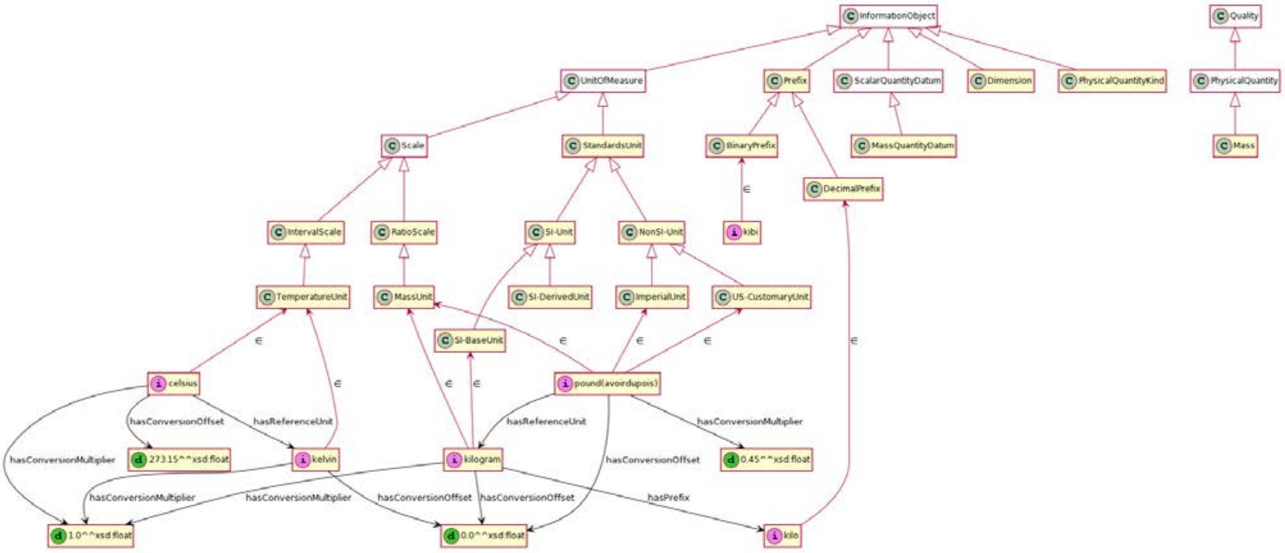


Figure G9.3: Example of quantity kinds and units of measurement

Bibliography

[1] *ISO 15926-2:2004, Industrial automation systems and integration — Integration of life-cycle data for process plants including oil and gas production facilities — Part 2: Data model*

[2] *ISO/TS 15926-12:2018, Industrial automation systems and integration — Integration of life-cycle data for process plants including oil and gas production facilities — Part 12: Life-cycle integration ontology represented in Web Ontology Language (OWL)*

[3] *ISO/IEC 81346-1:2009, Industrial systems, installations and equipment and industrial products — Structuring principles and reference designations — Part 1: Basic rules*

[4] *Optique D11.5, Upper ontology for industrial OBDA applications* (http://www.optique-project.eu/wp-content/uploads/2016/11/D11.5.pdf)

[5] *OWL 2 Web Ontology Language, Direct Semantics (Second Edition).* W3C World Wide Web Consortium Recommendation 11 December 2012 (https://www.w3.org/TR/owl2-direct-semantics/)

[6] *OWL 2 Web Ontology Language, Document Overview (Second Edition).* W3C World Wide Web Consortium Recommendation 11 December 2012 (https://www.w3.org/TR/owl2-overview/)

[7] *OWL 2 Web Ontology Language, Manchester Syntax (Second Edition).* W3C World Wide Web Consortium Working Group Note 11 December 2012 (https://www.w3.org/TR/owl2-manchester-syntax/)

[8] *OWL 2 Web Ontology Language, RDF-Based Semantics (Second Edition).* W3C World Wide Web Consortium Recommendation 11 December 2012 (https://www.w3.org/TR/owl2-rdf-based-semantics/)

[9] *OWL 2 Web Ontology Language, Structural Specification and Functional-Style Syntax (Second Edition).* W3C World Wide Web Consortium Recommendation 11 December 2012 (https://www.w3.org/TR/owl2-syntax/)

[10] *SKOS Simple Knowledge Organization System Reference*. W3C World Wide Web Consortium Recommendation 18 August 2009 (https://www.w3.org/TR/skos-reference/)

[11] F. Baader, D. Calvanese, D. McGuinness, P. Patel-Schneider and D. Nardi (Eds.). *The Description Logic Handbook: Theory, implementation and applications (2nd edition).* Cambridge University press, 2010.

[12] I. Horrocks, O. Kutz, U Sattler. *The even more irresistible SROIQ*. In Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), volume 6, pages 57-67. AAAI Press, 2006.

[13] M. Knapp and F. Hasibether. *Material master data quality*. In Proceedings of the 17th International Conference on Concurrent Enterprising (ICE 2011). IEEE, 2011. https://www.researchgate.net/publication/261208530_Material_master_data_quality

[14] M. G. Skjæveland, A. Gjerver, C. M. Hansen, J.W. Klüwer, M.R. Strand, A. Waaler and P. Ø. Øverli. *Semantic Material Master Data Management at Aibel*. Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018).

[15] F. Martín-Recuerda, D. Walther, S. Eisinger, G. Moore, P. Andersen, P.-O. Opdahl and L. Hella. *Revisiting ontologies of units of measure for harmonising quantity values – a use case.* In Proceedings of the 20th International Semantic Web Conference (ISWC 2020). Springer, Athens (Greece), 2020. Accepted for publication.

[16] H. Rijgersberg, M. Van Assem and J. Top. *Ontology of units of measure and related concepts.* Semantic Web Journal, 4(1): 3-13, 2013.

[17] R. Arp, B. Smith and A.D. Spear. *Building ontologies with basic formal ontology.* MIT Press, 2015.

[18] S. S. Stevens. *On the theory of scales of measurement.* Science, 103 (2684): 677–680, 1946.

[19] *ISO 80000-1:2009, Quantities and units — Part 1: Data model*

[20] S. Kaebisch, T. Kamiya, M. McCool, V. Charpenay, M. Kovatsch. *Web of Things (WoT) Thing Description*. W3C Recommendation, World Wide Web Consortium, October 2020. https://www.w3.org/TR/wot-thing-description/.

[21] A. Haller, K. Janowicz, S.J.D. Cox, D. Le Phuoc, K. Taylor, M. Lefrancois. *Semantic Sensor Network Ontology*. W3C Recommendation, World Wide Web Consortium, October 2017. https://www.w3.org/TR/vocab-ssn/.

[22] M. Koster, D. Anicic, A.S. Thuluva. *Schema.org Extensions for IoT*. Technical Report. https://www.w3.org/WoT/.

[23] M.G. Skjæveland, D.P. Lupp, L.H. Karlsen, H. Forssell. *Practical ontology pattern instantiation, discovery, and maintenance with reasonable ontology templates*. In Proceedings of the 17th International Semantic Web Conference (ISWC 2018),Monterey, CA, USA. volume 11136, pages 477-494. Springer (2018).

[24] https://orcid.org/0000-0002-7167-7321. *Reified Requirements Ontology (Revision: A01A)*. https://w3id.org/requirement-ontology/ontology/core/A01A. [Accessed 23-09-2020].

[25] Standards Norway. *Technical Information Requirements Catalog*. https://tirc.epim.no/. [Accessed 23 09 2020].

[26] G. Casini, T. Meyer, K. Moodley, U. Sattler, and I. Varzinczak. *Introducing defeasibility into OWL ontologies*. In Proceedings of the 14th International Semantic Web Conference (ISWC 2015), volume 9367, pages 409–426, LNCS Springer, October 2015.