Semantic Web-based Open Engineering Platform

STRP NMP2-CT-2005-016972 "SWOP" *

Start date of project: September 1st 2005

Duration: 36 months

D23

The SWOP Semantic Product Modelling Approach

With PMO – The SWOP Product Modelling Ontology

Due date of deliverable: 2006/05/31

Actual preparation date: 2008/04/15

Revision: 8

Circulation: Public (PU)

Partners: TNO, CSTB, VTT, SEMANTIC, PARAGON, USTUTT, CSIRO (international expert)

EU ManuBuild/InPro (harmonization actions)

Authors: Michel Böhms, Peter Bonsma, Peter Willems, Alain Zarli, Marc Bourdeau, Eric Pascual, Graham Storer, Sami Kazi, Matti Hannus, Javier A. García Sedano, Luján Triguero, Harry Tsahalis, Holger Eckstein, Frank Josefiak and Hans Schevers

Doc. Ref. Nº: SWOP_D23_WP2_T2300_TNO_2008-04-15_v12.doc

* Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

COPYRIGHT

DOCUMENT HISTORY

Version	Date	Comment
01	2006-05-30	First draft for comment
02	2006-05-22	Second draft for comment
03	2006-07-13	Final draft with ontology appendices
04	2006-08-28	Final version with extensive typical PMO example added
05	2006-10-05	Revision 1 - Many typos corrected - Comments CSIRO (Hans Schevers) processed - representation.owl extended (ManuBuild input) - appendices with OWL code out (via URLs) - updated GDL example
06	2007-02-22	Revision 2 - discussion on open/closed world assumption added - deletion of 'nature' annotation property - no more explicit disjunct clauses (now implicit) - one .owl file per class - typical example adapted accordingly - guidelines for name spaces & imports added
07	2007-03-16	Revision 3 - appendix 2 changed (parts imported) - changed CWA at subclasses
08	2007-03-23	Revision 4 - appendix 2 changed (parts not imported again) and for owl1.1 name space - decomposition syntax according to latest owl1.1 spec - changed CWA at parts
09	2007-05-31	Revision 5 - Decomposition closures back - Example extended (more typical)
10	2008-02-13	Revision 6 - Various updates based on end-user experience
11	2008-04-01	Revision 7 - Example updated (façade)
12	2008-04-15	Revision 8 - Object properties added/allowed - Typos corrected

ABBREVIATIONS USED

Abbreviation	Explanation [Optional Context]
API	Application Programming Interface
BP	Best Practices [SW]
CASE	Computer Aided Systems Engineering
CSS	Closed Source Software
DB	DataBase
DoW	Description of Work [SWOP]
DTD	Data Type Definition [W3C]
EC	European Commission
EU	European
FP	Framework Programme [EC]
FS	Functional Specification
FUTS	Functional Unite - Technical Solution [GARM]
GARM	General AEC Reference Model [ISO]
HP	Hewlett Packard
ID	Integral Design
IDE	Integrated (software) Development Environment (like Eclipse)
INF	+Infinity
IP	Integrated Project [FP(6)]
ISO	International Standardization Organization
OSS	Open Source Software
OWL	Web Ontology Language [SW]
PMO	Product Modelling Ontology [SWOP]
	A set of reusable OWL ontologies as a layer on top of OWL for product modelling on top of the protocol stack “RDF-RDFS-OWL-PMO”.
RDF	Resource Description Framework [SW]
RDFS	A layer on top of RDF distinguishing between ‘classes’ and ‘individuals’. The basis for the OWL language.
QCR	Qualified Cardinality Restriction [SW]
SE	Systems Engineering
SPFF	STEP Physical File Format [STEP]
STEP	Standard for the Exchange of Product model data [ISO]

SDAI	STEP Data Access Interface [STEP]
XML	eXtensible Mark-up Language [W3C]
XSD	XML Schema Definition language [W3C]
SW	SoftWare or Semantic Web activity[W3C]
SWOP	Semantic Web-based Open engineering Platform [EC]
SWRL	Semantic Web Rule Language [SW]
SWS	Semantic Web Services [W3C]
TBC	TopBraid Composer [Top Quadrant]
UR	User Requirements [SWOP]
W3C	World Wide Web Consortium
WP	Work Package [SWOP]
WWW	World Wide Web [W3C]

EXECUTIVE SUMMARY

This deliverable D2.3 is the outcome of task 2300 "**The SWOP Semantic Product Modelling Approach**"[1] of work package 2 "Ontology-based Modelling" of the IST/NMP SWOP project.

The overall objective of SWOP is to develop a "Semantic Web-based Open engineering Platform"

According to the latest Description of Work (DoW) version (slightly adapted/focussed with respects to the latest insights) this task 2300 recommends and facilitates a Semantic Product Modelling approach for engineering including the choice of technologies and tools to be used in SWOP. The recommendations are:

- Semantic Web technology such as the Ontology Web Language (OWL) as core technology platform in order to be compliant with the upcoming Semantic Web.
- An upper (generic, reusable) ontology for modelling engineering products within the SWOP platform with computer interpretable product structures and product data. Using the upper ontology, product-specific ontologies can be developed aligning them to benefit from the functionalities of the SWOP framework.
- Various choices for syntax forms, Application Programming Interfaces (API), Ontology query languages, modelling conventions and guidelines, persistency mechanisms and editorial tools and utilities etc.

The Product Modelling Ontology (PMO) developed by SWOP has sufficient power to make an end-user product ontology for any parametric/configurable producttype. This ontology models the product from a solution perspective ('what is possible'). The same ontology can be extended with an end-user rule set representing User Requirements (UR) like fixed single values, continuous or discrete ranges (logical OR-sense), including enumerations, regarding all relevant product/component datatype/object properties modelled taking into account their underlying datatypes/ranges.

Adding a fitness function to the ontology and user requirements makes the picture complete for advanced GA-based optimisation.

[1] The original title was: Semantic Product and Production Modelling Approach, but this can be condensed because of our broad definition of 'product' in earlier tasks in SWOP. See for more info deliverable D1.3, the "SWOP Functional Architecture".

TABLE OF CONTENTS

1 INTRODUCTION

In this deliverable we will introduce the SWOP **Semantic Product Modelling Approach**. This approach touches the two bottom 'innovation layers' introduced earlier in SWOP Work Package 1 deliverable D13, the 'SWOP Functional Architecture' (Figure 1). This modelling approach specifies exactly how to deal with the semantic ('smart') product data structures and associated semantic product data[2].

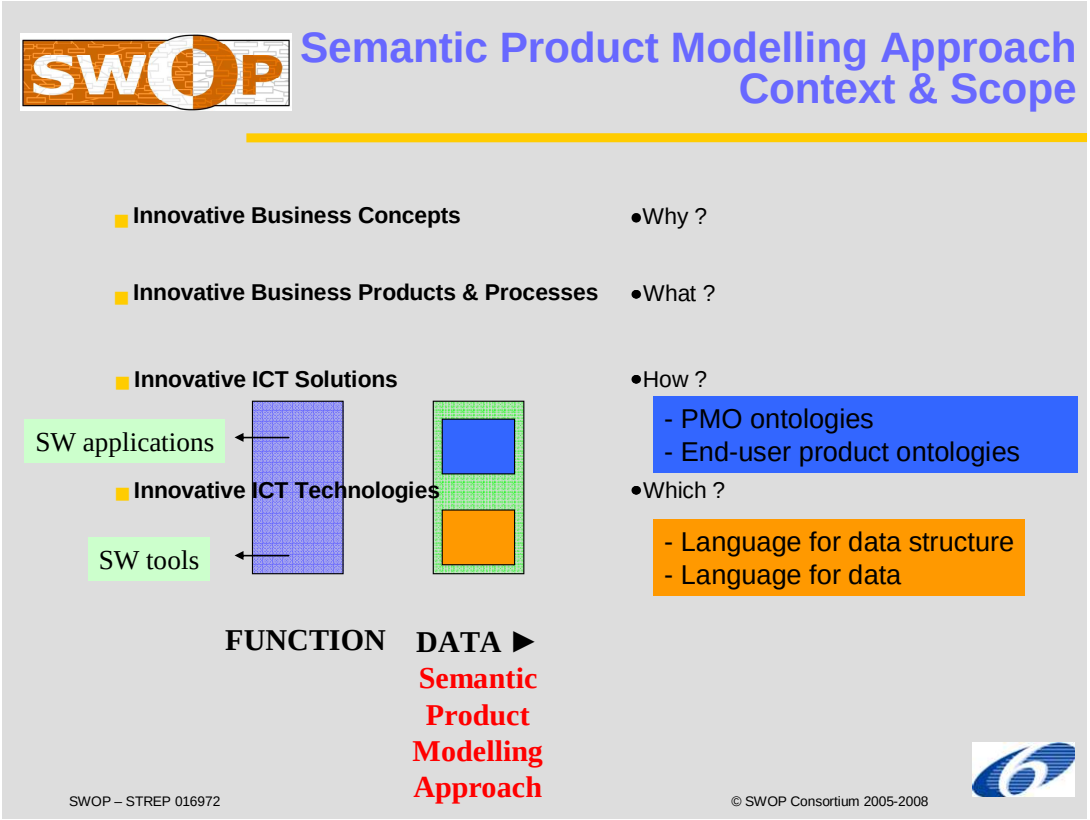


Figure 1 Scope for this deliverable

As can be seen in Figure 1 we divide the topic of this deliverable (the **green box**) in two areas:

1. The **ICT Solutions** side dealing with the data structures and associated data (manipulated by software applications in the **purple box**), and
2. the **ICT Technologies** side that has to cope with the languages (supported by software tools such as editors, rule engines etc. in the **purple box**) applied by the application layer above.

The emerging W3C Semantic Web as the next generation of the Internet is a large international research and development effort which has been perceived as very important and will have a big

[2] There will be similar approaches for the optimisation/configuration functionality aspects but they are not discussed here.

impact. Consequently the EU has developed a ‘Semantic Web action line’ programming their research to create this Semantic Web and to exploit its technologies and benefits. Obviously the SWOP project proposal was directly aligned from the start with this action line. Therefore ICT solutions and technologies are to be sought in the Semantic Web domain. Decisions with respect to the ICT Technologies (“data-side”) have been already taken partly before the project started as reflected by the actual name of the project referencing the Semantic Web.

2 SEMANTIC WEB TECHNOLOGIES

2.1 Introduction

In SWOP deliverable D13 [SWOP D13] it was explained that any (data) modelling approach basically decides on two main languages: (1) one for the information structure and (2) one for the data/content according to that structure. There are many approaches around with different levels of expressive power, ease of use, critical mass, stability, commercial viability etc. Without going in too much detail we mention here three main current approaches (figure 2) that are described below.

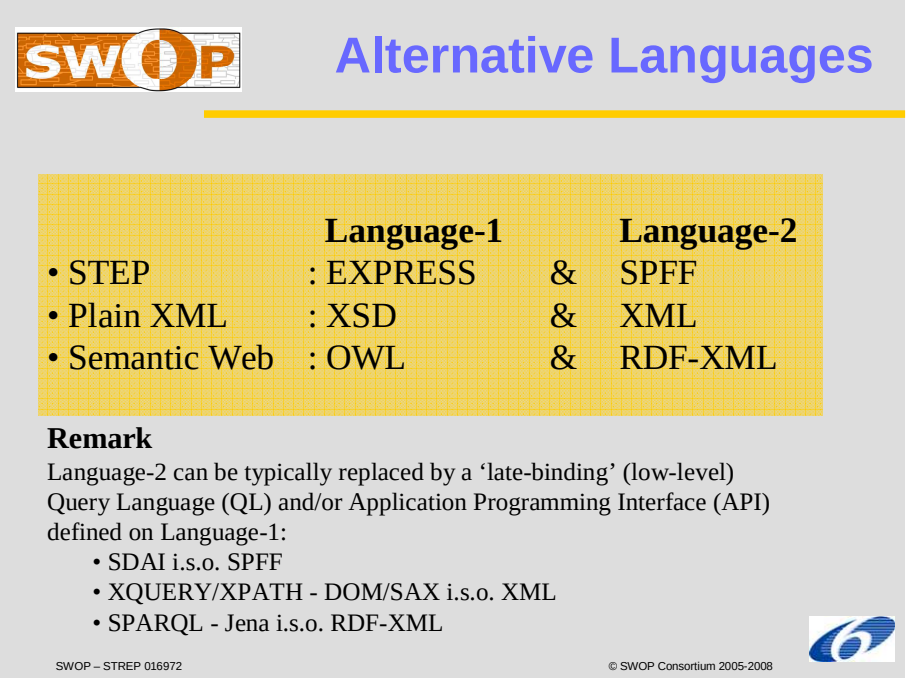


Figure 2 Alternative ICT Infrastructures

2.2 ISO STEP Technologies

Since the early days of product modelling, people are using STEP technologies from the ISO STEP standard (Standard for the Exchange of Product model data). The so-called schemas are expressed in the EXPRESS language, the actual data/content according to these schemas in STEP Physical File Format (SPFF) files. Syntax independency can be obtained by using the EXPRESS-level Application Programming Interface (API) here referred to STEP Data Access Interface (SDAI). The STEP community has standardized many different schemas for different engineering domains. The uptake of STEP schemas varies heavily. STEP is not directly related to Semantic Web technology yet many overlaps can be found. Currently research and development efforts are being carried out to bring STEP technology closer to the emerging Semantic Web technology.

2.3 W3C Plain XML Technologies

Gradually this approach was/is replaced by more web-oriented languages like in the form “Plain XML” where data structures are modelled in DTD’s (Data Type Definitions) or the more powerful XML Schema Definitions (XSD) and the associated content in plain XML (eXtensible Mark-up Language).

2.4 W3C Semantic Web Technologies

A smarter, more ‘semantic’ version of “Plain XML” adding more modelling power and more transparency towards the syntax forms is the ‘Semantic Web’ approach using the Web Ontology Language (OWL) for the structures and RDF-XML (XML specialisation by the Resource Description Framework) for the data/content.

2.5 Conclusions

For SWOP our primary choice is for the “Semantic Web” as a generic, powerful and well-used and supported (by software tools) modelling approach. Both RDF-XML and OWL are stable international open standards that are backed by a large community of end-users and software tool vendors. However, since OWL is a general data modelling language, it lacks built-in support for typical product modelling aspects. We will therefore specify in the next chapters how we will put a layer on top of OWL especially for the modelling of products. We will therefore follow the requirements as formulated in task 2200. Note that we will not change in any way the technologies themselves but only develop a generic, reusable, set of **Product Modelling Ontologies** (PMO) that can be imported and used by the end-user product ontologies (for houses, machine tools, processes etc.). Together with the right modelling guidelines this set of generic product modelling ontologies form **The SWOP Semantic Product Modelling Approach.** In the next figure we show how the different languages/ontologies are related.

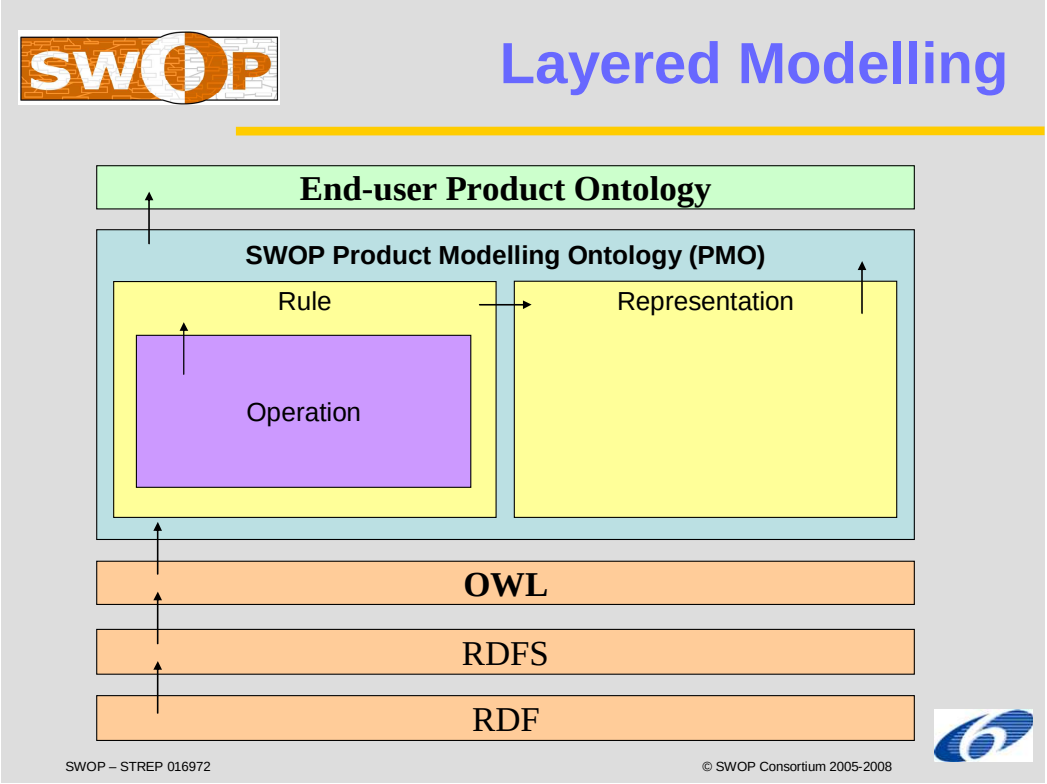


Figure 3 PMO on top of OWL

3 MODELLING WITH RDF/OWL

3.1 Introduction to the Resource Description Framework (RDF)

The basic specification underlying the semantic web technologies is the Resource Description Framework (RDF). This a very well defined basic building block. Many researchers have made it a logically and mathematically sound approach. In its essence RDF is a semantic network graphically equivalent to a so-called 'directed graph'. This graph is fairly simple: there are nodes and directed edges between these nodes. Two nodes and one edge from one node to the other makes a 'triple' and a directed graph is nothing more than a set of these triples (triples become connected via shared nodes).

The input node is called the 'Subject', the output node the 'Object' and the edge the 'Predicate'. This basic construct is used to model almost everything for both information content/data and structure! Changing data or structure (or the links between tem) finally comes down to creating or deleting triples as an atomic action. Said otherwise: the sets of triples can be regarded as the optimally normalized relational model. An example (let's call it MyOntology.owl) makes things more clear. Suppose we have Cars and Persons and I am a Person having a Car etc.…the following eight triples describe everything there is to know (Figure 4).

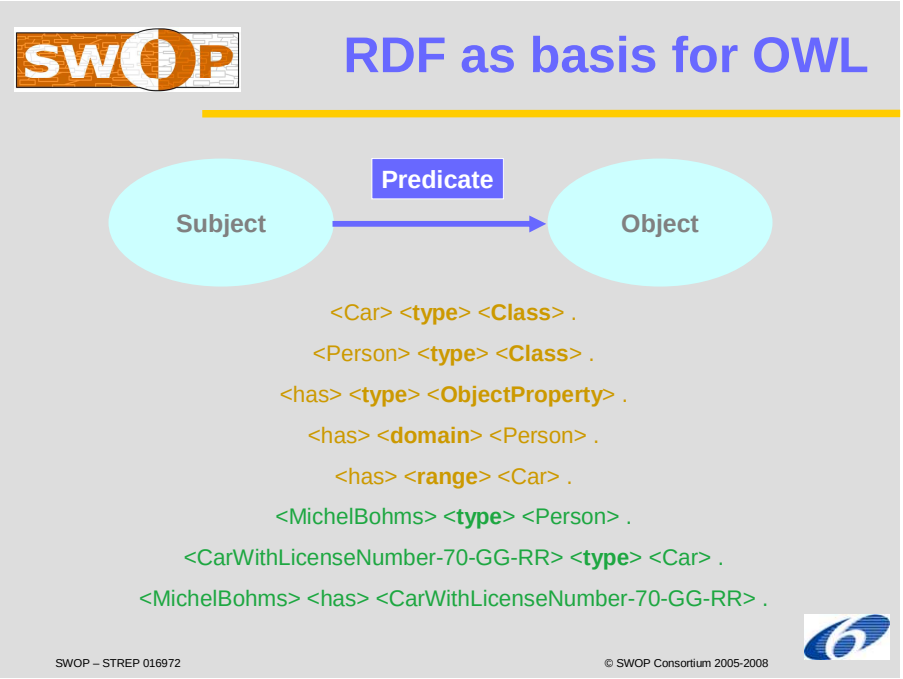


Figure 4 RDF as solid basis for OWL

The bold items are special because they are predefined in this case by RDF ('type'), RDFS ('domain' and 'range') or even OWL ('Class'). All details including name spaces / URIs are deleted for overview. The essential message here is that with RDF we can describe any web content/structure/meta-structure/etc. (when using 'type' in nested ways) in the most simple way. With an RDFS/OWL-hat on we distinguish the structure from the data.

The triples can be represented in several, equivalent, ways. The above example used N-TRIPLE. Other forms are N3 (non XML-based), or RDF/XML or RDF/XML Abbrev(iated). The latter is a

more compact form sacrificing however determinism. In RDF/XML Abbrev. there exist more than one RDF-XML file that differ more then just in the irrelevant order of triples. They are representing the same data but the final form is dependent on the order of parsing the file.

3.2 SWOP "World Assumption"

One of the most basic decisions in any modelling endeavour is the choice of the used 'world assumption': an Open World Assumption (OWA) or a Closed World Assumption (CWA). Without going into too many formal details, we can state the following:

- In case of an Open World Assumption, everything not said (stated, specified, modelled, …) is assumed to be "unknown", so it can still be true or false. In this case the model, ontology, schema etc. never forms a 'closed world': the environment is always taken into account. Nothing is assumed false when unknown because you never now someone else outside the current scope (say a modeller in Timbuktu) might state something is true or false after all.
- In case of a Closed World Assumption, everything not said (stated, specified, modelled, …) is assumed to be “false”. In this case the model, ontology, schema etc. forms a 'closed world' on itself where the outside world is kind of ignored. Everything not said in this scope is per definition not true aka false.

Most traditional modelling approaches (like in e.g. ISO STEP's EXPRESS) apply a closed world assumption whereas more modern, especially web-based, approaches start from an open world assumption. In the SWOP project we make use of semantic web technologies which typically assume an open world:

- A defined property in OWL is in principle a property of any class. So we have to add so-called ‘domain clauses’ to limit the relevance of properties to certain classes. In SWOP we assume only one class (having this property). So each user-defined property has a domain clause referencing exactly one domain class.
- Subclasses are in general non-complete (their union is not spanning the whole superclass) and overlapping (or not disjunct). In SWOP however we decided for simplicity to only allow complete and disjunct subclasses. For flexibility and simplicity we decided not to model these constraints explicitly. We assume implicitly completeness and disjunctness for all subclasses for each superclass at 'configuration-time' by our software. At design-time this means one can more easily add a certain subclass when desired without changing too many rules for the relevant superclasses[3]. We do not define a default subclass. For the PMO Configurator, the first specialisation sub class encountered (in the OWL file) is shown.
- We (pre)define in PMO a decomposition object property that has the generic 'Product' class as both domain and range class. Hence, any end-user ontology class can be part of any other end-user class. For our product modelling we have to be more precise. That's why we use closures and QCRs (Qualified Cardinality Constraints) to limit the decomposition possibilities. With 'closures' we state what classes of parts are possible/relevant for a certain whole class. For those possible ones the default min and max cardinalities (as for all properties) apply, being 0 for the min cardinality and +INF(inity) for the max cardinality. We use QCRs to further constrain

[3] It's also more flexible in case of adding ‘on-the-fly’ variant subclasses at configuration-time in advanced future scenarios.

these cardinalities as required. For products that have no further decomposition (or "atoms") we define that the max cardinality of the decomposition property is zero (non-qualified, so for all possible qualifiers). For decomposition however we need some “default amounts” too. Adding a kind of annotation to a (qualified) restriction (min/max cardinalities) was considered too complex. Therefore, the default here is the same as the min cardinality value. Rules will affect these amounts of things (derivations or assertions will be taken into account via cardinality modifications respectivily warnings). In the SWOP PMO Configurator GUI by TNO there will be a field for each qualified hasPart_directly object property indicating min and max cardinalities where an end-user can specify an actual amount in between.

3.3 The OWL Container

Each OWL file starts with the indication that it is actually an XML file:

Example

```
<?xml version="1.0"?>
```

Next this file is made more specific by stating that it is an RDF file. At the same time we specify all URLs for the definitions at hand (defined in this .owl file) and all resources used from the web.

Example

```
<rdf:RDF
    xmlns="http:// www.swop-project.eu /ontologies/facade.owl#"
    xml:base="http:// www.swop-project.eu /ontologies/facade.owl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:owl11="http://www.w3.org/2006/12/owl11#"
    xmlns:product="http://www.swop-project.eu/pmo/product.owl#"
        MORE…….
</rdf:RDF>
```

In the example we reuse an ontology that is available on the web in a certain file: http://www.swop-project.eu/ontologies/pmo/product.owl. In this file we can shortcut this long name with just “’product” as indicated. The top two lines refer to THIS file: its place where it can be found on the web and its own base name so that we can just provide a class name etc. in this file without mentioning the whole URL all the time (kind of local 'short name').

The container concept for an ontology as a grouping of classes, properties, individuals, conditions etc. is an Ontology. This is typically the first statement after the xmlns (XML name space) clauses within the rdf container. Inside we find the imported ontologies for all the things that are not just referenced but actually used in some way.

Example

```
<owl:Ontology rdf:about="">
  <owl:imports rdf:resource="http://www.swop-project.eu /ontologies/pmo/product.owl"/>
</owl:Ontology>
```

What follows are the classes and properties that will be discussed in the next sections.

3.4 Classes

Classes form the most basic meta concept in OWL. They are used to model the primary concepts with 'members' as their 'extension'. These classes are not "object-oriented classes" with methods but are reflecting sets of members defined in some logical way (like by using predicates or via enumeration). The members of the classes are called 'individuals' in OWL.

It is important to note that we have to be very clear on the interpretation of classes and individuals: individuals are occurrences that exist in reality (or *could* exist if they don't exist *yet*): one can (or could) point at them, everything else is a class. This means that a catalogue item is a class of which you can order three individuals. In product modelling we typically prefer a three-level approach ("generic-specific-occurrence") involving, beyond a generic class and a particular individual, some 'variant' in between (partially or fully specified) that can be placed in space and/or time several times. In principle there are several ways of mapping the required three levels to the two levels offered by classes and individuals (Figure 5).

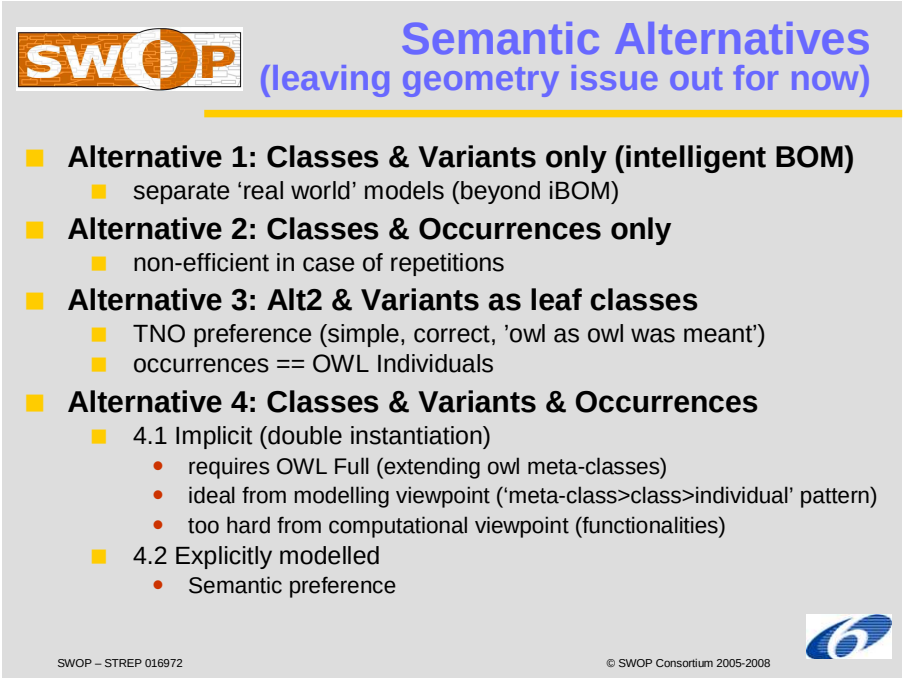


Figure 5 Alternative 3-to-2 level mappings

We have chosen for "Alternative 3" where variants are modelled as sub-classes. This is the most natural way since a variant indeed denotes a set of occurrences in the end that comply to the variant structure (just having a different placement in space or time). We can further distinguish predefined "standard" variants and on-the-fly defined 'end-user' variants.

Example

```
<owl:Class rdf:ID="Facade"/>
```

Note that we use the OWL language construct 'Class' for defining this specific class. The 'Class' class is a meta-class that can be instantiated in user-defined classes. This fact clearly shows the layered modelling approach in OWL.

When we directly instantiate this facade concept we get:

```
<Facade rdf:ID="Facade_01"/>
```

This construct defines an individual facade with the name “Facade_01” that is of type “Facade”. Because we did not define any relevant properties; there are no more details on this individual level. To make things more interesting (and useful) let’s see how we define properties in OWL in the next section.

3.5 Properties

Properties in OWL are a bit special for two reasons:

- They are so-called ‘first class’ concepts which means that they are on the same level as the Class concept. In many other modelling approaches, properties (attributes, slots, …) are secondary concepts: fìrst there are entities, classes, …. and thèn there are properties which are typically directly associated to such an entity, class, … Not so in OWL: classes and properties are considered equally important and modelled independently first and then interrelated where relevant.
- Properties in OWL do not just denote simple (datatype) properties having a 'value' according to some data type[4] like height, width etc. but they also cover relationships between classes. Said otherwise: if classes and datatypes are the nodes of an ontological network, the properties represent <u>all</u> the edges between them.

Let’s first address the more simple Datatype Properties in OWL. For each property a domain and a range is specified. In the example below the domain is a Window so it means that only a Window can have a windowWidth property. If nothing is specified, in principle, all classes can have this property. Next a range is specified, here being the float datatype reused from the XSD name space.

```
<owl:DatatypeProperty rdf:ID="windowWidth">
        <rdfs:domain rdf:resource="#Window"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
</owl:DatatypeProperty>
```

Besides floats, its also possible to have integers, strings, booleans or more specific ones involving times and dates. In case of strings we can enter enumerations: sets of allowed string values like for “colour”:

Example

[4] Integer, float, string, boolean etc.

```
<rdfs:range>
    <owl:DataRange>
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string">red</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first
                rdf:datatype="http://www.w3.org/2001/XMLSchema#string">blue</rdf:first>
              <rdf:rest>
                <rdf:List>
                  <rdf:first
                    rdf:datatype="http://www.w3.org/2001/XMLSchema#string">green</rdf:first>
                  <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
                </rdf:List>
              </rdf:rest>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
```

The above code also shows an example of not too efficient modelling using OWL[5]. Let's now have a look how to model interrelationships between classes. In fact we're talking properties now that have individuals both as domain and as range. Suppose we have two classes: a Product and a BoundingBox:

```
<owl:Class rdf:about="#Product"/>
<owl:Class rdf:about="#BoundingBox"/>
```

We then define:

[5] Hopefully OWL1.1 or an even later version will allow for more direct modelling of allowed values not involving the list structure above.

Example

```
<owl:ObjectProperty rdf:ID="boundingBox">
  <rdfs:range rdf:resource="#BoundingBox"/>
  <rdfs:domain rdf:resource="#Product"/>
</owl:ObjectProperty>
```

(Here we see some interesting other feature of OWL: the order of type, range, domain etc. clauses is completely irrelevant.)

In SWOP end-user product ontologies, both datatype and object properties will be defined. (the generic PMO defined object properties like hasPart_directly should of course be reused).

Furthermore, all these properties will have exactly one domain class defined. Cardinalities are controlled by min and max cardinalities:

- always 1..1 for datatype properties ,and

- default (being 0 respectively +INF(inity)) for object properties

With the notions of classes, individuals and properties we introduced so far all main 'archetypes' of OWL modelling. In a sense, all further modelling details are forms of what OWL calls 'condition modelling'. We will first consider a very important type of condition that got its own language element in OWL: subclassing enabling class specialisation to be modelled.

3.6 SubClasses

Remembering that all classes represent classes of individuals it follows quite logically to be able to define subsets of individuals satisfying certain conditions. If A1 is a subclass of A it means that all individuals of A1 are also individuals of A. An example is shown below:

Example

```
<owl:Class rdf:ID="Window">
   <rdfs:subClassOf
          rdf:resource="http://www.swop-project.eu/ontologies/pmo/product.owl#Product"/>
</owl:Class>
```

The subClassOf property is already predefined in the RDFS layer[6] of OWL. The same way another subclass of Product was defined: the Facade.

At design-time we assume a full open world assumption. However at configuration-time we will assume leaf classes having no further specialisation: if read in memory there is no known further specialisation. We will also always consider (implicitly) all same-level subclasses (sharing the same parent super-class) being disjunct, complete and allow only one super-class for each subclass.

[6] The layer on top of RDF that actually starts differentiating between the meta-concepts 'classes' and 'individuals'

Finally we will always assume a choice of subclass to be made when configuring. This way we get not too complex specialisation tree structures and not too much overhead.

3.7 Cardinalities

One can indicate for each property min and max cardinalities, in the context of a class (in the SWOP situation: in the context of its one domain class).

Example

```
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:minCardinality
        rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:minCardinality>
    <owl:onProperty …some property …
  </owl:Restriction>
</rdfs:subClassOf>
```

This example shows how a new, anonymous, superclass is defined representing 'all things having min cardinality 1' (there should be at least one…) for a certain property. By making the class of interest a subClass of this class we actually express the condition that should hold.

Instead of specifying a max cardinality of 1 of a property for multiple classes, one can specify in OWL that a property is functional (i.e. max cardinality is always 1 independent of class having this property). We will however **NOT** do this and just specify the max cardinality being 1 if relevant.

Another construct in OWL allows to combine min and max cardinality if they are the same (just 'cardinality'). We will again **NOT** use this construct and always use separate min/max cardinality restrictions when relevant.

For now PMO only supports min cardinality=1 and max cardinality=1 for user-defined datatype properties. Classes can have 'more than max=1' cardinality properties used in for example rules, but these are not of interest (they are temporary) in the final state and have the special max cardinality=0.

4 SEMANTIC PRODUCT MODELLING WITH PMO

4.1 Introduction

As made clear in the previous section, OWL has a lot of power to model 'anything' including products. Still, there are some missing features, some of which are generic and foreseen in the upcoming OWL update ([OWL1.1]) and some which are beyond the scope defined for this language. The latter are typically addressed as 'modelling patterns' to be reused as a kind of best practices (see also [SW BP]). Just as with implementation-oriented 'software patterns' we can define 'product modelling patterns' that are useful in many situations but that are not directly supported by the language (OWL). This can be regarded as a kind of layer in between the language (OWL) and the end-user product ontologies. We are talking small, reusable ontology parts like for modelling 'decomposition', 'units', 'default values', etc. Preferably these patterns are reused from a reliable, authoritative source. Fortunately W3C formed a "Semantic Web Best Practices and Deployment Working Group" for identifying, developing and promoting such patterns. Unfortunately they don't provide all patterns needed for SWOP and some in a way that are not directly suitable.

In this section we will define a minimal set of SWOP extensions needed to fulfil the SWOP product modelling requirements identified. In the end, all these constructs are collected in some small reusable OWL ontologies that have to be imported by any end-user ontology:

- product.owl (the toplevel PMO ontology)
- representation.owl
- rule.owl
- operation.owl

Collectively we refer to these ontologies as the SWOP Product Modelling Ontology (PMO).

4.2 Qualified Cardinality Restrictions (QCRs)

With normal (unqualified) cardinality restrictions one can say something about the amount of range individuals for a specific property in the context of a specific class. In case of data type properties this is typically fine. In case of object properties there could be alternative range classes relevant. If the range its type is a superclass with say 5 subclasses; we can limit the amount to 10 individuals of type superclass but we're not able to specify in more detail with respect to which type of subclass. QCRs add exactly this information. Instead of saying a Pizza had max 5 layers we can now express that it should have max 2 cheese layers, exactly one meat layer and one or two sauce layers. This added expressiveness is crucial in modelling product decomposition as will be explained in the next section.

4.3 Product Decomposition

"Decomposition" is one of the most missed OWL built-in mechanisms. One of the standard OWL abstraction mechanisms is "specialisation" using subclassing. For each class identified we can specify its superclass via a built-in "rdfs:subClassOf" property. This way we can model whole hierarchies of classes that are more or less generic/specific. Now when specialisation corresponds to the 'logical <u>or</u>'-relationship: a vehicle is a car or a boat <u>or</u> a plane; decomposition corresponds to a

complementary 'logical and'-relationship: a car consists of an engine, a chassis and the bodywork. Decomposition becomes a kind of orthogonal hierarchy with respect to the specialisation hierarchy. To some, decomposition is seen as even more important than specialisation since it seems to stand 'closer to reality': we 'think up' superclasses but we 'see' aggregates/composites. At class level we are talking typical decomposition and at individual level we have the actual decomposition (by some referred to as "object trees").

Anyway, many people want to express these decomposition structures in a suitable way and W3C has proposed a way to do it. For this purpose it defined two transitive object properties being 'hasPart' and 'partOf', being each others inverses. Via these properties a part is not only related to its whole but also to the whole of which the first whole is a part etc. Often you just want to model the first direct part-whole relationship so they introduced two subtypes of these properties too that are non-transitive: 'hasPart_directly' and 'partOf_directly', again being each others inverses. We can now for each class specify that this property has some values towards another class. An example will clarify.

Suppose we have a House which consists of a Kitchen, a LivingRoom and some BedRooms. We define the relevant classes in OWL and use the partOf relation in a "someValuesFrom" existential relationship condition obtaining for LivingRoom the following specification (in RDF/XML Abbreviated syntax and visualised using OWLViz):

Example

```
<owl:Class rdf:ID="LivingRoom">
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:someValuesFrom rdf:resource="#House"/>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#partOf_directly"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This is the common way to do it according to the best practise, in standard OWL 1.0.

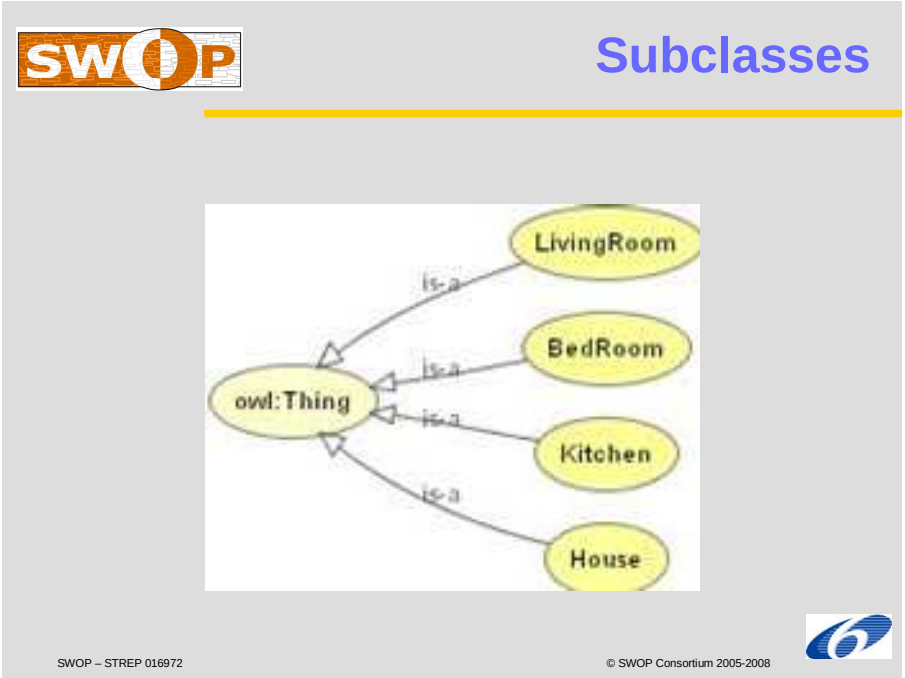


Figure 6 SubClassOf

A similar specification is relevant for the BedRooms and the Kitchen. In Protégé we can now browse the part-of hierachy that can be generated from the knowledge of these 'existential relationships':

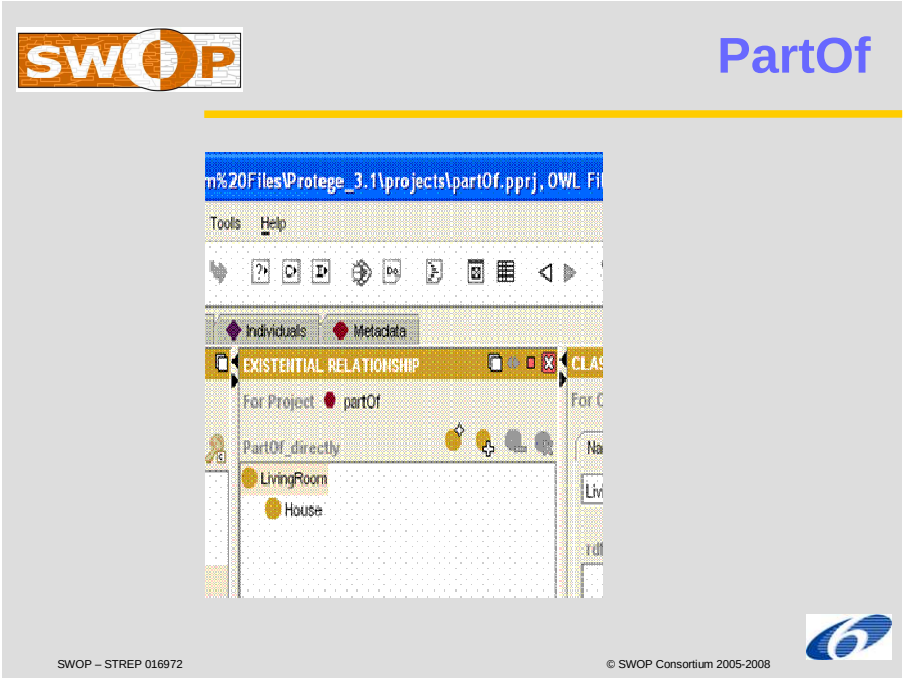


Figure 7 PartOf

Although for flexibility the guideline is to use 'partOf' it is easy to see that 'hasPart' gives a nicer tree as visualisation being more coherent with the modelling of specialisation (the higher the more

'generic' respectively 'global'). So if we now for the moment ignore the potential scaling issues and add hasPart and hasPart_directly (including the inverses) we obtain:

Example

```
<owl:Class rdf:ID="House">
   <rdfs:subClassOf>
     <owl:Restriction>
       <owl:someValuesFrom>
         <owl:Class rdf:ID="Kitchen"/>
       </owl:someValuesFrom>
       <owl:onProperty>
         <owl:ObjectProperty rdf:ID="hasPart_directly"/>
       </owl:onProperty>
…etc.
</owl:Class>
```

Note that in this case (modelling both partOf <u>and</u> hasPart) we have to add the someValuesFrom relation 'the other way round' too because:

A partOf someValuesFrom(B)

Does <u>not</u> imply:

B hasPart someValuesFrom(A)

Now we can generate the hierarchy top-down:

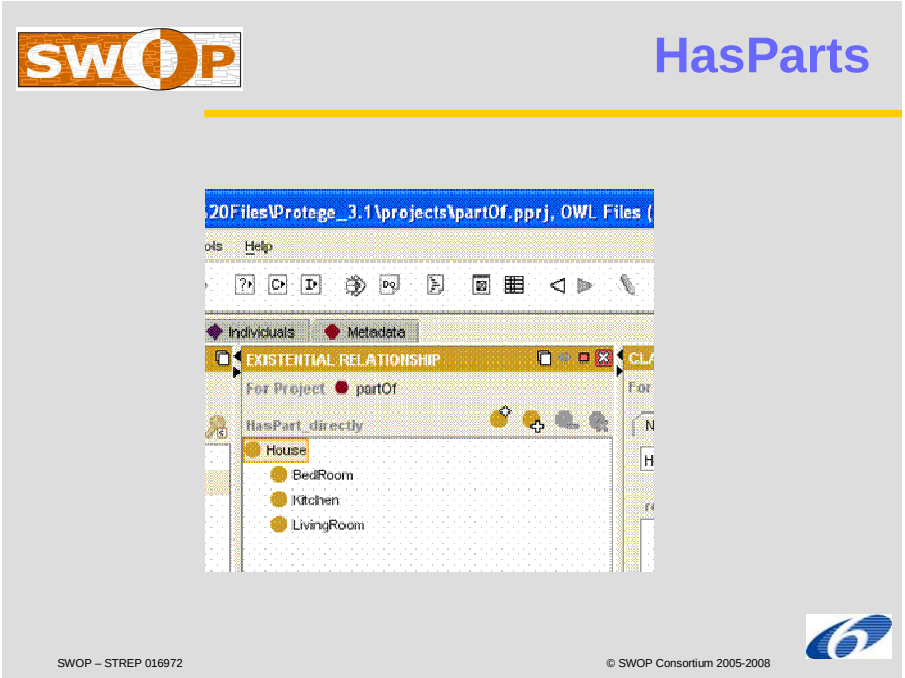


Figure 8 HasPart as inverse of PartOf

We run with this example in two well known issues:

- Each part always has at least one whole (some == at least one); it has no life of its own but always in the context of a whole, and
- The maximum cardinality cannot be controlled (we cannot state there is exactly one whole or ten, or less than fourteen etc.). The same is true in case of hasPart relationships if these are used (i.e. like 'a house has exactly three bedrooms' or 'maximum five bedrooms').

We can conclude that using "someValuesFrom" is not the optimal OWL mechanism. We expect much more from the yet-to-be-formally-introduced "Qualified Cardinality Restrictions (QCRs)" which give us the power to control both min and max cardinalities for both directions (partOf and hasPart) in a more precise way indicating the valid target class amounts.

This new mechanism is expected to be present in the upcoming OWL1.1 update. We will now show how this powerful approach, as chosen for SWOP/PMO, works. In SWOP we will only use the hasPart_directly variant. Instead of the someValuesFrom condition we get:

Example

```
<owl:Class rdf:ID="House">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onClass>
        <owl:Class rdf:ID="BedRoom"/>
      </owl:onClass>
      <owl:minCardinality
```

```
        rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:minCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasPart_directly"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
        …
</owl:Class>
```

Here the someValuesFrom is replaced by a min cardinality constraint being 1 and no max cardinality constraint (default being '+infinity"). So a House has 1 or more BedRooms. Note that we have to specify clearly the underlying type now being BedRoom via the new (OWL1.1) 'onClass' tag. Visualized in Protégé:

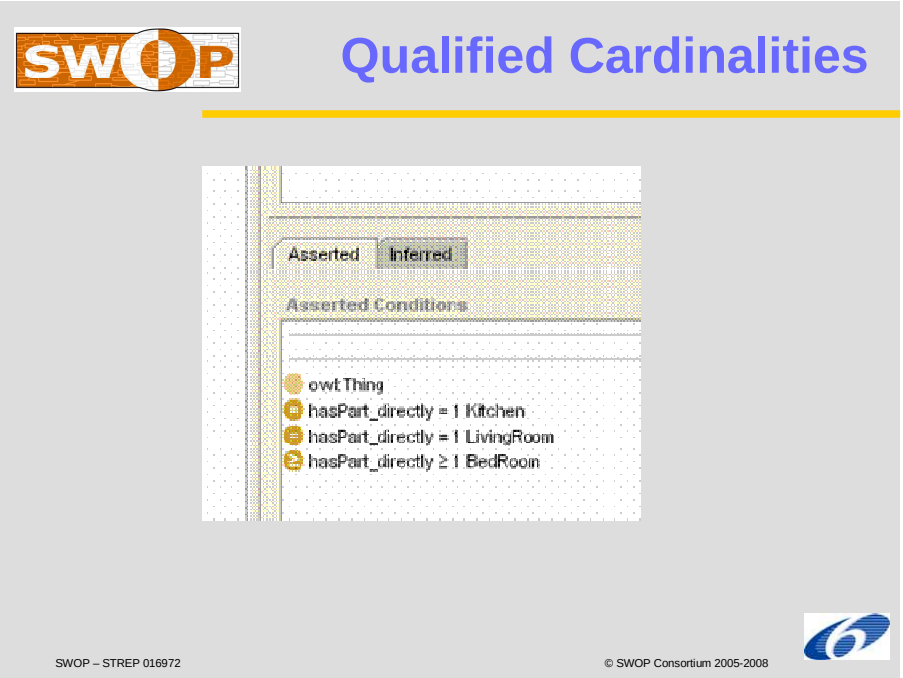


Figure 9 Qualified Cardinalities controling decomposition

Because of our Open World Assumption we now finally have to state that all parts of a House are Kitchens or LivingRooms or BedRooms. Therefore we add the constraint:

```
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:allValuesFrom>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="Kitchen"/>
          <owl:Class rdf:about="LivingRoom"/>
          <owl:Class rdf:about="BedRoom"/>
        </owl:unionOf>
```

```
        </owl:Class>
      </owl:allValuesFrom>
      <owl:onProperty rdf:resource="http://www.swop-
        project.eu/ontologies/pmo/product.owl#hasPart_directly"/>
      </owl:Restriction>
    </rdfs:subClassOf>
```

Finally we have to add for all atomic products (that are products having no parts):

```
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:maxCardinality
       rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
       >0</owl:maxCardinality>
    <owl:onProperty rdf:resource="http://www.swop-
      project.eu/ontologies/pmo/product.owl#hasPart_directly"/>
  </owl:Restriction>
</rdfs:subClassOf>
```

(more explanation of cardinality modelling follows later)

Why so much fuss about decomposition? Well, we think its one of the most important abstraction mechanisms for modelling objects around! With a clear best practice in the OWL context and supported by tools, it can be regarded as THE approach for modelling 'class & object trees' complementing the built-in specialisation mechanism of OWL itself.

4.4 Meta-properties: Units and Default Values

Being able to model that "the weight of this machine is 14" does not say much. We have to indicate the unit of measurement for the value "14". In a sense, we have to put this value in the right context. There are many ways to add the fact that we mean "14 kg". Some initiatives put it in the value: "14 kg" instead of just "14". Other initiatives define a full blown unit ontology which is then related to the property and its value. In SWOP we have chosen a lightweight solution using "annotations". Annotations are OWL's way of escaping pure OWL, a means to extend OWL.

Formally annotations are just treated as meta-information not necessarily processed by OWL parsers but all signs are that they will get more importance in future versions of OWL (since meta-modelling is seen as key to more flexible modelling in general).

That's why we decided in SWOP to use this feature for meta-data on properties, not only units but also for default values. For the current handling/visualisation of ontologies (when no individuals are available yet) we use the "defaultValue" information to decide actual values for user-defined datatype properties.

Example

```
<owl:DatatypeProperty rdf:ID="facadeWidth">
  <product:unit rdf:datatype="http://www.w3.org/2001/XMLSchema#string">m</product:unit>
  <product:defaultValue rdf:datatype="http://www.w3.org/2001/XMLSchema#anySimpleType>5.5</product:defaultValue>
</owl:DatatypeProperty>
```

This way the units are modelled as meta-data at class level without the need to repeat them on individual level.

4.5 Representation

4.5.1 Representation ontology in PMO

SWOP is primarily concerned with semantic information. Still there is a need to also handle non-semantic, representational information linked to or derived from the pure semantic information. Typically this information deals with shape aspects where implicit semantic information like height, depth and width are represented as parametric cubes, BREPs or even presented in nice (pixel-based) pictures. Unfortunately there is not one standard for expressing this information in a generic way. IAI IFC is a way of doing for the construction industry (especially for modelling 'buildings'). Another problem is the fact that such standards differ greatly in expressive power (even in this type of modelling we can distinguish different levels of smartness ranging from fully parametric explicit shape objects to stupid pixels).

For this reason we decided to define a high level representation ontology ("representation.owl"). This ontology is imported by the product modelling ontology. This representation ontology is used as generic linking pin towards domain-specific representation schemes such as IAI IFC for construction. Some notions with respect to this representation.owl ontology:

- It contains a class Color that is used to define a color, subclasses AmbientColor, DiffuseColor, EmissiveColor and SpecularColor are available to define the color more detailed. Each Color has a Red, Green and Blue component together with an alpha component
- It contains a class Material that is used to represent geometry objects material information, this includes color using the subclasses of class Color, but also shininess. A geometry object can never contain color directly, only through its material information
- It contains the class Representation. This class is the superclass of class Geometry and class Topology
- It contains a class Geometry that is used for all geometry concepts. The concepts that define a real position inside the 3D area, i.e. TransformationMatrix and Vector
- It contains a class TransformationMatrix that is used to represent relative position, orientation and scaling of objects
- The class Vector enables the possibility to define a point somewhere in the 3D area driven by a formula
- It contains a class Topology that is used for all topology concepts. The concepts that define a conceptual description of a geometrical object. Together with the possiblity to link the conceptual description to geometry it enables the user to define new geometrical concepts. We find 4 topological building blocks: Vertex (0D), Edge (1D), Face (2D) and Solid (3D)
- The class Vertex makes it possible to define the concept of a 0D topology, i.e. a point in space. It can be link to geometrical representation Vertex that exactly locates this point based on formulas and/or fixed values
- The class Edge makes it possible to define the concept of a 1D topology, i.e. a line in space. It can be link to geometrical representation Vertex that defines this line in 3D space based on formulas and/or fixed values, it can alternatively be linked to exactly 2 Vertexes

- The class Face makes it possible to define the concept of a 2D topology, i.e. an area in space. It can be link to geometrical representation Vertex that defines this area in 3D space based on formulas and/or fixed values, it can alternatively be linked to a set of at least one Edges
- The class Solid makes it possible to define the concept of a 3D topology, i.e. a volume in space. It can be link to geometrical representation Vertex that defines this line in 3D space based on formulas and/or fixed values, it can alternatively be linked to a set of at least one Faces
- The class ConstructiveSolidGeometry exists of two derivation rules. Each rule will define some kind of representation, the type of the ConstructiveSolidGeometry will define how both concepts interact with eachother, see also <http://en.wikipedia.org/wiki/Constructive_solid_geometry>
- The class Volume is an abstract base class, subclasses can be created that define topo-logical concepts. These concepts can then be used by other PMO files.

Representation contains in this way complex and very abstract ideas. The reason for making this decision is threefold:

- We want to be able to define simple 0D, 1D, 2D and 3D concepts in very short .
- We want to store geometry concepts without loss of conceptual ideas
- The level of detail/genericity of representation.owl is now similar to de level of detail of rule.owl, product.owl and operation.owl

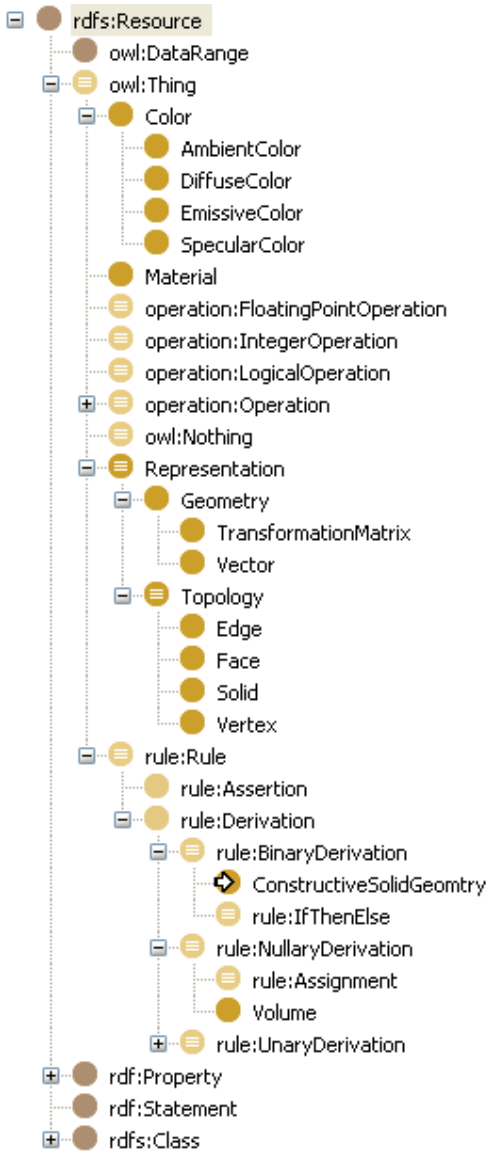


Figure 10 Representation classes in PMO

4.5.2 The example of a Cuboid.

There are 4 ways to model this:

- a Cuboid modelled as a solid existing of faces existing of edges existing of vertexes
- as previous, but now edges modelled as formulas instead of vertexes
- as previous, but now faces modelled as formulas instead of edges
- as previous, but now solids existing as formulas instead of faces

For all 4 solutions we create a new class Cuboid, this Cuboid has three parameters length, height and depth.

Note: If we look at the Cuboid, then we can divide it into a fixed amount of Vertexes, Edges, Faces and Solids. All examples up till the creation of this document will always needs this restiction. This means that solids can only be created when they are represented by formulas or by a fixed amount of Vertexes, Edges, Faces and Solids, this way all typical Solids like Sphere, Cylinder etc. can be modelled, but for example extruded polygons with a free amount of polygons cannot. This problem will be fixed as soon as PMO supports arrays for DatatypeProperties (we will then also intorduce subclasses from classes Vertex, Edge and Face).

A box modelled as a solid exisiting of faces existing of edges existing of vertexes

A Cuboid has Typicaly 8 vertices (figure 11).

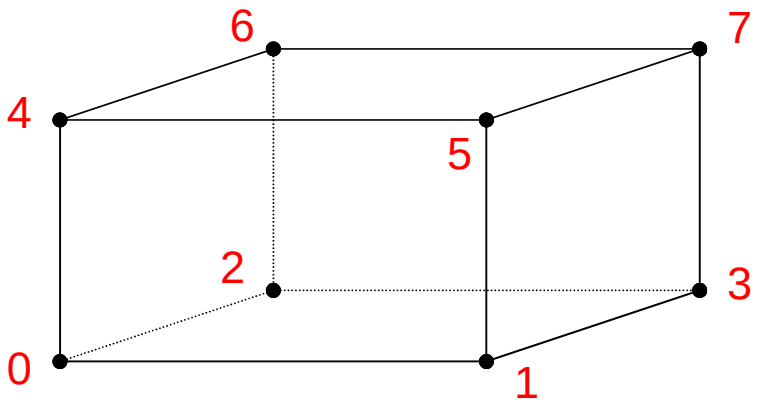


Figure 11 Vertices

We name the points VertexCuboid_0 till VertexCuboid_7 (Figure 12).

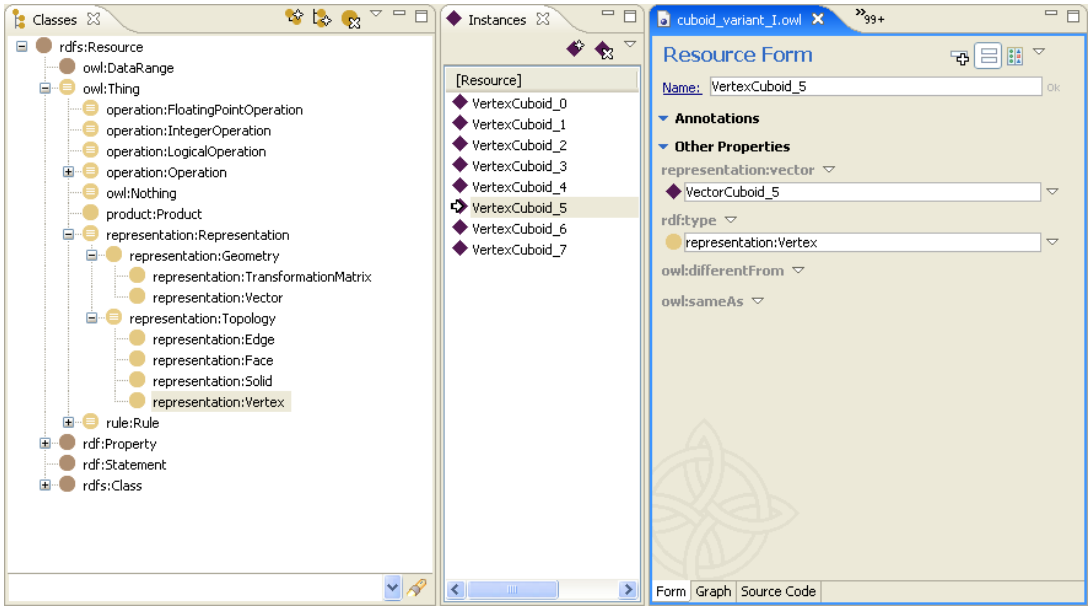


Figure 12 Vertices in TBC

Each Vertex is only a topological point, it has however a reference to a Geometrical representation in a Vector. As we see in the screenshot for example a VertexCuboid_5 has a reference to Vector-Cuboid_5.

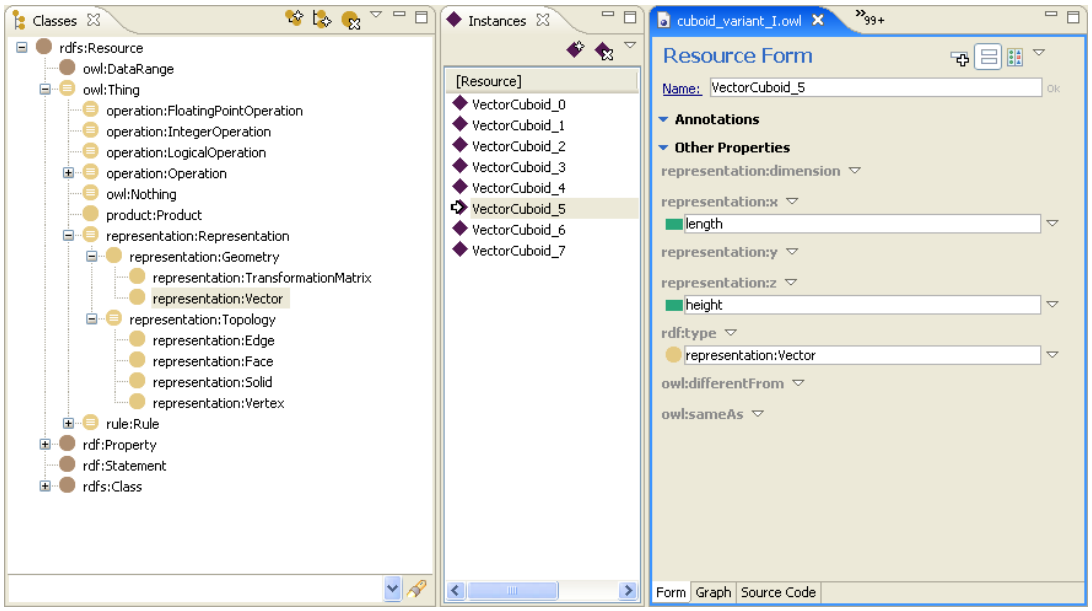


Figure 13 Vertices referencing geometry (vectors)

Each VectorCuboid_... contains information about it geometrical location. For VectorCuboid_5 it means that its x component has the value of DatatypeProperty length. The vectors y component has value zero, what is default, so no value is given and the vectors z component has the value height. At this moment the Topological Vertices and the related Geometrical Vectors that contain the real position in space are defined.

A Cuboid has Typicaly 12 edges (Figure 14).

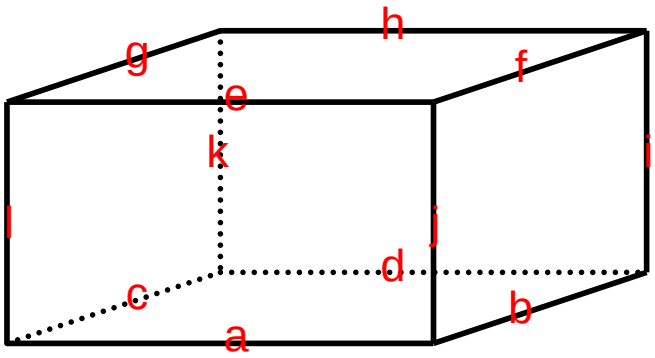


Figure 14 Edges

We name the lines EdgeCuboid_a till EdgeCuboid_l.

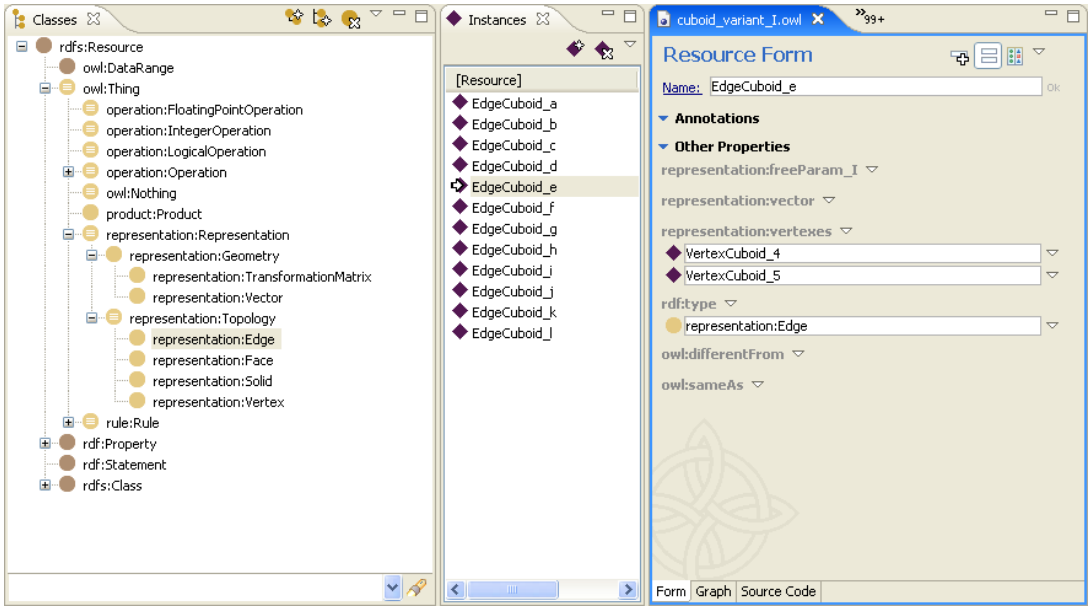


Figure 15 Edges in TBC

Each Edge is only a topological line between two points, so it's a topological relation between edges and vertices and only the vertices contain a link to the geometrical information. As we see in the screenshot for example a EdgeCuboid_e has a reference to VertexCuboid_4 and VertexCuboid_5.

A Cuboid has Typicaly 6 faces (Figure 16).

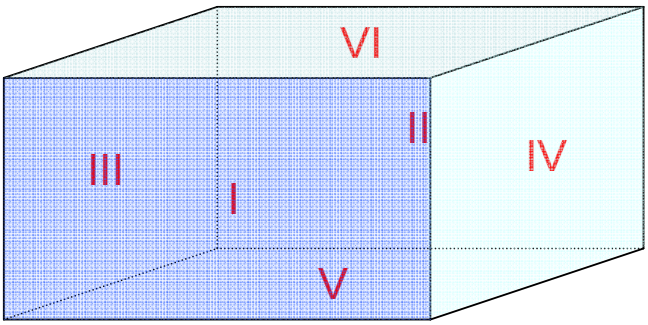


Figure 16 Faces

We name the 2D areas FaceCuboid_I till FaceCuboid_VI.

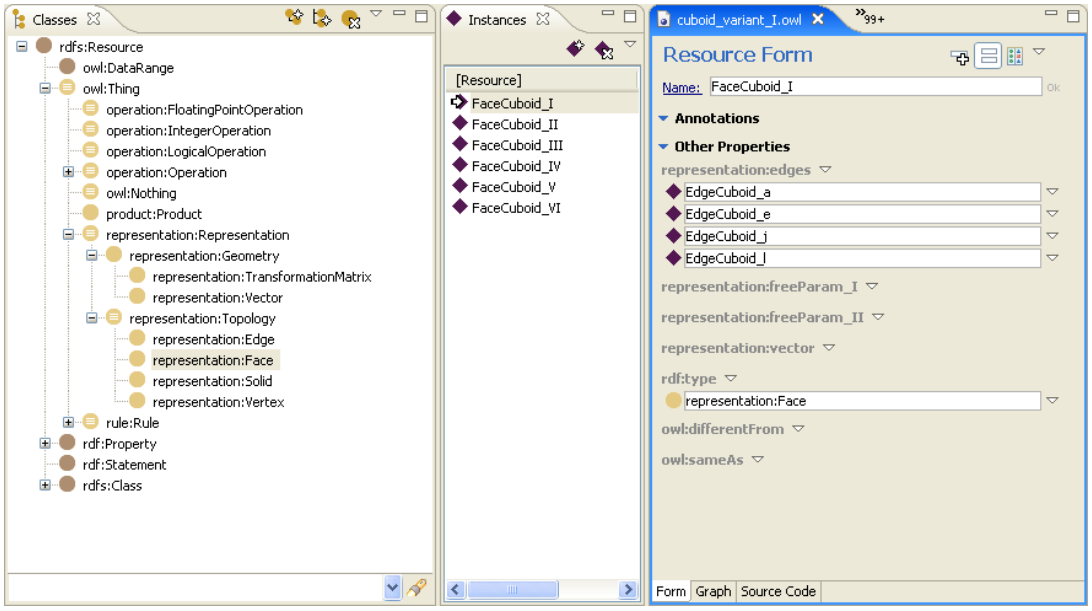


Figure 17 Faces in TBC

Each Face is only a topological 2D area bounded by one or more lines with in this case always exactly four points, so it's a topological relation between faces and edges and edges on its turn with vertices and only the vertices contain a link to the geometrical information. As we see in the screenshot for example a FaceCuboid_I has a reference to EdgeCuboid_a, EdgeCuboid_e, EdgeCuboid_j and EdgeCuboid_l.

A Cuboid has Typicaly 1 solid, (Figure 18).

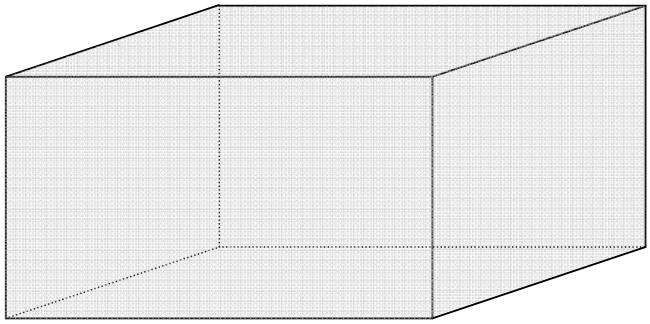


Figure 18 One solid

We name this 3D volume SolidCuboid (Figure 19).

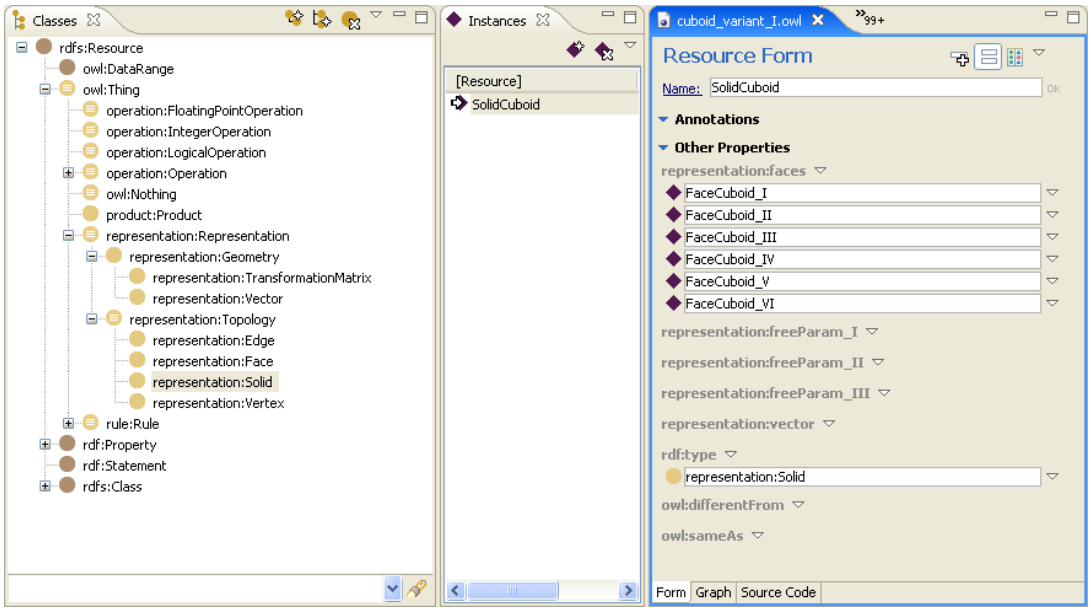


Figure 19 Solid in TBC

Each Solid is only a topological 3D area bounded by one or more 2D areas with in this case always exactly six 2D areas, so it's a topological relation between solids and faces, on its turn with faces and edges and edges on its turn with vertices and only the vertices contain a link to the geometrical information. As we see in the screenshot the SolidCuboid has a reference to FaceCuboid_I, FaceCuboid_II, FaceCuboid_II, FaceCuboid_IV, FaceCuboid_V and FaceCuboid_VI.

Now we created the complete Topological relations construction including the link to the Geometrical representation. We already defined DatatypeProperties length, height and depth to be used in the geometrical representation.

What we do now is create a new class Cuboid that is the is a subclass of Solid (Figure 46).

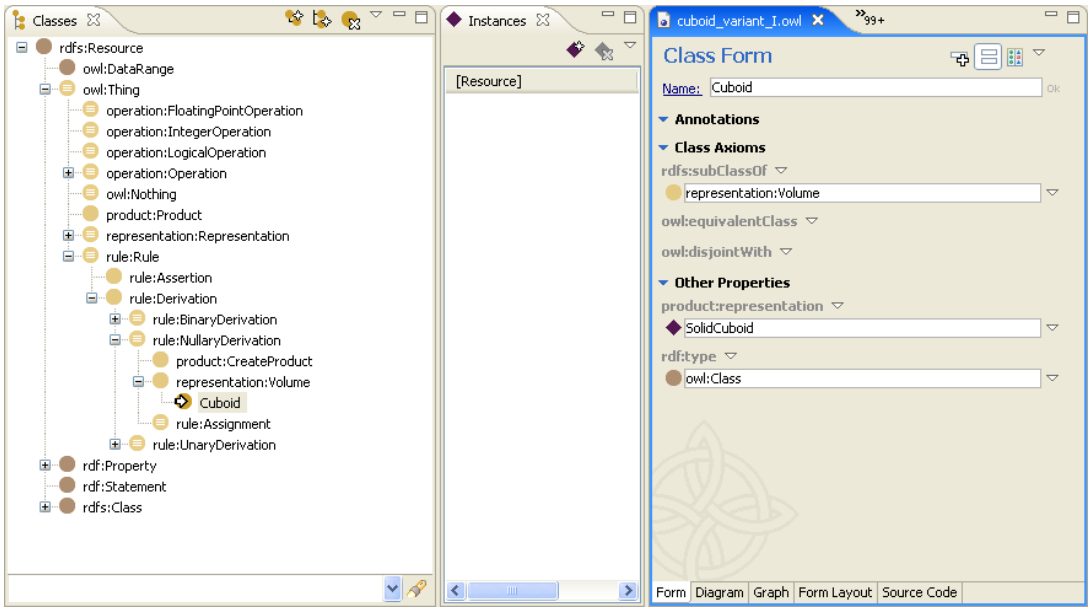


Figure 20 Subclassing the solid in cuboid

As can be seen in the screenshot this new class Cuboid references to the SolidCuboid Topological representation.

The DatatypeProperties will have as domain the classes that they are influencing, in this case length, height and depth will be domain of Cuboid. See example of height.

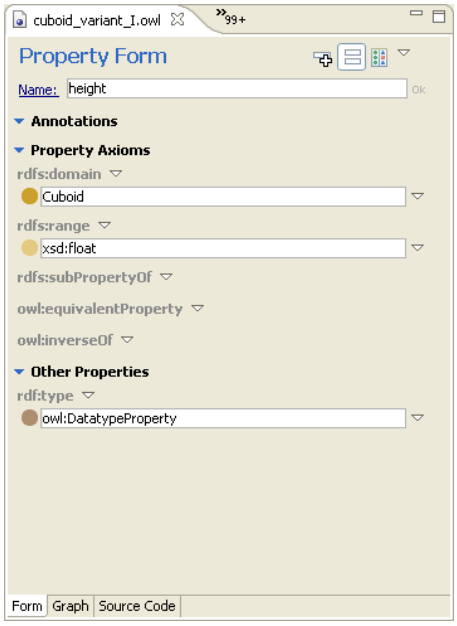


Figure 21 Subclassing the solid in cuboid

As previous, but now edges modelled as formulas instead of vertexes

In previous example we found a solution to build the Topological structure of a Cuboid. In this example each Topological Vertex is defined. The Edges themselve could however also be linked to a Vector directly. To link an Edge directly to a Vector we need to define a free parameter for each Vector. Let's consider once more the earlier figure of edges.

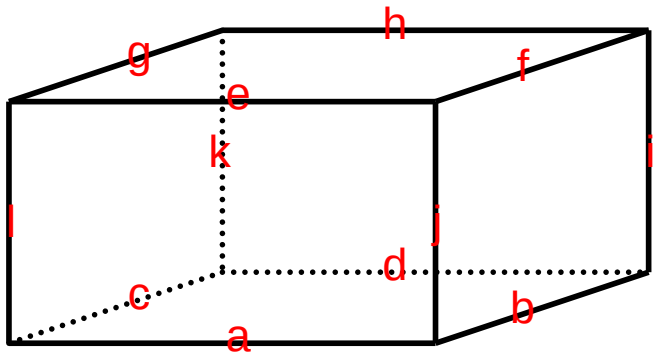


Figure 22 Edges revisited

If we look at Edge e we see that we have to define a Vector that is a strait line between the two points:

- [0, height, 0]
- [length, height, 0]

In the Edge a free parameter will be defined. The line will be defined for the value of this free parameter [0, 1]. A new DatatypeProperty param is defined, also the Multiplication Operator Multiplication_param_x_length that defines a multiplication between param and length.

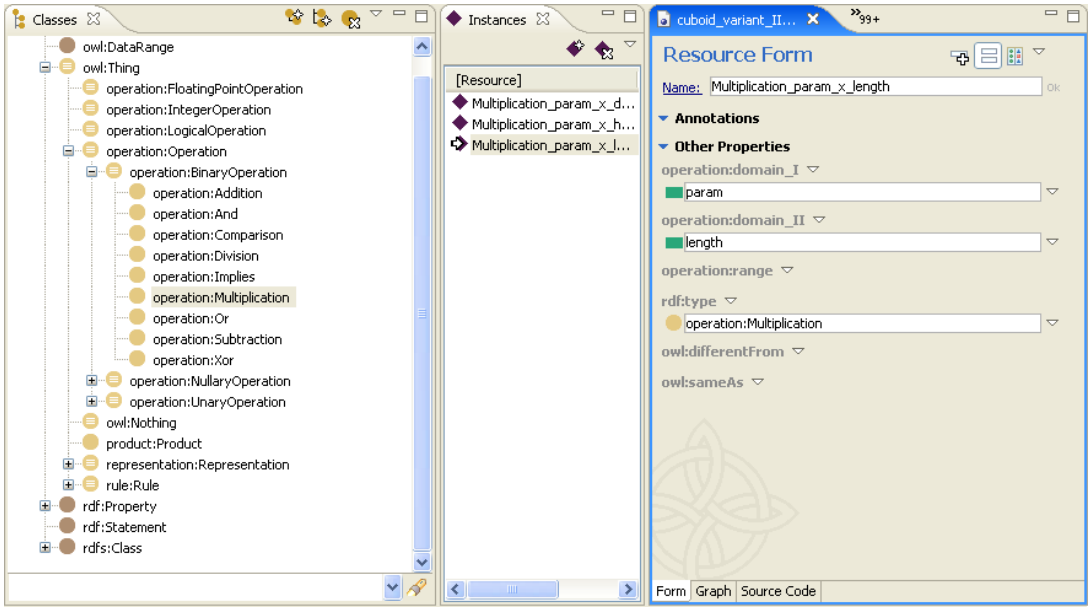


Figure 23 Parameter definition

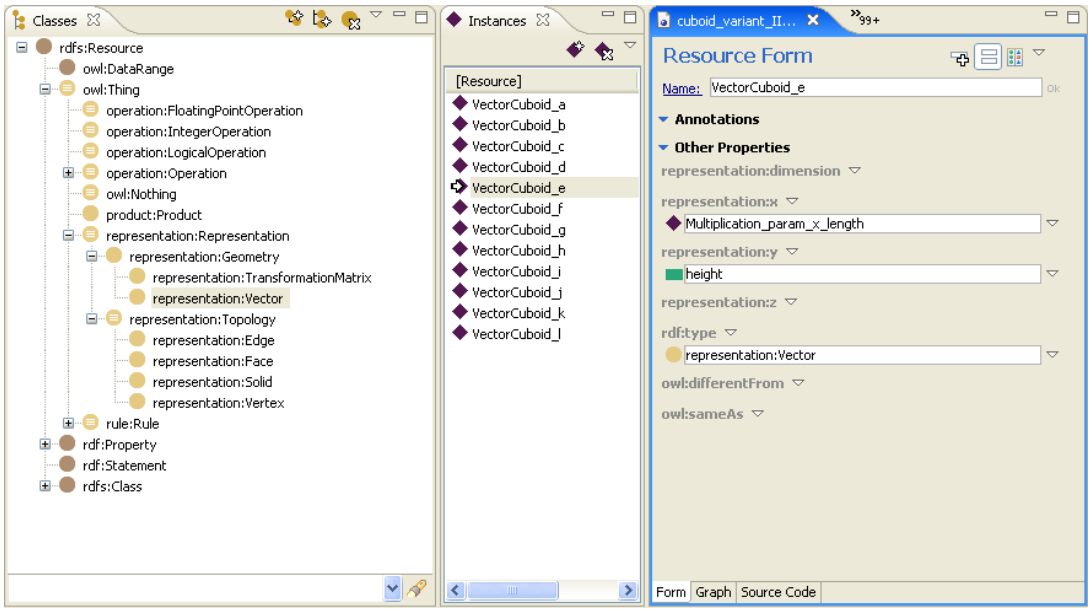


Figure 24 Vector definition

A VectorCuboid_e is defined. This Vector defines the Geometric representation.

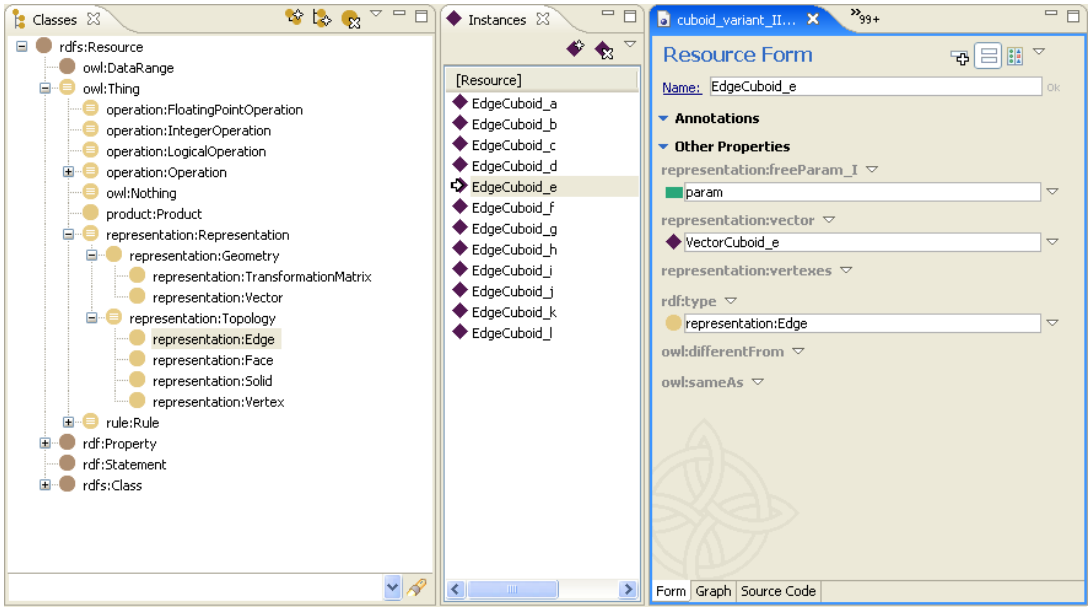


Figure 25 Edge definition

The Topology representation EdgeCuboid_e is referencing to the Geometrical representation VectorCuboid_e. Within EdgeCuboid_e the free param is defined, in this case param. So when param is varying between [0, 1] the Vector will vary between

[0 x length, height, 0] en [1 x length, height, 0] = [0, height, 0] en [length, height, 0].

So it is exactly representing the correct line. In this solution no Vertices will be defined.

As previous, but now faces modelled as formulas instead of edges

In the previous two examples we found solutions to build the Topological structure of a Cuboid. In these examples each Topological Edge is defined. The Faces themselve could however also be linked to a Vector directly. To link a Face directly to a Vector we need to define two free parameters for each Vector. Let's consider once more the earlier figure of faces.

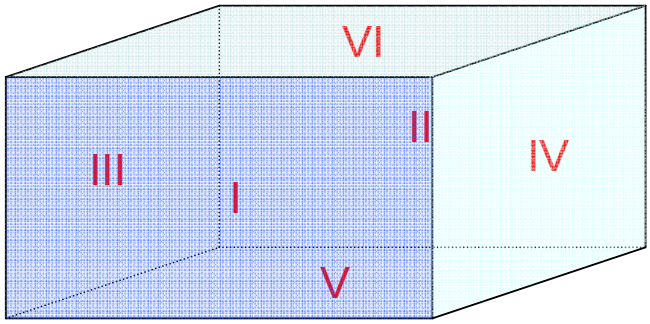


Figure 26 Faces revisited

If we look at Face I we see that we have to define a Vector that is an exact square between the four points:

- [0, 0, 0]
- [length, 0, 0]
- [length, height, 0]
- [0, height, 0]

In the Face two free parameters will be defined. The edge will be defined for the values of this free parameters [0, 1] and [0, 1]. Two new DatatypeProperties param_one and param_two are defined, also some Mulitplication Operators like Multiplication_param_one_x_length are defined that defines a muliplication between param and length, like in the previous solution.

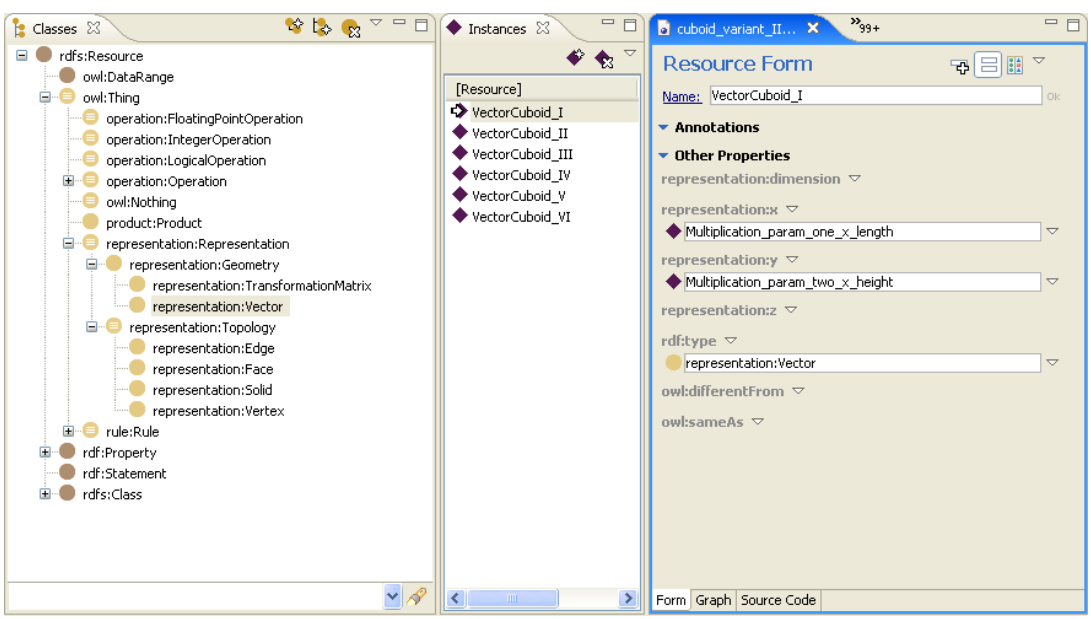


Figure 27 Parametric definition

A VectorCuboid_I is defined, this Vector defines the Geometric representation.

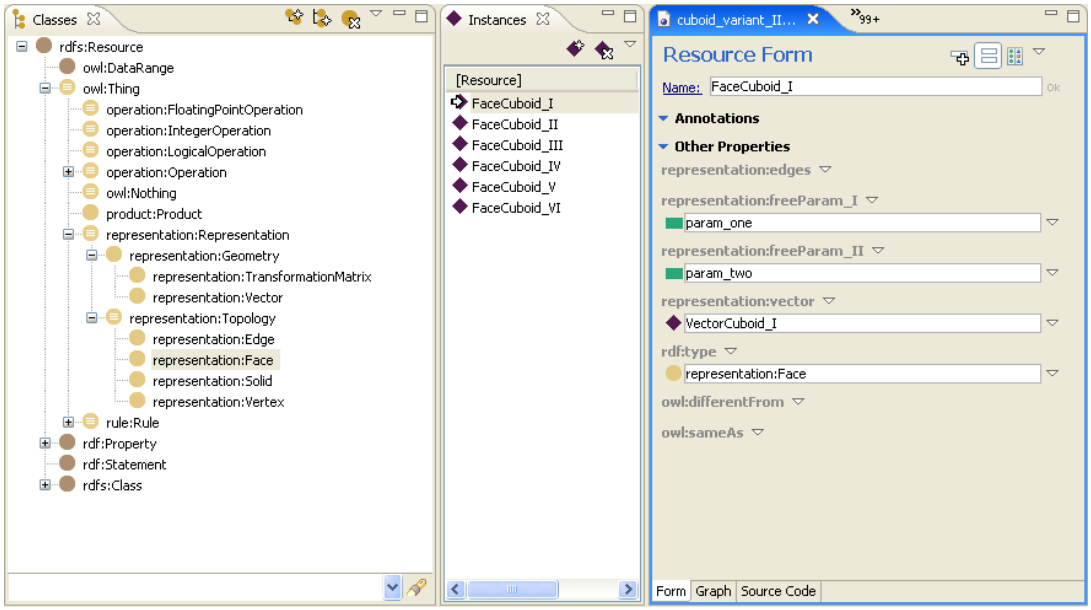


Figure 28 Face definition

The Topology representation FaceCuboid_I is referencing to the Geometrical representation VectorCuboid_I. Within FaceCuboid_I two free params are defined, in this case param_one and param_two. So when param_one and param_two are varying between [0, 1] the Vector will vary between

[0, 0, 0], [length, 0, 0] [0, height, 0] en [length, height, 0].

So it is exactly representing the correct 2D area.In this solution no Vertices nor Edges will be defined.

As previous, but now solids existing as formulas instead of faces

In previous 3 examples we found solutions to build the Topological structure of a Cuboid. In these examples each Topological Face is defined. The Solid it selve could however also be linked to a Vector directly. To link a Solid directly to a Vector we need to define three free parameters for each Vector. Let's consider once more the earlier figure of a solid.

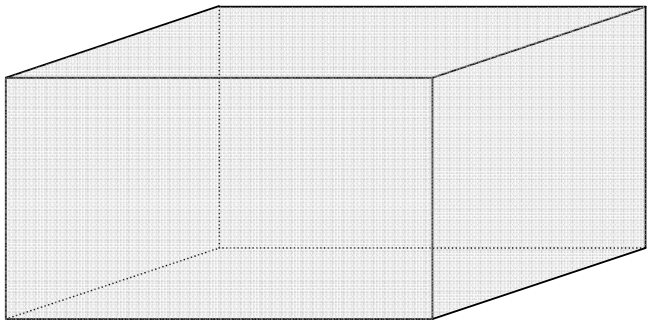


Figure 29 Solid revisited

If we look at the Solid we see that we have to define a Vector that is an exact cuboid between the eight points:

- [0, 0, 0]
- [length, 0, 0]
- [length, height, 0]
- [0, height, 0]
- [0, 0, depth]
- [length, 0, depth]
- [length, height, depth]
- [0, height, depth]

In the Solid three free parameters will be defined.

Three new DatatypeProperties param_one, param_two and param_three are defined, also some Mulitplication Operators like Multiplication_param_one_x_length are defined that defines a multiplication between param and length, like in the previous two solutions.

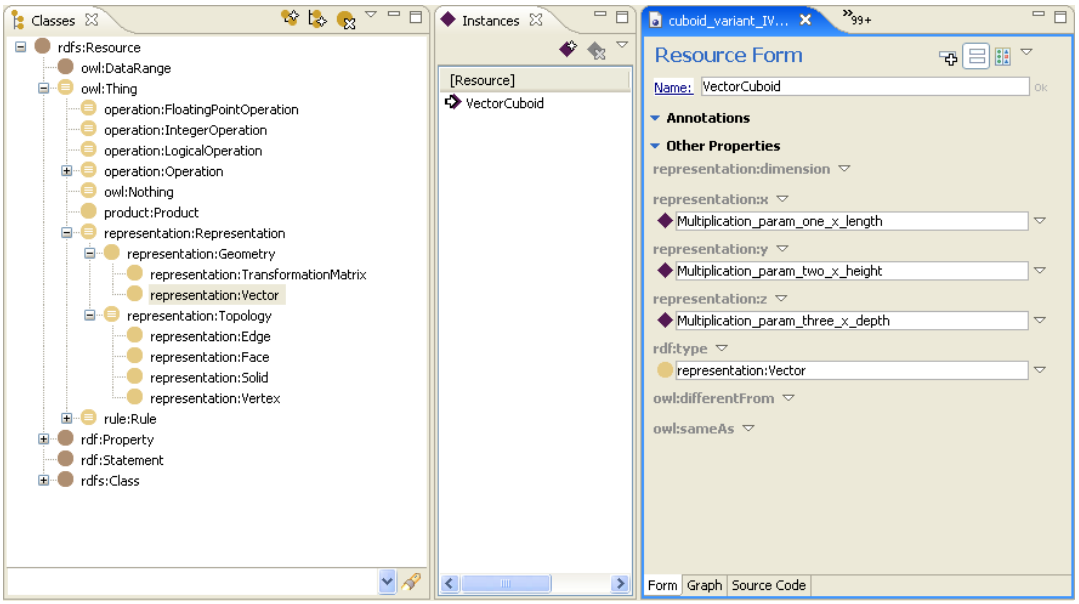


Figure 30 Parametric definition

A VectorCuboid is defined, this Vector defines the Geometric representation.

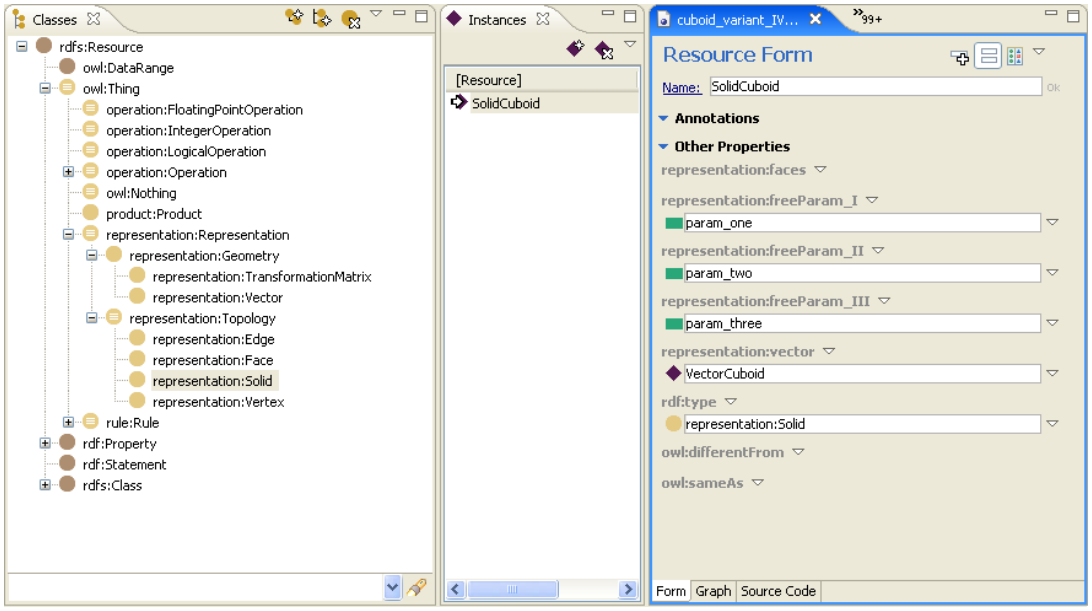


Figure 31 Solid definition

The Topology representation SolidCuboid is referencing to the Geometrical representation Vector-Cuboid. Within FaceCuboid three free params are defined, in this case param_one, param_two and param_three. So when param_one, param_two and param_three are varying between [0, 1] the Vector will vary between [0, 0, 0] … [length, height, depth]. So it is exactly representing the correct 3D volume.In this solution no Vertices nor Edges nor Faces will be defined.

Use of Cuboid (independent of solution)

The use of this Cuboid is now easy. No matter the solutions we discussed the use will be the same. Include one of the solution files and a Cuboid can be defined with filling in only three parameter values.

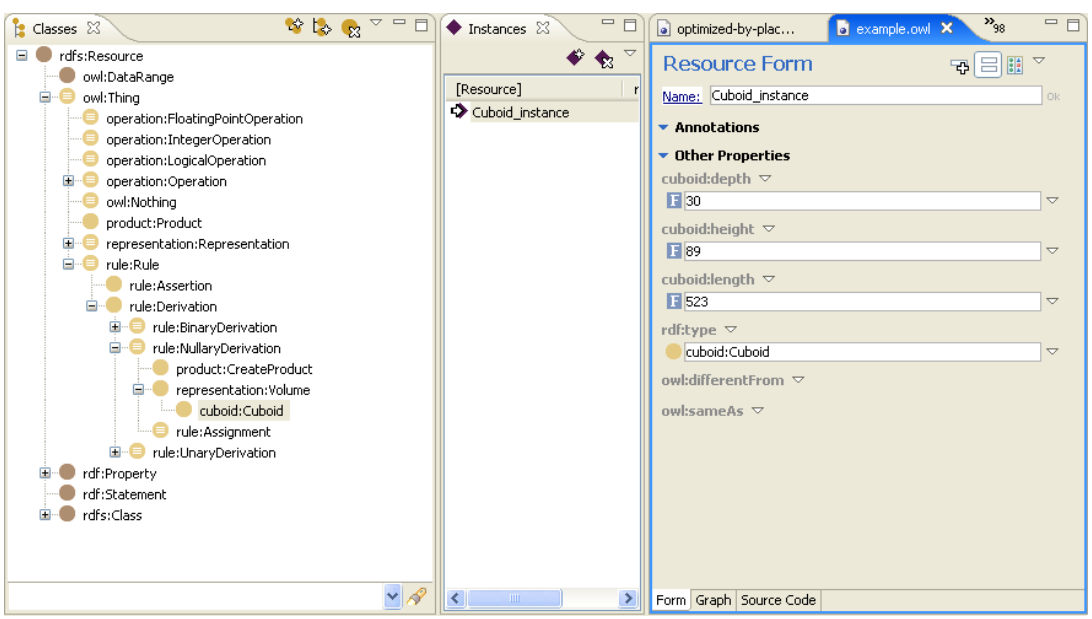


Figure 32 Using the Cuboid

All this rather technical (but end-user product dependent!) representation knowledge will be described seperately from the semantic information. It will be kept in an ontology called name-representation.owl where "name" is replaced by the set of class ontologies belonging together. In the example that follows later this will be cuboid.owl stored in the d23 folder.

4.6 Rules

In the previous sections we have seen many forms of local rules or 'conditions' as they are called in OWL which are expressed within the ontology. Clearly in SWOP there is a need for several types of global rules. These rules are typically positioned above/on top of the ontology. We identify two main rules types (see also [REWERSE]):

1. Assertions as Integrity Rules, that are checked/validated, and
2. Derivations as Production/Action Rules, that are done/executed.

Another distinction for rules is in 'determined' or 'undetermined'. A rule is 'undetermined' if it still contains some freedom like in case of logical OR-expression or numerical OR-range like 'X>25'. Assertions can be both determined or undetermined. Derivations have to be determined. Both types involve logical and non-logical (like numerical or string) binary, unnary and/or nullary operations.

Some more examples to clarify:

Assertions:

wheelDiameter > (2.8 * axisDiameter)

IF (AMOUNT[7] (PowerSupply) >= 1) THEN (safetyLevel = "1" OR "2")

IF (AMOUNT (window) >= 3) THEN (AMOUNT (door) < = 1)

IF machineType = "special" THEN (bladeColour = green)

Derivations:

windowHeight := ((2 * windowWidth) + 0.04)

IF machineType = "special" THEN (bladeColour := green)

Rule level

All rules above are class-level rules: they are defined for classes, properties etc. and are in principle relevant for all individuals of the classes and their properties. For SWOP we also want to support individual-level rules that are associated to not all but one or some individuals.

Rule Implementation Approach

In this stage of SWOP we identified three main alternative implementation approaches:

1. Use an existing rule language like SWRL or JenaRL

[7] 'AMOUNT' is a predefined nullary (non-logical) operation that gives the amount of individuals for a Class or ObjectProperty. It can be used e.g. in comparisons to specify more structural rules involving a product subclass or part actually chosen, controlled by the relevant cardinalities involved.

Typically the expressive power is very limited. If applicable there is the advantage of reuse of existing rule engines like Racer or Pellet. Such language are formally very well defined but typically fail to deliver the needed 'business logic'.

2. Use a SWOP-specific meta-ontology

   Clearly, in this case you have to write your own rule engine too. This approach gives you the flexibility to define your own expressiveness (and all kind of satisfied modeling assumptions typically not considered by generic engines).

3. Use a non-owl format (like a Microsoft Excel spreadsheet)

   Although theoretically less nice it would mean a more end-user oriented approach to knowledge representation. Its easier for an end-user to edit an Excel spreadsheet then a specific complex global rule in say Protégé or TopBraid Composer. Disadvantage is the very limited expressive power (simplicity comes at a price).

For SWOP we decided to go for option 2 which seems the best compromise between usefulness/expressive power and ontology integration (still in OWL). A detailed model can be found in appendix D. In the next figure the overview is given.

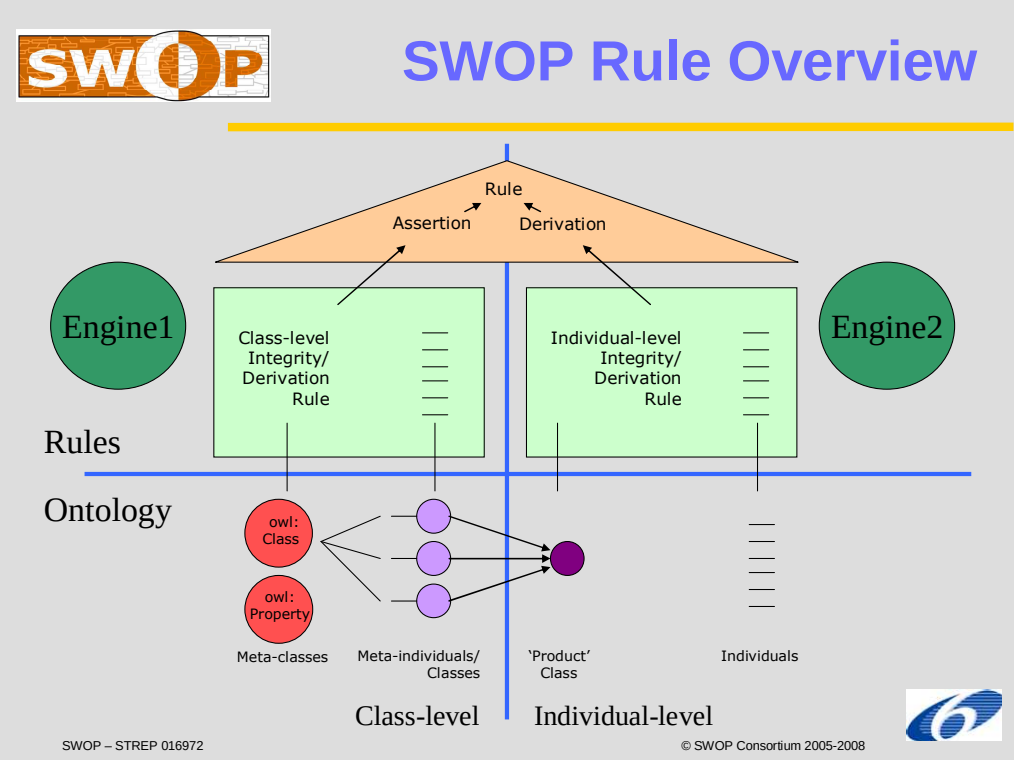


Figure 33 Rule Types

The Engines are needed as modules to check/validate and/or do/execute with respect to the underlying ontology. Engine1 is dealing with the class level rules, engine2 with the individual-level rules. The latter rules are all instances of a rule structure that only 'knows' our PMO's generic 'Product' class. It cannot be dependent on end-user product ontology classes since the structure of these rules is to be processed by a generic engine2 module.

The rule.owl ontology combines complex operations and includes programming concepts:

- class Rule, all kind of rules.
- class Assertion, rules that are always true. These rules can be defined for classes and for individuals.
- class Derivation, this class integrates the sequential concept of a derivation. The reason that this (programming) concept not is defined as stand alone concept is that OWL does not support ordering, this has as a consequence that grouping statements and then adding ordering to this grouped statements will result in redundant information. Integrating the sequential concept in (every) Statement and making this optional prevents us from creating redundant information.
- class NullaryDerivation, each derivation that doesn't contain a (sequence of) derivations itself. A typical nullary derivation is an assignment.
- class UnaryDerivation, each derivation that contains 1 (sequence of) derivation(s) itself, in this structure it is always a reference to this (in this relation) underlying (sequence of) derivation(s). A typical unary derivation is a while-loop.
- class BinaryDerivation, each derivation that contains 2 (sequences of) derivations itself, in this structure it is always a reference to this (in this relation) underlying derivations. A typical binary derivation is an if-then-else derivation
- each derivation has a condition, the condition represents if and which (sequence of) derivation(s) has to be executed

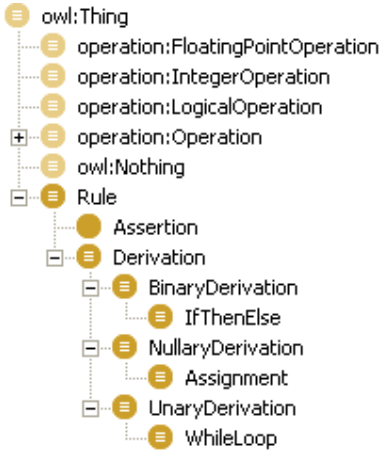


Figure 34 Rule Types in TBC

The Rule Ontology makes use of a separate Operation ontology containing:

- class Operation, this is the parent of all possible operations from what a complex operation can exist, with exception of DatatypeProperty. A DatatypeProperty can be seen as a child of Value from the NullaryOperation (in the context of the specialisation relation). To make optimal use of OWL concepts we use owl:DatatypeProperty.
- Class NullaryOperation, each operation that isn't defined on top of another operation. A typical nullary operation is Value
- class UnaryOperation, each operation that itself is defined on top of another operation, in this structure it is always a reference to this other operation. A typical unary operation is a

Minus where the operation gives the negation value of the operation it is defined on top of (for the operation his child relation)

- class BinaryOperation, each operation that itself is defined on top 2 other operations, in this structure it is always a reference to these (in this relation) underlying operations. A typical binary operation is the Addition where the summation of the two underlying operations is the result of this operation

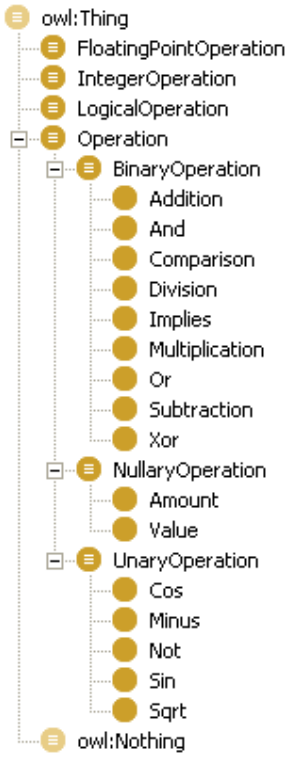
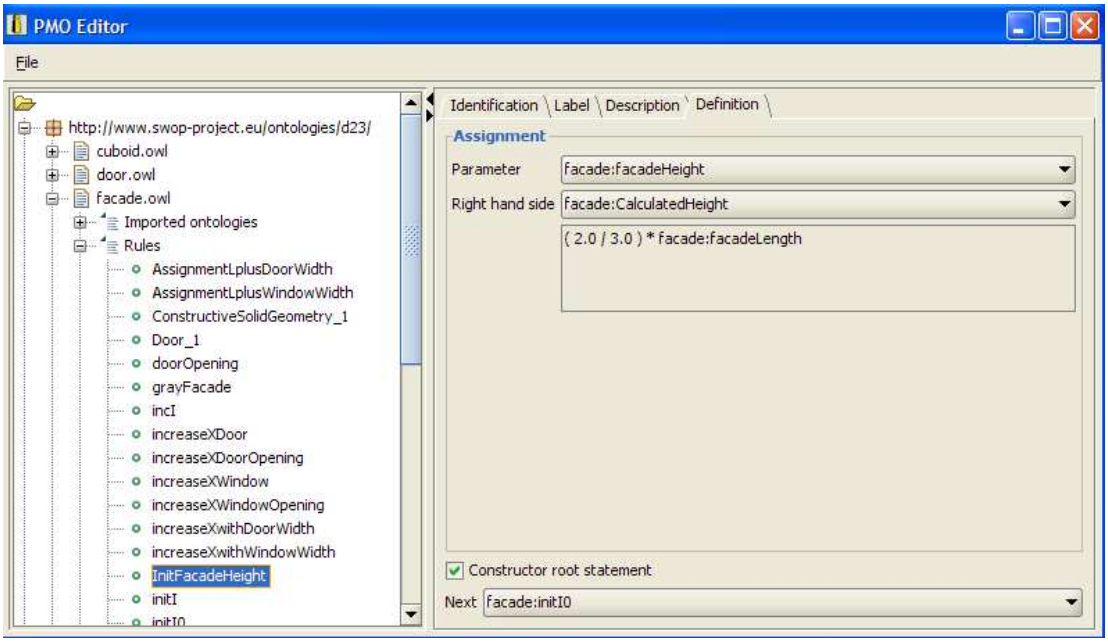


Figure 35 Operations in TBC

More complex logical operations can be constructed from existing logical operations via expressions involving logical operators. These available operators are defined in Appendix D.

PMO Editor

Typically the drawback of a meta-approach in OWL for rules is the tedious input process. Luckily EU project ManuBuild (building already on SWOP) developed a PMO Editor where you can specify the rules and their operations in a more user-friendly way and generate the actual OWL code needed.



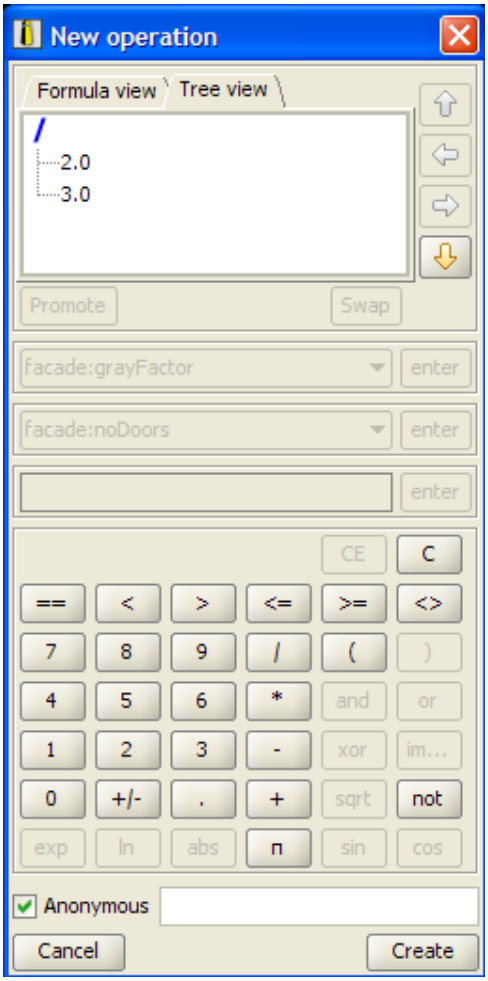


Figure 36 PMO Editor (operation part) by ManuBuild

5 APPLYING PMO, A TYPICAL EXAMPLE

In this chapter we will show for a, not too complex but typical, example how the SWOP Product Modelling Ontology (PMO) approach is applied. In this example obeject properties are not yet present. We start with a specific real life situation of a facade individual as depicted in the next figure.

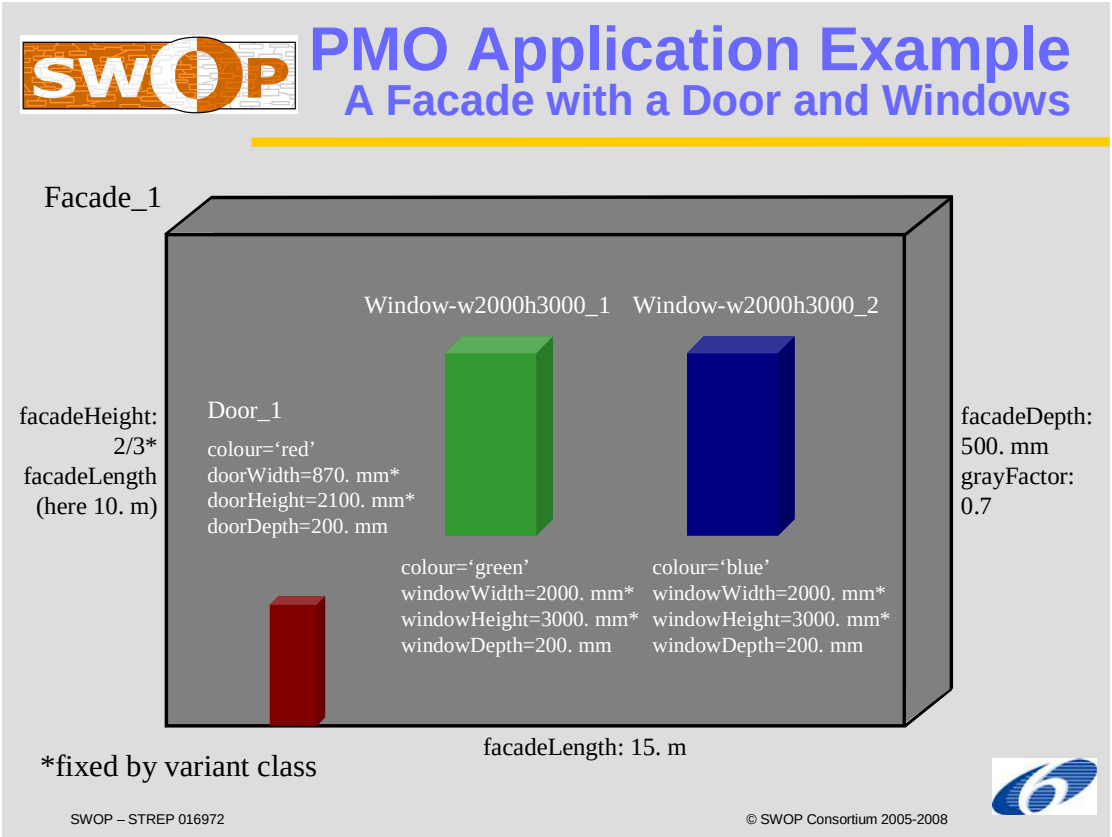


Figure 37 The example graphically

This individual Facade, referred to as "Facade_1" has a length of 15. meter (m), a derived height of 10. meter (m) and a depth of 500 mm). These float values could have varied in a flexible way, the default for the length being 12.5 m; the default for the depth being 500 mm and no default for the height since it is always 'overruled' by the calculation: the height is 2/3 of the length. This facade has two windows as parts that have the same width and height but differ in colour. The width and the height of a window are measured in mm. Here, the width and height values are associated with a specific 'variant' of a window, a "Window-w2000h3000". This class variant's width and height are always fixed to respectively 2000. mm and 3000. mm. The depth is variable (here 200. mm being the default value). The colour can take the value 'red', 'green' or 'blue' ('red' being the default). Another variant not chosen is a window with fixed width of 1800 mm, Window-w1800. In the individual facade at hand, the left one is green and the right one blue. The colour of the façade is determined by its greyFactor from black ("0") to white ("1"). For this individual facade the grey factor is moderate grey ("0.7"). A facade can have in general 1 to 4 windows. Furthermore this facade has one (default) red door with a default width of 870. mm, a default height of 2100. mm and a default depth of 200. mm. These float values could have varied (they are not fixed with respect to some variant as with the windows). A facade has always exactly one door. The default amount of parts

for windows and doors are equal to the corresponding minimal amount (here being “1” for both window and door).

We use the latest version of the TopBraidComposer (TBC) ontology editor (version 2.5.0) for going through the following 7 typical steps:

- STEP1: Define the Ontology files/headers
- STEP2: Define the relevant Classes & Put them in a specialisation hierarchy
- STEP3: Put the Classes in an orthogonal (and typical) decomposition hierarchy
- STEP4: Define the relevant (datatype) properties for all the classes with their datatypes
- STEP5: Define the semantic rules (here: derivation of facadeHeight)
- STEP6: Define the rules for generating an IFC representation

  Assume an implicit placement of the door and windows in the façade like for example for 1 door and 2 windows:

  Door:
  Placement vert. = 0
  Placement hor. = (width facade - (width door + 2 * width window))/(1+2+1)

  First window:
  Placement vert. = (height facade - height window) * 3 /4
  Placement hor. = 2 * (width facade - (width door + 2 * width window))/(1+2+1) + width door

  Second window:
  Placement vert. = (height facade - height window) * 3 /4
  Placemen hor. = 3* (width facade - (width door + 2 * width window))/(1+2+1) + width door + width window

  (similar in general case of other amounts of windows)

- STEP7: Configure the typical example in the PMO Configurator: Show it for the individual we modelled. This should result in a visualisation resembling the figure 37 we started with….i.e. the ’proof-of-the-pudding’!)

STEP1: Define the ontology files/headers

We will always have one file per relevant end-user ontology class. In such a file all relevant knowledge for that class is collected: specialisation info, decomposition info, (user-defined) datatype properties, rules etc.. The name of the file is the same (but ‘de-capitalised’) as the class name and has the extension ‘.owl’. We will collect all the files in a (web) folder with a lower case name that is indicating the root class files (like ‘pmo’, ‘trimek’ or in case of this example ‘d23’)[8].

In TBC we define for each ‘one class’-ontology the name spaces used in this ontology together with their local prefixes (short cuts) including the standard generic ones like xsd, rdf, rdfs, owl and owl11 but also our own PMO ones like product, representation, rule and operation (see the next figure). We declare a ‘Default Namespace for the current ontology and a ‘Base URI’ making it easier to define its content (this way you don’t have to repeat all the time for each defined class, property etc.). The actual names used are again the same as the file name (without the extension). Some of the identified name spaces/ontologies are actually imported. The exact rules for adding non-generic (user-defined) name spaces and imports are described in appendix B.

[8] Local files can be organized differently (agreed or not). In SWOP we agreed a folder/file structure as indicated in Appendix F.

In our example we have six classes so we will have six files in our 'd23' folder: one for each relevant class and one for the instantiated occurrence.

- facade.owl
- window.owl
- door.owl
- window-w2000h3000.owl
- window-w1800.owl (the alternative variant not chosen in the example)
- facade_1.owl

As an example we will show the ontology header for the facade class. The others are similar.

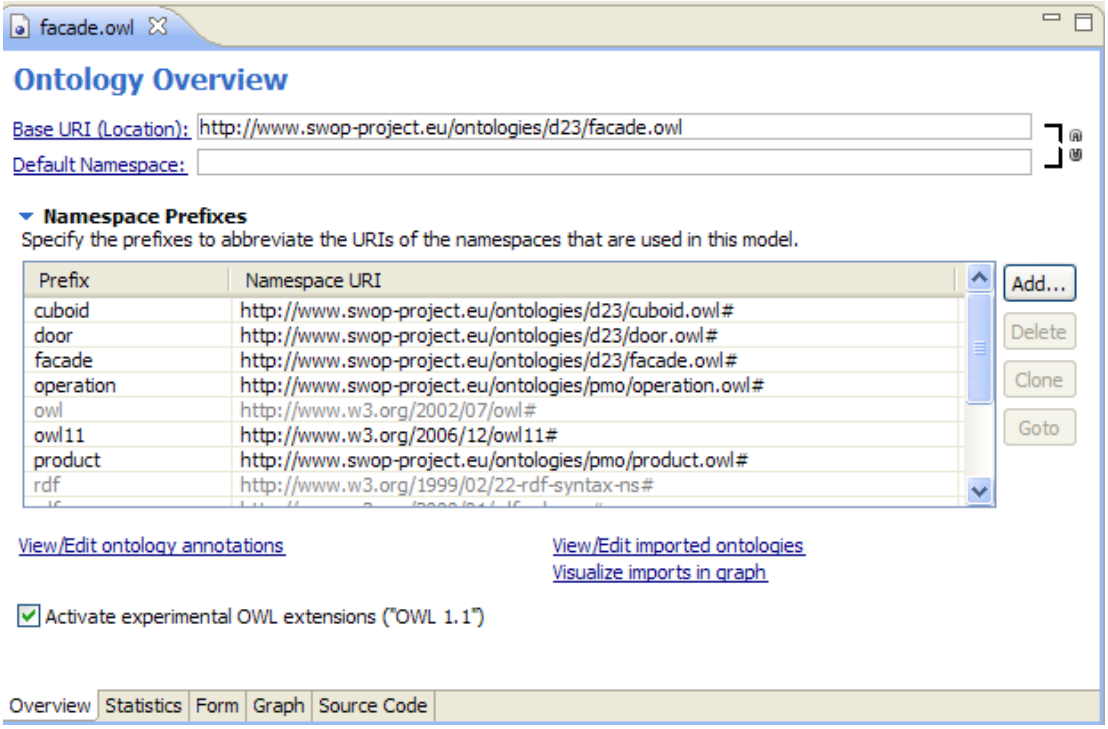


Figure 38

This results "under the hood" in the following OWL code:

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:facade="http://www.swop-project.eu/ontologies/d23/facade.owl#"
    xmlns:representation="http://www.swop-project.eu/ontologies/pmo/representation.owl#"
    xmlns:owl11="http://www.w3.org/2006/12/owl11#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:operation="http://www.swop-project.eu/ontologies/pmo/operation.owl#"
    xmlns:window="http://www.swop-project.eu/ontologies/d23/window.owl#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rule="http://www.swop-project.eu/ontologies/pmo/rule.owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:cuboid="http://www.swop-project.eu/ontologies/d23/cuboid.owl#"
    xmlns:product="http://www.swop-project.eu/ontologies/pmo/product.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:door="http://www.swop-project.eu/ontologies/d23/door.owl#"
  xml:base="http://www.swop-project.eu/ontologies/d23/facade.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.swop-project.eu/ontologies/d23/window.owl"/>
    <owl:imports rdf:resource="http://www.swop-project.eu/ontologies/d23/door.owl"/>
```

```
  </owl:Ontology>
 ...
</rdf:RDF>
```

Note that the “rdf:about” tag in “<owl:Ontology rdf:about="">” is left blank. It is now derived by default from the URI name space defined (here: “http:// www.swop-project.eu/ontologies/d23/facade.owl”).

Our containers are now set up to start STEP2.

STEP2: Define the relevant Classes & Put them in a specialisation hierarchy

The classes we need to start with are:

- Facade
- Door, and
- Window

These top-level classes become a subclass of the already existing most generic PMO Class “Product”. Implicitly these classes are disjoint (something is a facade òr a window òr a door).

Furthermore we define two Window variants (still classes, not yet individuals!) as a subclass of Window:

- Window_w2000h3000 & Window_w1800

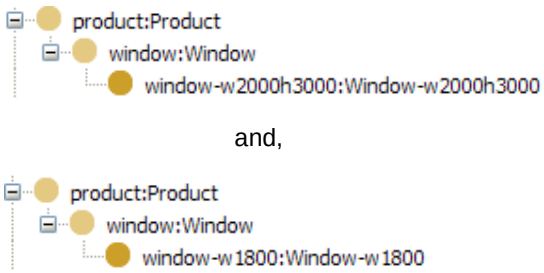


Figure 39

Later on, when we have defined the properties we will indicate that say class “Window_w2000h3000 represents the set of all windows having a width of 2000 mm and a height of 3000 mm. For now, the relevant added OWL code distributed over the three different files is:

```
<owl:Class rdf:ID="Facade">

    <rdfs:subClassOf rdf:resource="http://www.swop-
      project.eu/ontologies/pmo/product.owl#Product"/>

</owl:Class>

<owl:Class rdf:ID="Window">

    <rdfs:subClassOf rdf:resource=" http://www.swop-
      project.eu/ontologies/pmo/product.owl#Product"/>

</owl:Class>

<owl:Class rdf:ID="Door">

    <rdfs:subClassOf rdf:resource=" http://www.swop-
```

```
      project.eu/ontologies/pmo/product.owl#Product"/>

</owl:Class>

<owl:Class rdf:ID="Window-w2000h3000">

    <rdfs:subClassOf rdf:resource=" http://www.swop-
      project.eu/ontologies/d23/window.owl#Window"/>

</owl:Class>

<owl:Class rdf:ID="Window-w1800">

    <rdfs:subClassOf rdf:resource=" http://www.swop-
      project.eu/ontologies/d23/window.owl#Window"/>

</owl:Class>
```

STEP3: Put the Classes in an othogonal (typical/class-level) decomposition hierarchy

PMO has already an object property defined called 'hasPart_directly' (the 'non-transitive' form…). We want to model that each facade has 1 to 4 windows and exactly one door. We can do this by defining min(imum) and max(imum) cardinalities in class axioms for the Class "Facade". In standard OWL this results in the situation below (last 4 lines; ignore the other cardinalities for the moment). We also add a "closure" axiom stating that all hasPart_directly links go to Window or Door individuals (and not to something else). In the figure it's the fifth line from below.

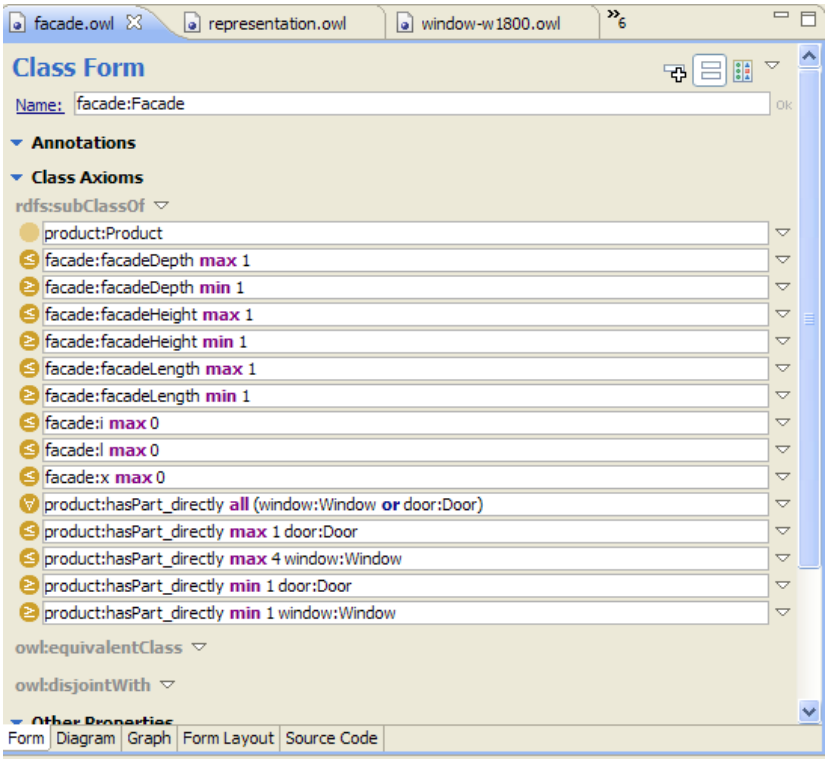


Figure 40

The corresponding OWL code for the Facade class is now extended with extra code like (for the relationship towards windows as parts):

```
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:valuesFrom rdf:resource="http://www.swop-
            project.eu/ontologies/d23/window.owl#Window"/>
        <owl:minCardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
        <owl:onProperty rdf:resource="http://www.swop-
            project.eu/ontologies/pmo/product.owl#hasPart_directly"/>
      </owl:Restriction>
    </rdfs:subClassOf>

    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:valuesFrom rdf:resource="http://www.swop-
            project.eu/ontologies/d23/window.owl#Window"/>
        <owl:maxCardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >4</owl:maxCardinality>
        <owl:onProperty rdf:resource="http://www.swop-
            project.eu/ontologies/pmo/product.owl#hasPart_directly"/>
      </owl:Restriction>
    </rdfs:subClassOf>
```

For other type of parts the extra conditions are of course similar.

As can be seen, the OWL 1.1 tag “valuesFrom” tag in the two code fragments adds the QCR information. Also clearly visible is the way axioms are defined using the subclass mechanism: ‘on the fly’ an anonymous Class is defined with members having the required characteristic(s) and then the current class is made a subclass of this axiom class.

Window, and its subclasses, has (here) no further decomposition. We make this ‘atomacy’ explicit by specifying at Window a max cardinality of 0 for the hasPart_directly property:

```
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:maxCardinality
      rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >0</owl:maxCardinality>
    <owl:onProperty rdf:resource="http://www.swop-
            project.eu/ontologies/pmo/product.owl#hasPart_directly"/>
  </owl:Restriction>
</rdfs:subClassOf>
```

STEP4: Define the (datatype) properties and relate them to the classes

Now we add the datatype properties relevant for the facades, windows and doors. As agreed earlier, each datatype property will have exactly one domain defined. Thus, we have to differentiate between the width of a facade and the width of a window explicitly. We first define 12 datatype properties with initially no cardinality restrictions (meaning: having in principle zero or more values) in the right .owl files:

- facadeLength, a float in m that is asserted (i.e. not 'derived'), with default value 12 m
- facadeHeight, a float in m that is derived, with no default value
- facadeDepth, a float in mm that is asserted, with default value 500 mm
- grayFactor, a float with no unit and a value between 0 (black) and 4 (white) that is asserted, with a default value of 0.3

- windowWidth, a float in mm that is asserted, with default value 1800 mm. This property is constrained for the variant class Window-w2000h3000 to exactly 2000 mm via a 'has-Value'-condition and for the variant class-w1800 to exactly 1800 mm ('default' becomes 'fixed').
- windowHeight, a float in mm that is asserted, with a default value of 400 mm. This property is constrained for the variant class Window-w2000h3000 to exactly 3000 mm via a 'has-Value'-condition
- windowDepth, a float in mm that is asserted, with default value 200 mm
- windowColour, an enumeration value that is asserted, with literal value of "red", "green" or "blue", with default value "red"
- doorWidth, a float in mm that is asserted, with default value 870 mm.
- doorHeight, a float in mm that is asserted, with a default value of 2100 mm.
- doorDepth, a float in mm that is in general asserted but derived when being part of a facade, default value 200 mm
- doorColour, an enumeration value that is asserted, with literal value of "red", "green" or "blue", with default value "red"

Note that cardinality restrictions, if relevant, are always defined at the class, not at the property (in SWOP). In the next figure we show how we define this information in TBC for facadeHeight. Also the domain for this property (here the class Facade) is specified via the "rdfs:domain" tag.

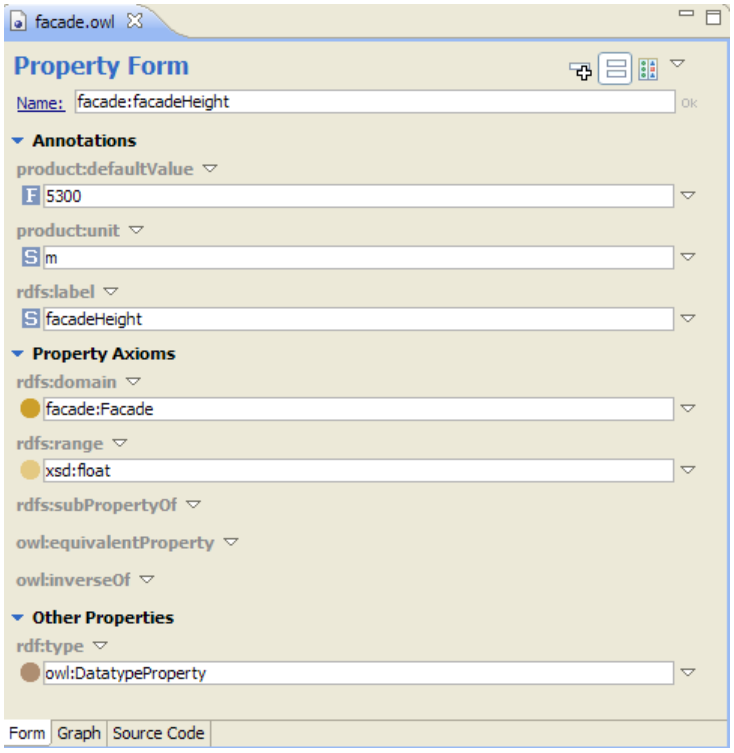


Figure 41

The windowColour property is a bit special:

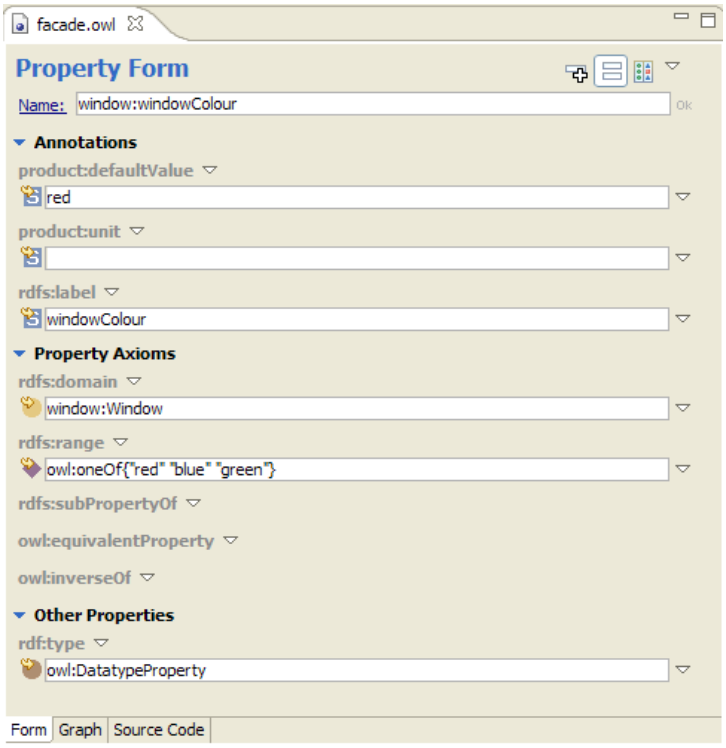


Figure 42

The range (or underlying datatype) of this property is defined as an 'owl:oneOf' of the strings mentioned earlier. In this example we also made the defaultValue "red". Let's have look at the corresponding OWL code for both examples facadeHeight and windowColour:

```
  <owl:DatatypeProperty rdf:about="#facadeHeight">
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >facadeHeight</rdfs:label>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
    <product:defaultValue
      rdf:datatype="http://www.w3.org/2001/XMLSchema#float"
    >5300</product:defaultValue>
    <rdfs:domain rdf:resource="#Facade"/>
    <product:unit rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >m</product:unit>
  </owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#windowColour">
    <product:unit rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    ></product:unit>
    <product:defaultValue
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >red</product:defaultValue>
    <rdfs:range>
      <owl:DataRange>
        <owl:oneOf rdf:parseType="Resource">
          <rdf:rest rdf:parseType="Resource">
            <rdf:rest rdf:parseType="Resource">
              <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-
                syntax-ns#nil"/>
              <rdf:first
                rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >green</rdf:first>
```

```
            </rdf:rest>
            <rdf:first
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >blue</rdf:first>
          </rdf:rest>
          <rdf:first
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >red</rdf:first>
        </owl:oneOf>
      </owl:DataRange>
    </rdfs:range>
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >windowColour</rdfs:label>
    <rdfs:domain rdf:resource="#Window"/>
  </owl:DatatypeProperty>
```

(The syntactic verbosity here, allowed values represented as a list, is a known OWL1.0 issue…)

So far, we have the following ontology defined (UML, fully generated by TBC):

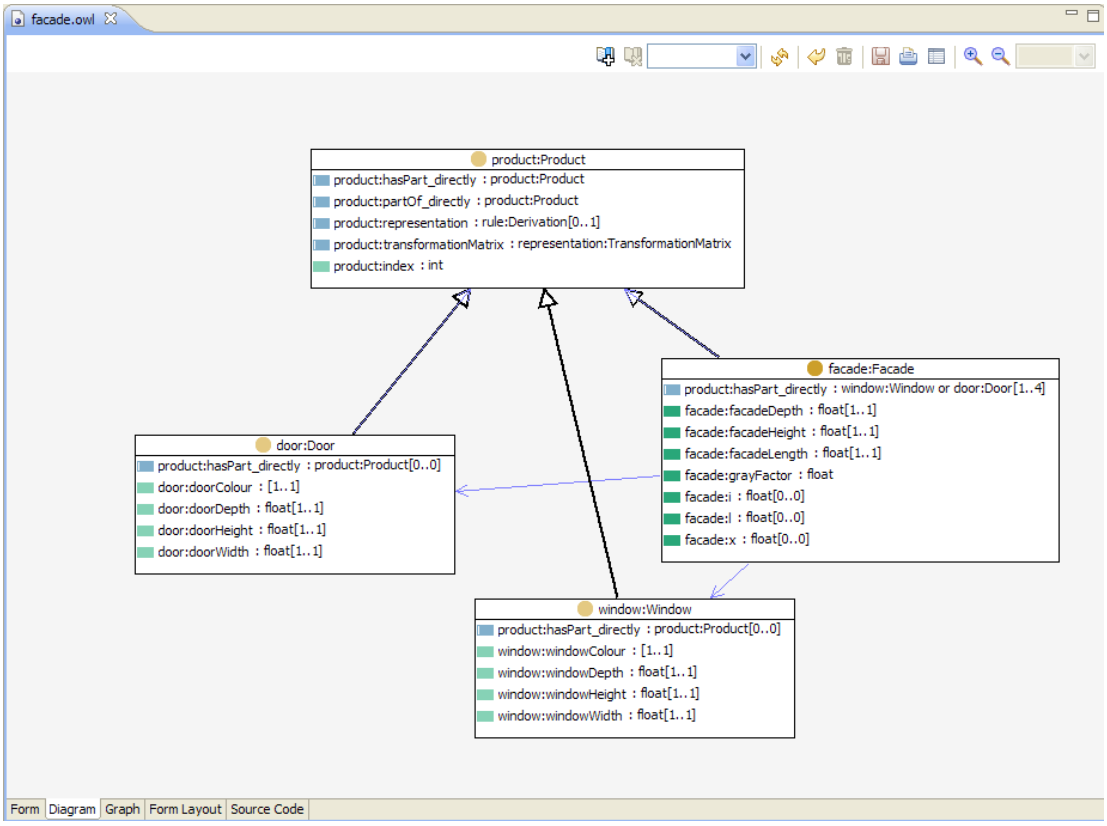


Figure 43

Note that the QCR-info is already correctly depicted here. For the moment ignore all the min/max cardinalities indicated in the brackets the will be described in the next step. What you can see though is the fact that “[0, 0]” is specified for the Window class since it’s a leaf-class for the decomposition tree.

Now we have the properties available we can also define more clearly our earlier defined window variant by defining two extra axioms at this class level:

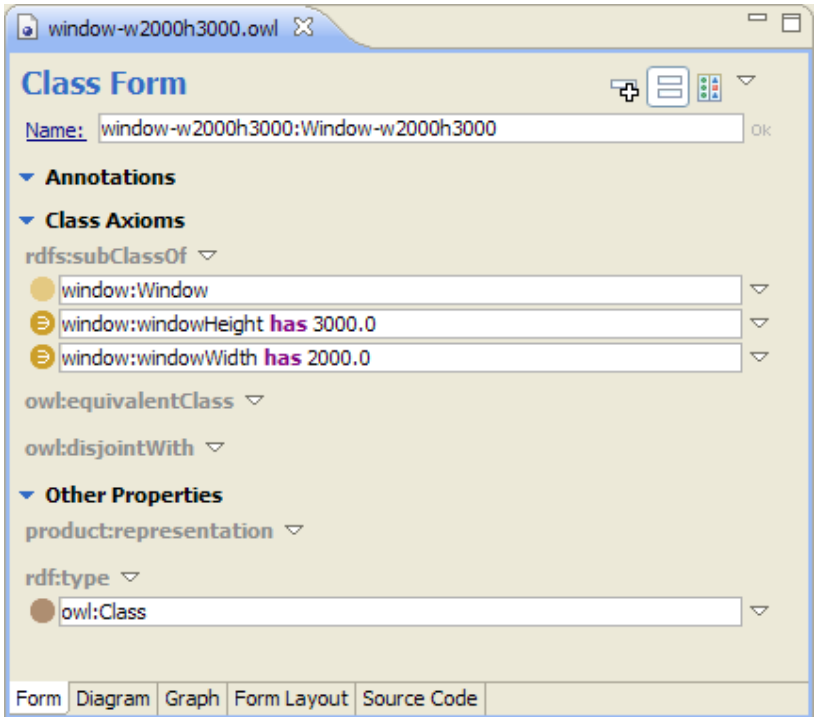


Figure 44

Or in OWL code (extended version of earlier one):

```
<owl:Class rdf:ID="Window-w2000h3000">
    <rdfs:subClassOf rdf:resource="http://www.swop-
       project.eu/ontologies/d23/window.owl#Window"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://www.swop-
           project.eu/ontologies/d23/window.owl#windowWidth"/>
        <owl:hasValue
           rdf:datatype="http://www.w3.org/2001/XMLSchema#float"
        >2000.0</owl:hasValue>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://www.swop-
           project.eu/ontologies/d23/window.owl#windowHeight"/>
        <owl:hasValue
           rdf:datatype="http://www.w3.org/2001/XMLSchema#float"
        >3000.0</owl:hasValue>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
```

Each individual created for this window variant will get these ('defining') class variable values for its width and height (well in principle, a reasoner like the built-in Pellet has or other software like our PMO Configurator has to actually derive this information).

Finally we define extra cardinality constraints for all properties: a Facade has always one height and one width, the same for the window height and width, and colour. So we add “min cardinality = 1” and “max cardinality = 1” restrictions for all properties in the context of their domain class (note that the domain specification says nothing about these min cardinalities!). As already indicated in the earlier shown diagram we also see a “max cardinality = 0” for Window for the hasPart_directly property.

For the Window class we get:

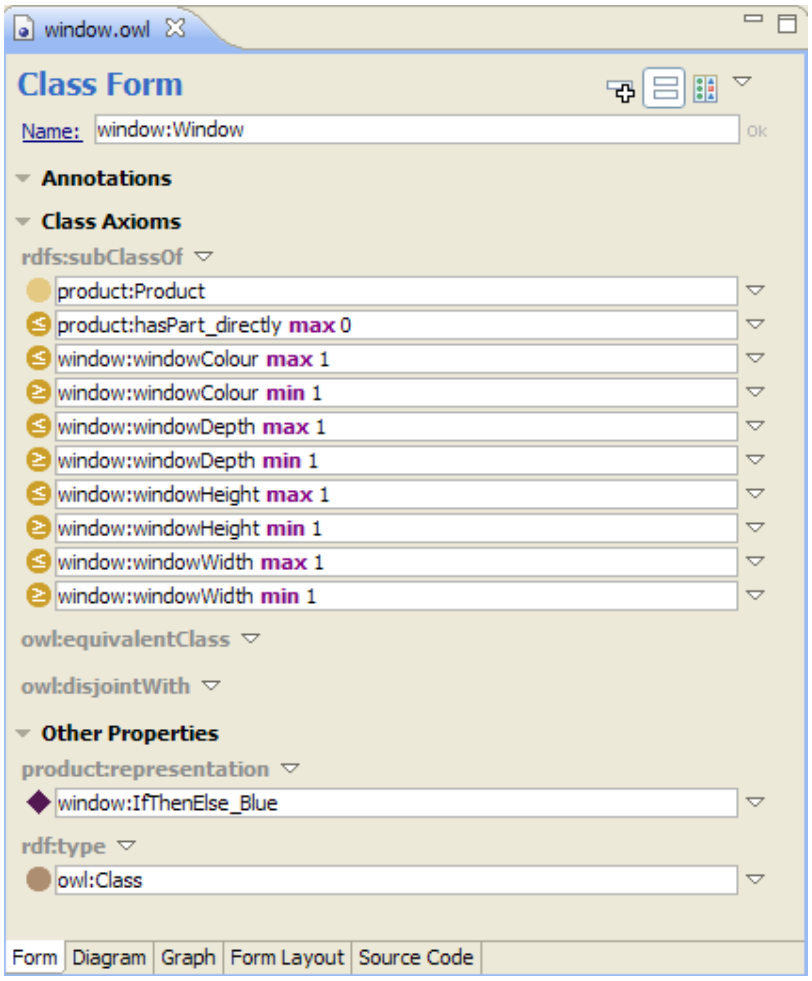


Figure 45

In OWL code the specification is extended towards:

```
<owl:Class rdf:ID="Window">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="windowHeight"/>
        </owl:onProperty>
        <owl:maxCardinality
           rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
```

```
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="windowWidth"/>
      </owl:onProperty>
      <owl:maxCardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="windowDepth"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="windowColour"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >0</owl:maxCardinality>
      <owl:onProperty rdf:resource="http://www.swop-
         project.eu/ontologies/pmo/product.owl#hasPart_directly"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#windowDepth"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="http://www.swop-
     project.eu/ontologies/pmo/product.owl#Product"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#windowHeight"/>
      </owl:onProperty>
      <owl:minCardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#windowWidth"/>
```

```
        </owl:onProperty>
        <owl:minCardinality
           rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
           rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#windowColour"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
```

Similar extensions are added for the Facade class.

STEP 5: Define the rules

We define only one class-level rule (more precisely a 'derivation' rule):

For all facades: facadeHeight = 2/3 * facadeLength.

We define this more easily using our PMO Editor (we defined the rule meta-ontology ourselves so we cannot expect TBC to be of much help here…). After loading the facade.owl ontology we define with it:

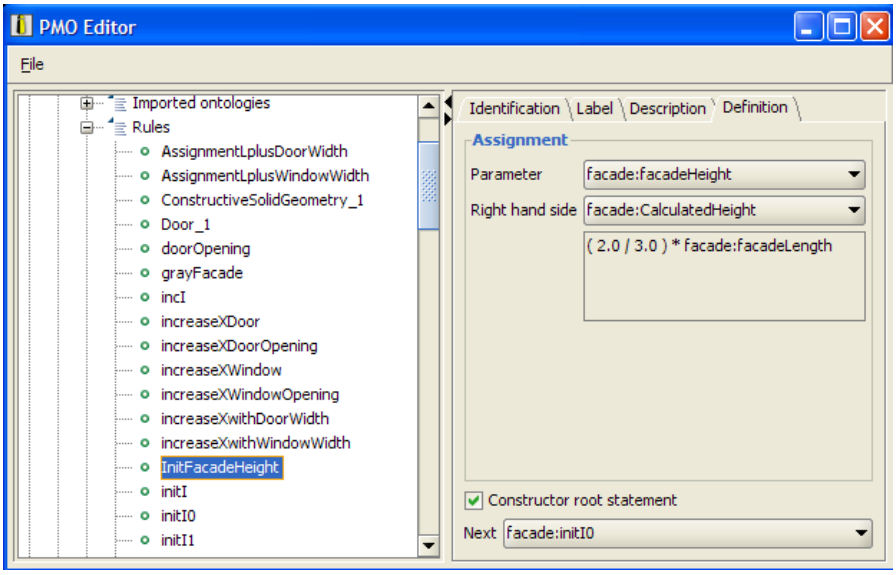


Figure 46

The generated OWL Code:

```
    <rule:Assignment rdf:ID="InitFacadeHeight">
      <rule:rhs>
        <operation:Multiplication rdf:ID="CalculatedHeight">
          <operation:domain_I>
            <operation:Division rdf:ID="TwoThrid">
```

```
            <operation:domain_I rdf:resource="#ValueTwo"/>
            <operation:domain_II rdf:resource="#ValueThree"/>
          </operation:Division>
        </operation:domain_I>
        <operation:domain_II>
          <owl:DatatypeProperty rdf:about="#facadeLength"/>
        </operation:domain_II>
      </operation:Multiplication>
    </rule:rhs>
    <rule:parameter>
      <owl:DatatypeProperty rdf:about="#facadeHeight"/>
    </rule:parameter>
  </rule:Assignment>
```

STEP7: Define the rules for generating an IFC representation

To support the representation of the façade, door and windows we need to define concepts we can use. The file cuboid.owl primarily contains the concept of cuboids that is the basic form for all façades, windows and doors extended with the materials and colours used for this shape. To define the concept of cuboid we used the most small definition possible, for an exact explanation see also 4.5.2.

The materials are available in 4 colors: red, green, blue and gray. The color red is for example defined with only a red color component. All materails only have the ambient color defined.

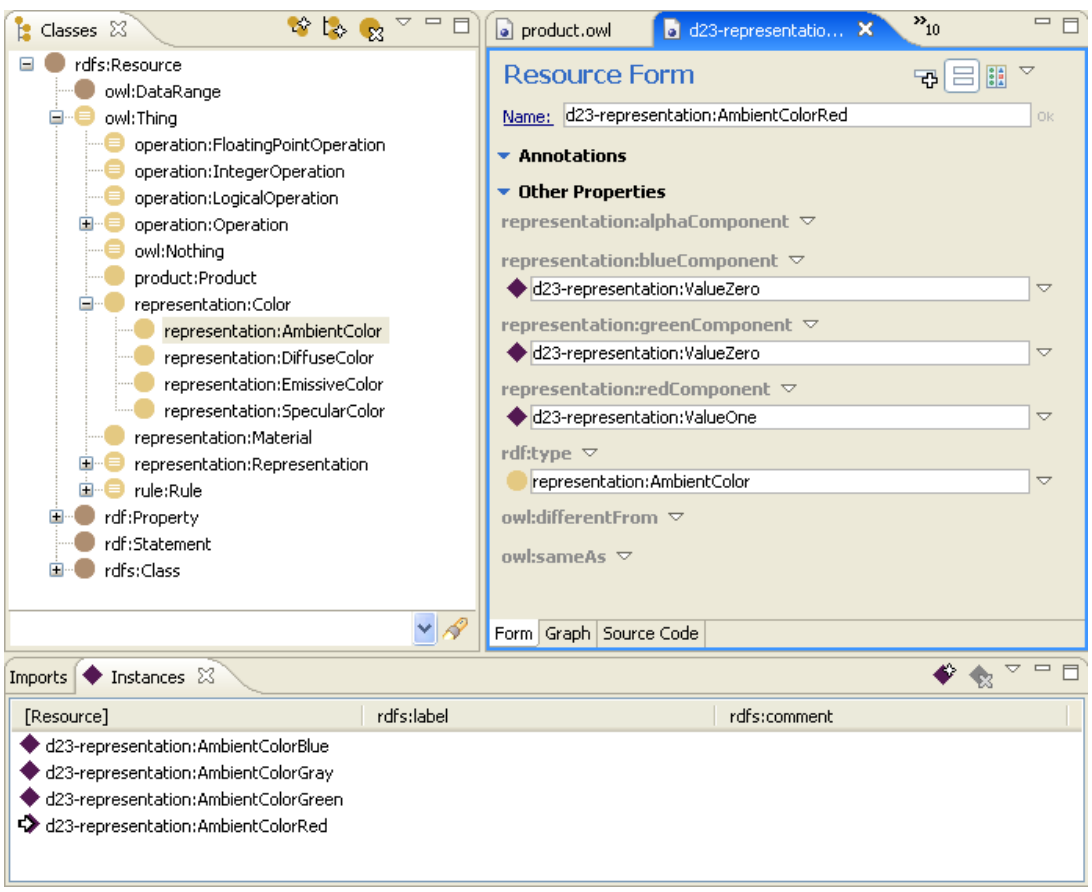


Figure 47

The code for all 4 materials is the following:

```
<operation:Value rdf:ID="ValueHalf">
  <operation:value
    rdf:datatype="http://www.w3.org/2001/XMLSchema#float"
  >0.5</operation:value>
</operation:Value>
<representation:AmbientColor rdf:ID="AmbientColorGreen">
  <representation:blueComponent>
    <operation:Value rdf:ID="ValueZero">
      <operation:value
        rdf:datatype="http://www.w3.org/2001/XMLSchema#float"
      >0.0</operation:value>
    </operation:Value>
  </representation:blueComponent>
  <representation:redComponent rdf:resource="#ValueZero"/>
  <representation:greenComponent>
    <operation:Value rdf:ID="ValueOne">
      <operation:value
        rdf:datatype="http://www.w3.org/2001/XMLSchema#float"
      >1.0</operation:value>
    </operation:Value>
  </representation:greenComponent>
</representation:AmbientColor>
<representation:Material rdf:ID="MaterialRed">
  <representation:ambientColor>
    <representation:AmbientColor rdf:ID="AmbientColorRed">
      <representation:redComponent rdf:resource="#ValueOne"/>
      <representation:blueComponent rdf:resource="#ValueZero"/>
      <representation:greenComponent rdf:resource="#ValueZero"/>
    </representation:AmbientColor>
  </representation:ambientColor>
</representation:Material>
<representation:Material rdf:ID="MaterialBlue">
  <representation:ambientColor>
    <representation:AmbientColor rdf:ID="AmbientColorBlue">
      <representation:redComponent rdf:resource="#ValueZero"/>
      <representation:greenComponent rdf:resource="#ValueZero"/>
      <representation:blueComponent rdf:resource="#ValueOne"/>
    </representation:AmbientColor>
  </representation:ambientColor>
</representation:Material>
<representation:Material rdf:ID="MaterialGreen">
  <representation:ambientColor rdf:resource="#AmbientColorGreen"/>
</representation:Material>
<representation:AmbientColor rdf:ID="AmbientColorGray">
  <representation:blueComponent rdf:resource="#ValueHalf"/>
  <representation:greenComponent rdf:resource="#ValueHalf"/>
  <representation:redComponent rdf:resource="#ValueHalf"/>
</representation:AmbientColor>
<representation:Material rdf:ID="MaterialGray">
  <representation:ambientColor rdf:resource="#AmbientColorGray"/>
</representation:Material>
```

Now the different owl files door.owl, window.owl and facade.owl can import this cuboid.owl and make use of the materials and cuboid concept. A red door can now very simply been defined in the following way

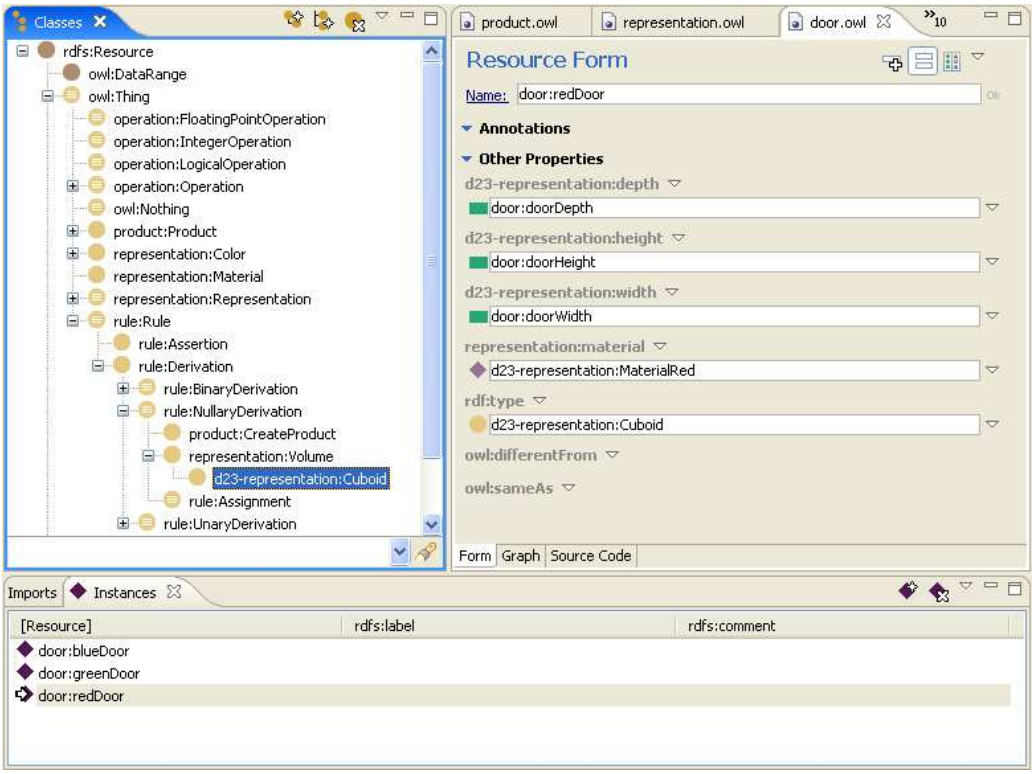


Figure 48

The concept of a Cuboid from cuboid.owl and the red material form this same file can directly be instantiated and refered. The owl code is minimal:

```
<cuboid:Cuboid rdf:ID="redDoor">
  <cuboid:width>
    <owl:DatatypeProperty rdf:about="#doorWidth"/>
  </cuboid:width>
  <cuboid:height>
    <owl:DatatypeProperty rdf:about="#doorHeight"/>
  </cuboid:height>
  <cuboid:depth>
    <owl:DatatypeProperty rdf:about="#doorDepth"/>
  </cuboid:depth>
  <representation:material rdf:resource="http://www.swop-
    project.eu/ontologies/d23/cuboid.owl#MaterialRed"/>
</cuboid:Cuboid>
```

This enables us to model the door, window and façade in a much more natural way. If a more complex form or material should be needed it can be defined separately from the actual product modelling in cuboid.owl.

STEP7: Instantiate the Facade Class into the Facade_01 individual

Now we're finally ready to model the individual facade we started with:

We load our total facade.owl (including quite some representation details not mentioned here) in the PMO Configurator resulting in a default configuration.

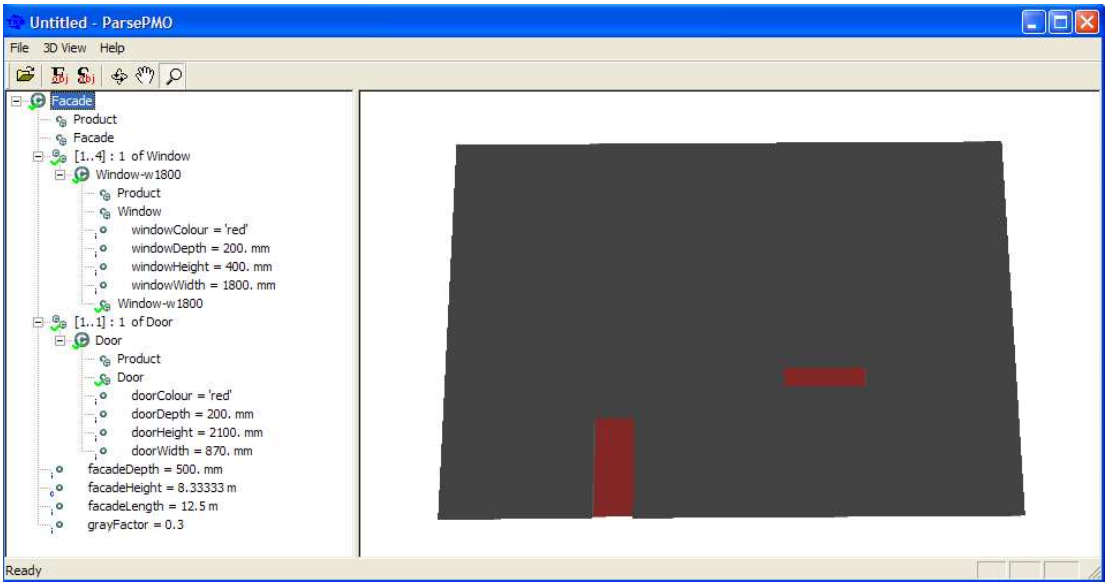
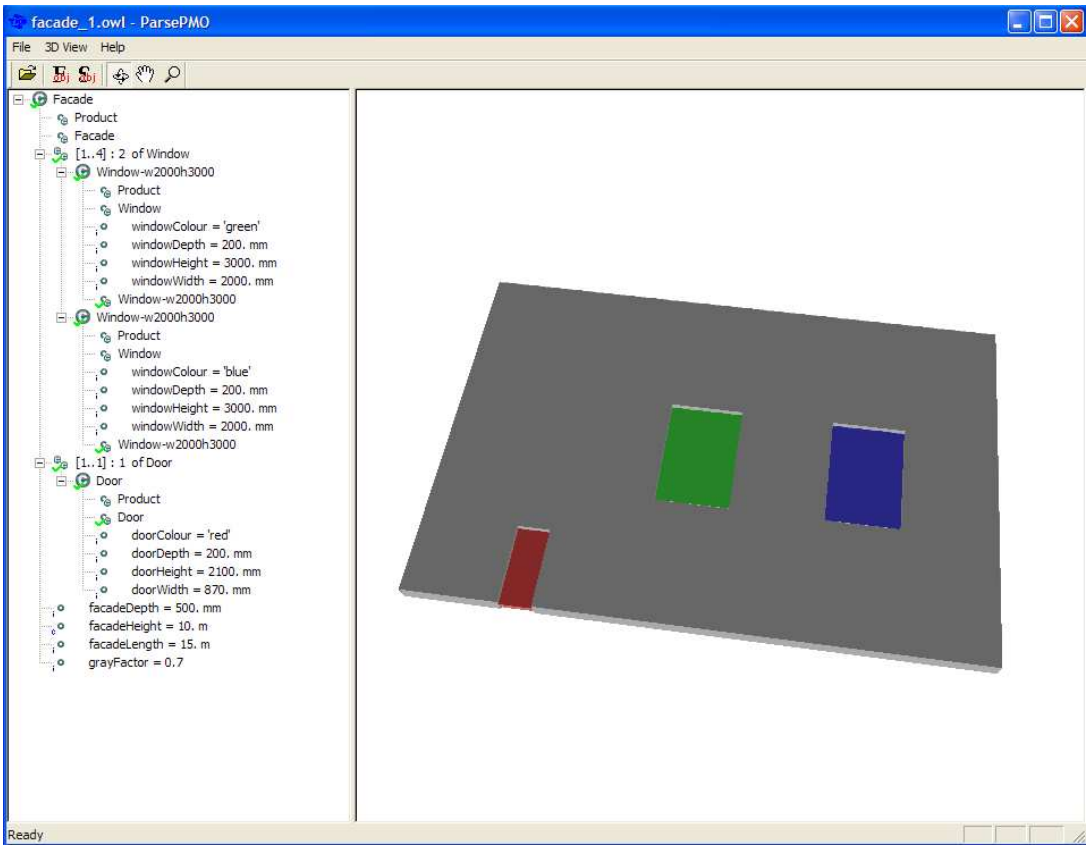


Figure 49

Next we assign the right value for types, cardinalities and properties as above an save it as “façade_1”.



6 ONTOLOGY EDITOR / SERVER

In SWOP we have investigated several 'Semantic Web'-based modelling tools in both a Java and a Microsoft environment. Java seems the most applicable programming language for the SWOP software development. Looking at existing Java-based Ontology Editor / Server tools we identified:

- The OSS Protégé tool with OWL plugin of Stanford University, and
- The non-OSS ('commercial') TopBraid Composer (TBC) tool of TopQuadrant.

A quick comparison can be found in the next figure (indicating pros and cons of TBC with respect to Protégé).

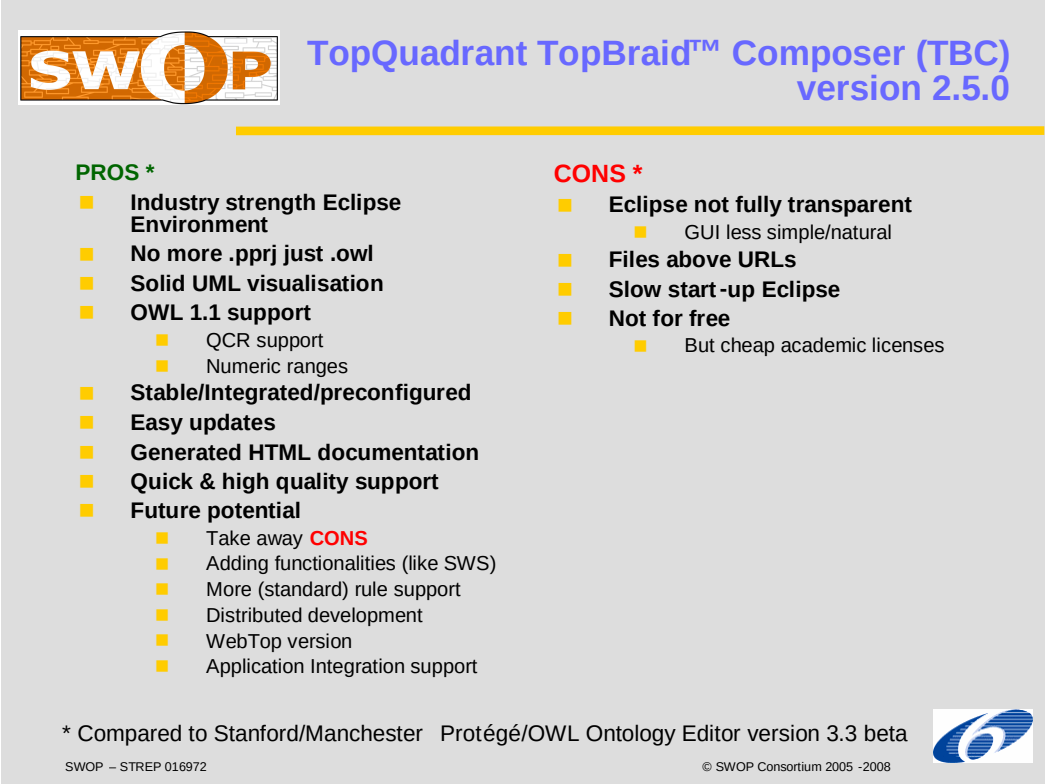


Figure 50: Mainstream Ontology Editors compared

Based on this comparison we recommend SWOP to go for TBC as plug-in to the Eclipse development environment. Although the Protégé is for free and closer to the open spirit of SWOP we foresee to great risk in developing prototype software on top of prototype software. TBC offers less functionality which is however more stable and tightly integrated. Java-based configurators, configurator-generators etc.etc. can be developed in Eclipse interfaced via the standard late-binding Jena OWL-API (by the HP Labs in Bristol) [Jena]. Higher level access can be delivered via Jastor ([Jastor]), an open source Java code generator that generates Java Beans from ontologies enabling type safe access RDF stored in a Jena Semantic Web Framework model. Jastor generates Java interfaces, implementations, factories, and listeners based on the properties and class hierarchies in the ontologies thus providing an early binding OWL-API on top of Jena.

The PMO-API (java-binding) we use ourselves for our tools is based on the Jena API. Our SWOP tool architecture will be based on HP's (late-binding) Jena (java-)API for RDF/OWL. However Jena does not yet support OWL1.1-level QCR's. We will therefore access this info on RDF-level and interprete QCR semantics ourselves in the PMO-API call implementations.

7 REFERENCES & FURTHER INFORMATION

GARM	General AEC Reference Model (GARM), ir. W.F. Gielingh, TNO-report, BI-88-150, TNO Building and Construction Research, October 1988.
Jastor	http://jastor.sourceforge.net/
Jena	http://www.hpl.hp.com/semweb/ & http://jena.sourceforge.net/
Numeric Ranges	http://protege.stanford.edu/plugins/owl/xsp.html
OWL1.1	http://www-db.research.bell-labs.com/user/pfps/owl/overview.html
Protégé	The Protégé open-source, Java tool (Full version 3.3 beta, Build 408) that provides an extensible architecture for the creation of customized knowledge-based applications, including the "OWL Plugin", the Ontology/Knowledge Editor for the Semantic Web. More info at http://protege.stanford.edu/.
QCR	http://www.cs.vu.nl/~guus/public/qcr.html, http://protege.stanford.edu/mail_archive/msg17798.html
Rules	http://www.w3.org/2004/12/rules-ws/
	Especially: http://www.w3.org/2004/12/rules-ws/report/
	http://www.w3.org/2005/rules/wg
	http://www.w3.org/2005/rules/wg/wiki/FrontPage
REWERSE	http://oxygen.informatik.tu-cottbus.de/rewerse-i1/7GR.pdf
SW	The Semantic Web activity, World Wide Web Consortium (W3C), http://www.w3c.org/.
SW BP	http://www.w3.org/2001/sw/BestPractices/
SWOP D13	The SWOP Functional Architecture, SWOP_D13_WP1_T1300_TNO_2007-05-01_v08_revision.doc.
SPARQL	http://www.w3.org/TR/rdf-sparql-query/
Spring	http://www.springframework.org/
TopBraidComposer	TopQuadrant's Ontology Editor/Server (Eclipse plug-in). http://www.topbraidcomposer.com/.
W3C	World Wide Web Consortium, http://www.w3c.org
W3C BP	Beste Practices Working Group, http://www.w3.org/2001/sw/BestPractices/
XML	http://www.w3.org/XML/

APPENDIX A – FULL ONTOLOGIES LOCATION

PMO Ontologies:

http://www.swop-project.eu/ontologies/pmo/product.owl

http://www.swop-project.eu/ontologies/pmo/representation.owl

http://www.swop-project.eu/ontologies/pmo/rule.owl

http://www.swop-project.eu/ontologies/pmo/operation.owl

Typical End-user Product Ontology Example:

http://www.swop-project.eu/ontologies/d23/facade.owl

http://www.swop-project.eu/ontologies/d23/window.owl

http://www.swop-project.eu/ontologies/d23/door.owl

http://www.swop-project.eu/ontologies/d23/window-w2000h3000.owl

http://www.swop-project.eu/ontologies/d23/window-w1800.owl

http://www.swop-project.eu/ontologies/d23/cuboid.owl

http://www.swop-project.eu/ontologies/d23/configurations/facade_1.owl

APPENDIX B – GUIDELINES FOR NAME SPACES & IMPORTS

1. Each .owl file of the end-user product ontology should:

Add name spaces:
Standard xsd, rdf, rdfs and owl name spaces
Pre-standard owl11 name space: xmlns:owl11="http://www.w3.org/2006/12/owl11#
"Own" name space (according to the base url) with a prefix that matches the name of the owl file (except .ext)
xmlns:product="http://www.swop-project.eu/ontologies/pmo/product.owl#"
xmlns:product="http://www.swop-project.eu/ontologies/pmo/representation.owl#"
xmlns:product="http://www.swop-project.eu/ontologies/pmo/rule.owl#"
xmlns:product="http://www.swop-project.eu/ontologies/pmo/operation.owl#"
if relevant the name space of its direct superclass
if relevant the name space(s) of its direct parts

Add imports:
if the class is a root (toplevel) superclass (the other get it via bullit 2…):
<owl:imports rdf:resource="http://www.swop-project.eu/ontologies/pmo/product.owl"/>
if relevant the .owl file of its direct superclass

2. Each .owl file containing individual information should:

Add name spaces:
Standard xsd, rdf, rdfs and owl name spaces
Pre-standard owl11 name space: xmlns:owl11="http://www.w3.org/2006/12/owl11#
"Own" name space (according to the base url) with a prefix that matches the name of the owl file (except .ext)
For all instantiated classes including all classes upward in the specialisation tree their associated name space. See also the second remark below.

Add imports:
For all instantiated classes (only) their associated .owl files. No need for classes upward because they are already imported indirectly (see at 1.).

Remarks:
(1) No need for PMO import and/or names spaces:
Import of product.owl already via end-user product ontology class(es)
Name space not needed
(2) In case name spaces of classes in the path upward are left out TBC adds them (because it imports them, directly or indirectly). It also shows in TBC the right prefixes. But in the file you get added anonymous name space declarations anyway (like p.0) which is not so nice. So we will always add the name spaces ourselves explicitly.

An application of these guidelines can be seen in the typical d23 facade example.

APPENDIX C – PMO / TOOLS ISSUES

Latest PMO Tools at: http://www.swop-project.eu/swop-solutions

- [TOOLS] Change for implementing Tools only
- [PMO] Change for PMO and/or agreed Way of Modelling with PMO

C1. Small Issues

1. Updated ifcXML generation (done by AEC3; to be integrated by PB) [TOOLS]

2. Introduction ifc.owl grouping for (product) properties like ifcEntityType (to control IFC generation) [TOOLS]

3. More coherent approach to combination of material representation and shape representation modelling [PMO/TOOLS]

- things like colours always via material rep., no agreed solution yet

- orthogonal treatment of material/shape reps. (not pre-integrated too much; more flexibility and:

- predefined reps. delivered with PMO (not per end-user ontology) like cuboid.owl

4. Adding proxy/property set/property back-links to library (acc. to PMO or from-PMO-generated ISO 12006-3 SPFF files) [TOOLS]

5. Java3D alternative visualisation for IFC (Bulgaria) [TOOLS]

6. Client/server access [TOOLS]

- PMO-API on server side in Tomcat

- Java C/S-link

7. No saving of 'max card=0'-properties (script variables having no 'final state' meaning) in configurations [PMO/TOOLS]

8. Correct classification of calculated (c) properties in configurator [TOOLS]

- wrong 'i' (should be 'c') when used locally and script executed after externally determined

9. Need for Multiple Composition / Grouping [PMO/TOOLS]

C2. Bigger Issues

1. PMO Browser / PMO Configurator integration [TOOLS]

2. PMO Configurator / GA Engine integration [TOOLS]

- "advise inputs" button" in PMO Configurator

- including agreement on end-user requirements/objectives modelling

C3. Big Issues (mostly beyond SWOP timeframe)

1. Multiple Inheritance Support (multiple superclasses) [PMO/TOOLS]

Currently PMO supports single inheritance that is each class can have just one superclass ('parent'). This way we obtain a relatively simple specialization tree. OWL does however support multiple inheritance. Single inheritance is not hardcoded in PMO but we assume that all our end-user subclasses in the end-user product ontologies have as maximum one superclass. The current PMO supporting software such as the PMO-Configurator, PMO-Browser and the PMO-API reference implementation are based on this assumption. One could want to be able to assign multiple superclasses to a class and inherit properties from all these classes. The subclass becomes a subclass of the intersection between the involved superclasses. Clearly the tree structure is replaced now by a more general directed network. Important subissue is in this scenario the handling of precedence and conflicts.

2. Multiple classification [PMO/TOOLS]

Currently PMO supports single classification that is each class can only have subclasses that are all mutually disjunct. We did model this explicitly in PMO (using the disjunt constraint) but it is assumed to be true for all classes in the end-user product ontologies. In configuration this means that for each subclassing for a class we have to make only (and exactly) one choice. In a multiple classification schema the subclasses don't have to be disjunct: an individual can belong to (be a member of) two or more subclasses ate the same time. In such a way the individual data can be distributed over different tree branches (hence the sometimes used term "complex data"). One can always map multiple classification to single classification by organizing overlapping subclasses in multiple subclass layers. Clearly when you want to subclass a class in 3 different ways involving say an average of 3 subclasses per classification you end up with 3x3x3=27 classes (this way you model with single classification all potential combinations in multiple classification). Multiple classification is on the wish list for a future PMO.

3. Complex properties [PMO/TOOLS]

So far we can only use scalar (rank 0 tensor) datatype properties in PMO. We foresee the future need for higher order tensors (rank 1: vectors, rank 2: matrices) involving different dimensionalities having more complex values then just 1 (i.e. multi-dimensional arrays of values). Currently we support push/pop operations for usage of arrays in scripts (derivation rules).

4. Features [PMO/TOOLS]

Currently we have one PMO root class being ‘Product’. Everything defined in an end-user product ontology is directly or indirectly a subclass of this ‘Product.’ It is argued that we have to distinguish between physical products and non-physical/virtual products. An example of the latter is a Feature like "Hole Pattern". Features are special in the sense that they cannot exist without their "wholes" (they are "existence-dependent on their wholes"). It's an issue for future versions of PMO whether we have to explicitly distinguish between parts and features.

5. End-user (“configuration-time”) variants [(PMO)/TOOLS]

Currently the configuration process takes classes (generic or specific/variants) and generates occurrences/individuals. It could however be useful to be able to define non-standard/end-user/’on-the-fly’ variant classes (that are too specific to model as variants during “engineering-time”) that can be instantiated at “configuration time”.

6. <u>Integrated</u> requirements modelling (PMO/TOOLS]

How can we implement GARM's FUTS (Functional Unit - Technical Solution) approach with PMO. We see the right FUTS handling (especially the Functional Specification (FS) being part of it) as a key success factor for any form of Systems Engineering (SE) / Integral Design (ID).

For now we focus on the resolution for less integrated requirements modelling (related to Issue C.2 issue 2).

7. Local Occurrences: variants with predefined parts [TOOLS]

Currently variant Classes can have certain datatype properties predefined at class level. It would however be convenient to also be able to define predefined parts at this class level. PMO can actually do this (via hasValue for hasParts_directly) but the tools do not support it yet. This way we can efficiently model a fixed Bill of Material (BoM) and reuse it many times for many individuals (like 4 wheels on a car that are the same).

8. Part in whole info

For both class-level (see issue 7) and individual-level part definitions we often have the need to attach information to the part in the context of its whole. In essence we have a need to objectify the hasParts relationship so that we can add extra info. Currently we do this by introducing extra classes (like HorizontalAxis for the Blum case). This approach only works when option are small (certainly finite) such as vertical or horizontal. Otherwise we have to introduce a more generic class and attach the info not in the class name but as an explicit datatype property for that new class. For the future we might however define something on PMO level (i.e. a mechanism in the language) for this.

APPENDIX D – LOGICAL OPERATION TYPES

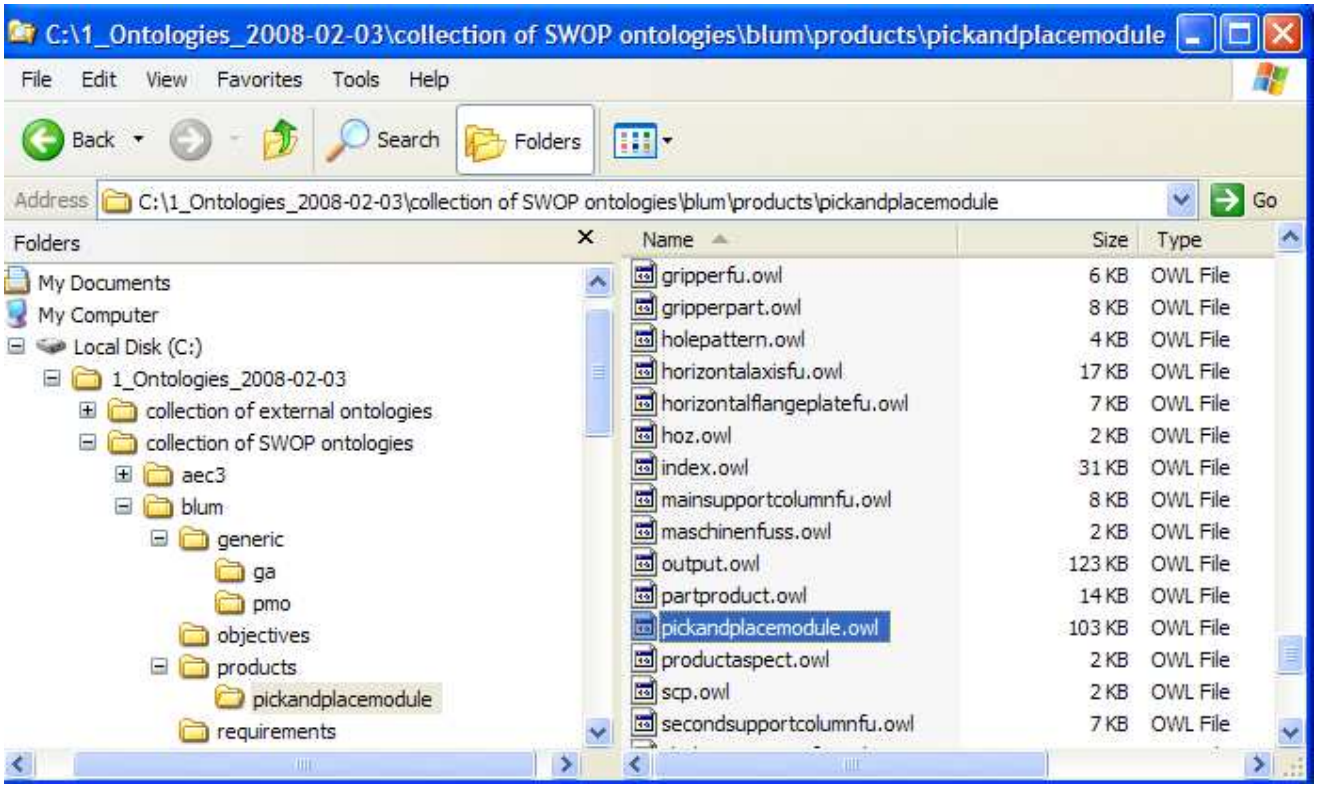
16 Logical Operation Types

X	Y	AND (X, Y)	OR (X, Y)	XOR (X, Y)	IFTHEN (X, Y)	IFTHEN (Y, X)	X	Y	true
0	0	0	0	0	1	1	0	0	1
1	0	0	1	1	0	1	1	0	1
0	1	0	1	1	1	0	0	1	1
1	1	1	1	0	1	1	1	1	1
nullary							x	x	x
unary(nullary)									
binary(nullary, nullary)		x	x	x	x	x			
unary(binary(nullary, nullary))									

X	Y	NOT (AND (X, Y))	NOT (OR (X, Y))	NOT (XOR (X, Y))	NOT (IFTHEN (X, Y))	NOT (IFTHEN (Y, X))	NOT (X)	NOT (Y)	false
0	0	1	1	1	0	0	1	1	0
1	0	1	0	0	1	0	0	1	0
0	1	1	0	0	0	1	1	0	0
1	1	0	0	1	0	0	0	0	0
nullary									x
unary(nullary)							x	x	
binary(nullary, nullary)									
unary(binary(nullary, nullary))		x	x	x	x	x			

* Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

APPENDIX E – LOCAL FOLDER/FILE STRUCTURE EXAMPLE



* Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)