# PROPERTY COMPOSITION AND OTHER CONSTRUCTS TO SUPPORT IDENTITY IN OWL

Robert M. Colomb
Department of Information Systems
Faculty of Computer Science and Information Systems
University of Technology Malaysia
colomb@utm.my

**Abstract:** This paper presents proposals for augmenting OWL Full with constructs necessary for supporting identity, an essential facility for ontologies governing interoperating information systems. These constructs include declaration that a property is one-to-one, a representation of the image of a property, and composition of properties. Using definitions from Category theory, property composition allows definition of Cartesian product without requiring that an individual have any internal structure. Category theory-style definitions and reasoning work entirely with properties, therefore are not affected by the open world semantics of OWL. The definitions permit representation of n-ary relations, association classes, parameterised families of properties and indexed families of classes.

**Keywords**: OWL, information systems interoperation, ontology

## 1. INTRODUCTION

OWL as a knowledge/ontology representation language has a rich set of constructs, but is very limited in expressive power compared for example with Common Logic or even SQL. There are many applications for which the expressive power of OWL is insufficient. But extensions to OWL are best if they respect the basic structure of OWL, founded on classes, properties, and the absence of the unique names assumption. This paper argues for a number of extensions based on property composition, which it argues fit well into the OWL way of doing things.

OWL has a set of base constructs: *class* and *property*. Properties have *domain* and *range*. There are special kinds of properties: *functional*, *inverse functional*, *transitive*, *symmetric*. In addition, it is possible to construct restriction classes using a number of conditions on properties (*allValuesFrom*, *someValuesFrom*, *hasValue*, *minCardinality*, *maxCardinality*). There is a limited ability to construct derived properties using *inverseOf*. Further, there is a limited facility for property composition, in that composition of a transitive property results in the same property; and a limited algebra of operators on properties: *inverseOf* applied to *inverseOf* is the identity, and *inverseOf* is the identity on a symmetric property. Finally, in OWL individuals by default fail to satisfy the unique names assumption, but properties and

classes by default do satisfy the assumption. Of course classes can be declared to be equivalent, as can properties, but they are by default assumed to be distinct.

In this paper, we argue for a more complete facility for property composition, a richer facility for construction of classes, and the use of aggregation for deriving classes and properties. Some of the proposals are informed by category theory, which suggests definitions that do not require instance-level reasoning so are compatible with the absence of the unique names assumption. All of the proposals are informed by well-established requirements of large information systems, and in some cases borrow from facilities of SQL designed to serve these requirements.

We first consider the problem of identity, which requires that properties be declared to be injective and surjective, and also the construction of derived classes using Cartesian product. These lead to the need for property composition, which allows the definition of Cartesian product without requiring a class to have internal structure, instance-independent definitions of surjection and injection, and a class of integrity constraint called the principle of consistent dependency useful in information systems applications. These new constructs are applied to the representation of n-ary properties, to association classes and to correlations among classes. They are further applied to structures useful for large ontologies, namely parameterised families of properties and indexed families of classes. Finally, a proposal is made for declaration of local closed worlds permitting the definition of aggregation functions used in data warehousing applications. The proposals are finally summarized in a table, and the issue of placement in OWL Full rather than OWL DL is addressed.

## 2. IDENTITY

An ontology is a representation of a world shared among a community of agents. This world consists of a collection of objects organized into classes and related through properties. A key requirement for two agents to interoperate is the ability to tell if they are working with the same object. In other words they need to be able to agree on the identification of the objects they share. Consider the situation in Figure 1.

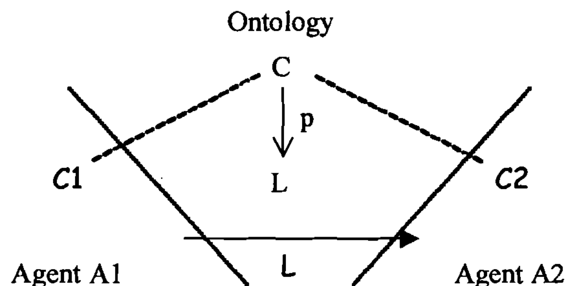


Figure 1. Interoperating Agents

The ontology consists of a concept *C* which is the in the domain of a property *p* whose range instance is a single literal *L*. The ontology supports a group of interoperating agents, including agents *A1* and *A2*. Each agent maintains an internal representation of concept *C* as *C1* and *C2* respectively. But *C1* is not visible to any observer outside *A1*, and similarly *C2* is not visible to any observer outside *A2*.

The agents communicate using messages constructed out of literals. So if agent *A1* wants to communicate with *A2* about concept *C*, the message will contain the literal *L*. *A1* finds *L* by navigating from *C1* to *L* via *p*, while *A2* must find *C* by navigating from *L* to *C2*. This implies a property whose range includes *C* and whose domain includes *L*. The property *p* carries identity for the class including *C*.

OWL has a very weak concept of identity. Individuals can have names, but names do not necessarily satisfy the unique name assumption. Names may be synonymous. The fact that two names are synonymous may not be known, so two distinctly named individuals may not necessarily be distinct objects.

Literals do satisfy the unique name assumption, in that for example two distinct strings are different. If there were a one-to-one property whose domain was a class of individuals and whose range was a class of literals, it could be identifying. But it is impossible to declare such a property in OWL DL. One can declare a property to be functional, and that its inverse is also functional, but a property cannot have a class of literals as its domain. OWL Full allows the declaration of a property whose range can be a class of literals to be both functional and inverse functional, but there is still no property whose domain is a class of literals.

There are many applications like Figure 1 where a community of autonomous agents interoperate in a closed world. Electronic commerce exchanges are an obvious example. These can have tens of thousands of autonomous players, but they must all be members of the exchange and must all commit to the ontology. These agents need identity, and the representations they have to work with are restricted to collections of literals.

Besides its use in the interoperation of information systems, identity is a central concern of the OntoClean [4] method for validating the subclass structure of an ontology. The properties used to identify instances of a superclass must also be capable of identifying instances of all subclasses.

SQL systems represent identity by allowing columns of tables to declared keys. Entity-Relationship models and UML models have cardinality/multiplicity constraints on all ends of relationships/associations, so in particular can declare one-to-one constraints.

An equivalent facility in OWL would be the inclusion of an additional restriction class on a property, namely the subset of its domain which is in one-to-one correspondence with a subclass of its range. We might call this new restriction *isOneToOneTo*. But such a restriction

class is different from other restriction classes in OWL. The other restriction classes are all predicates which can be satisfied or refuted by a single individual, so that it is possible to compute the restriction class by testing all individuals, putting those satisfying the predicate into the restriction class, and leaving the others out. The proposed restriction *isOneToOneTo* can't be computed this way.

### 2.1 Restrictions: Injection and Surjection

The restriction *isOneToOne* is more naturally evaluated by testing the individuals in the range of the property. Each instance of the range corresponding to a single instance of the domain determines an instance of the domain belonging to the restriction class. So *isOneToOneTo* can be defined as a restriction for an *objectProperty*. But we would also like *isOneToOneTo* to apply to a *datatypeProperty*. This is more difficult, because literal datatypes are generally not finite classes.

OWL has a construct owl:DataRange, which is defined as a subclass of literals. The only constructor for owl:DataRange is the ability to construct an enumerated class of literal instances using owl:oneOf. A *datatypeProperty* whose range is an owl:DataRange can support our definition of *isOneToOneTo*.

A possible representation in RDF of *isOneToOneTo* (restriction(ID isOneToOneTo (required))) along the lines of [8] is

```
_:x rdf:type owl:Restriction .
_:x rdf:type owl:Class [opt]
_:x rdf:type rdfs:Class [opt]
_:x owl:onProperty T(ID)
_:x owl:isOneToOne T(required)
```

*ID* and *required* are parameter variables, *ID* a property and *required* a class. The [opt] annotation means that the OWL reasoner can infer this (a *Restriction* is by definition an OWL class, which is by definition an RDFS class.) The notation T() refers to a translation of the parameter to an RDF object.

A *DataRange* defined using owl:oneOf has an extent defined a priori and explicitly. This is inconvenient for many common situations. For example, suppose we have a *datatypeProperty studentID* whose domain is *Student* and whose range is xsd:string, and want to know which strings are actually associated with instances of *Student* in that class' extent at a particular time. To this end, we propose a derived class *imageOf class C from property P*, or *C imageOf P*

```
    datarange(imageOf(required) onProperty(ID))
      _:x rdf:type owl:DataRange
      _:x rdf:type rdfs:Class [opt]
      _:x owl:onProperty T(ID)
      T(ID) rdf:type owl:datatypeProperty
      T(required) rdf:type owl:Class
      T(ID) rdfs:domain D
      T(required) rdfs:subclassOf D
      _:x owl:imageOf T(required)
and class(imageOf(required) onProperty(ID))
      _:x rdf:type owl:Class
      _:x rdf:type rdfs:Class [opt]
      _:x owl:onProperty T(ID)
      T(ID) rdf:type owl:objectProperty
      T(required) rdf:type owl:Class
      T(ID) rdfs:domain D
      T(required) rdfs:subclassOf D
      _:x owl:imageOf T(required)
```

So if we have a property *P* with domain *D* and range *R*, and want to say that *P* is injective to a subclass *T* of *R* on a subclass *S* of *D*, we can first declare

```
      I imageOf S onProperty P
then
    T rdfs:subclassOf I
then the restriction
    isOneToOneTo T onProperty P
and finally
    S rdfs:subClassOf restriction
```

An identifier for *S* must also satisfy the restriction *cardinality* = *1*, since every instance of the domain class must be identified.

We now have a proposal for OWL constructs supporting the restriction that a property be injective, and hence an identifier. But along the way we have also proposed a construct that can be used to declare that a property is surjective, since a property is by definition surjective if it is range restricted to any subclass of its image.

## 2.2 Cartesian Product

But a single property is often not sufficient for identification. Relational systems make great use of compound keys reflecting the need to identify objects in particular applications by combinations of attributes. For example, in RDF, a non-reified statement in a named graph is identified by the combination of four properties *subject*, *predicate*, *object* and *graph.* So we need to be able to designate a combination of properties as carrying identity for a class.

One way to do this, analogous to the way compound keys are handled in relational systems, is to allow a new class to be derived as the Cartesian product of a number of existing classes.

In relational systems, a Cartesian product is defined as a set of tuples, but in category theory (see eg [1]), a Cartesian product is defined without specifying the internal structure of the derived class. Instead, the construction includes a collection of total functional projection properties whose domain is the Cartesian product and whose range is one of the classes from which the product is derived. This latter approach is probably more suited to OWL, since individuals in OWL have no internal structure.

If a collection of properties is together identifying for a class, it is equivalent to a single property whose range is the Cartesian product of the ranges of the original properties. The original range classes can be reached using the projection properties.

Cartesian product can be defined in a way analogous to the definition of class intersection in [8], using blank nodes. First we define the product itself

```
        cartesianProduct(description1...descriptionn)
          _:x rdf:type owl:Class
          _:x rdf:type rdfs:Class [opt]
          _:x owl:cartesianProduct T(SEQ description1...descriptionn)
then the projections
        projection(description, descriptioni) i = 1...n
          _:x rdf:type owl:functionalProperty
          _:x rdf:type rdf:Property [opt]
        _:x owl:projection T(SEQ description, descriptioni)
        _:x rdfs:domain T(description)
        _:x rdfs:range T(descriptioni)
        T(description) rdfs:subClassOf T(restriction(_:x, minCardinality(1)))
```

where the first *description* is the product and *descriptioni* is one of the components of the product. The projection property has minimum cardinality 1 on the product.

This definition is somewhat ambiguous, but will be augmented below.

### 3. Composition of Properties

A property in OWL associates each individual with zero or more individuals. So if we have two properties *P1* and *P2*, with *P1* associating individual *I1* with *I2* and *P2* associating *I2* with *I3*, then the composition *P2.P1* associates *I1* with *I3*, and is semantically a property in OWL. It would seem reasonable therefore for OWL to have a syntactic mechanism for property composition. We propose the construct *composedWith* for this purpose, where *P2 composedWith P1* = *P2.P1* as above. The construct *composedWith* can be defined in the same sort of way as *intersectionOf* and *unionOf*, connecting the construct to the type rdf:Property via a blank node.

_:x rdf:type owl:ObjectProperty .
_:x rdf:type rdf:Property . [opt]
_:x owl:composedWith P(SEQ property1...propertyn)

where *property1...propertyn* are all object properties.

We need rules to derive the domain and range of a property composition given the domains and ranges of the properties composed. Since property composition is associative, we need only look at the case of composition of two properties, *P1* and *P2*. Let *D1*, *D2* be respectively the domain of *P1*, *P2* and similarly *R1*, *R2* the range of *P1*, *P2*. Let *P* be *P1* composed with *P2*, with domain *D* and range *R*. Clearly,

D = D1

Further,

R = imageOf (R1 intersect D2) onProperty P2

Property composition is a very useful tool. One very common use is to present aspects of an ontology to a player in a form appropriate to their participation in the application supported by the ontology. In relational database implemented information systems, this facility is called a *view*. For example, suppose we have a class *Wine* which is the domain of a property *availableFrom* whose range is *WineStore*, and that *WineStore* is the domain of a property *situatedIn* whose range is *Suburb*. If we have a menu planning application, it might want only to know which suburb a wine is available in, abstracting away from the stores. We can create a property *availableIn* with domain *Wine* and range *Suburb* as an *equivalentProperty* to *situatedIn composedWith availableFrom.*

A second kind of use for composition is constructing derived properties in semantically dense networks of base properties. A very familiar example is family relationships. If we have *parentOf* we can define *grandParentOf* as an *equivalentProperty* to *parentOf composedWith parentOf.* If we have *childOf* as *inverseOf parentOf*, we can define *siblingOf* as *equivalentProperty* to *childOf composedWith parentOf.* Similarly for uncles, cousins, in-laws and so on.

Note by the way that the construction of *composedWith* results in a blank node, which can be named with *subPropertyOf* as well as *equivalentProperty*, so that for example recursive properties can be defined.

### 3.1 Application to Cartesian Product

We can complete the definition of Cartesian product of classes with a more technical use of property composition. As noted above, in category theory, Cartesian product of classes is defined without reference to any internal structure of the product class by defining the product class in terms of properties. Class *C* is the product of classes *A* and *B* with projections

respectively *pa* and *pb* if given any properties *a* and *b* with domain *D* and range respectively *A* and *B*, there is a unique property *c* with domain *D* and range *C* such that *a* is *c* composed with *pa* and *b* is *c* composed with *pb*. It is customary to illustrate this with a diagram, as in Figure 1.

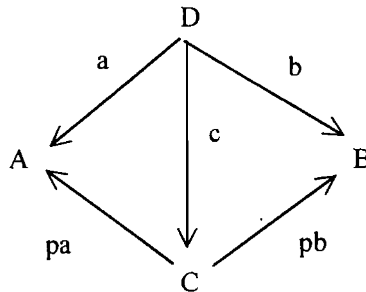


Figure 1 Diagram of Cartesian product

In this diagram, it makes sense to think of property *c* as the product of *a* and *b*. If *a* and *b* together carry identity for *D*, then we can say that *c* carries the compound identity.

### 3.2 Principle of Consistent Dependency

That for example *pa composedWith c* is *equivalentProperty* to *a* is called in category theory that the diagram *commutes*. That two property compositions with the same domain and range commute gives an important kind of integrity constraint called *principle of consistent dependency* by Dampney (see [2]). For example, suppose our menu planner creates a plan for a client. The client is located in a suburb via a composition of properties, and the wines are available in a suburb via a composition of different properties. We might want to stipulate that the wines must be available in the same suburb that the client lives in. Declaring that the two compositions commute (in OWL terminology that the two composed properties are *equivalentProperty*) expresses that constraint.

Situations where the principle of consistent dependency is a useful integrity constraint are extremely common: we want a delivery of a product in respect of an order to be the same product as that ordered; we want the institution granting a degree to a student to be the same institution as offered the courses the student takes; we want the destination of a flight taken with a ticket to be the same as the destination on the ticket. That the constraint is seldom made explicit is partly due to lack of a language to do so, and partly to the fact that these sorts of constraints are taken for granted in single information systems. Where systems are built using cooperating autonomous agents, failures are more likely and have greater consequences. It is therefore more important that commitment to the ontology include commitment to constraints like these.

### 3.3 Reasoning on Properties Only: Monomorphisms and Epimorphisms

Notice that the property *c* in Figure 1 is defined without regard for any instances of either *D* or *C*, unlike our earlier definitions of injective and surjective properties. This is an advantage, since the default open world semantics for OWL makes reasoning at the individual level problematic. For example, if we have a restriction on property *P* that *cardinality = 1*, and have the triples

i1 P j1
i1 P j2

we can't conclude that the restriction is violated unless we further know

j1 differentFrom j2

similarly, we can not conclude from the absence of a triple whose subject is *i1* and whose predicate is *P* that the constraint is violated. We simply may not know the object of the triple, so can't represent it.

In Category Theory, there is a generalization of injective, called monomorphism, and a generalization of surjective, called epimorphism, defined solely in terms of commuting diagrams. We will refer to Figure 2

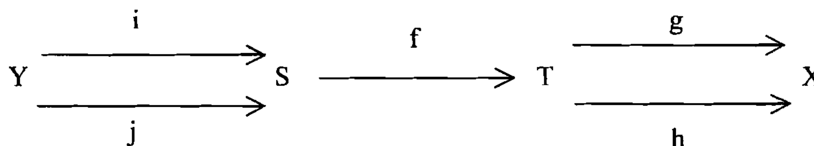


Figure 2 Diagrams for monomorphism and epimorphism

A property *f* with domain *S* and range *T* is a *monomorphism* (injective) iff for every properties *i* and *j* with domain *Y* and range *S*, *equivalentProperty(f composedWith i, f composedWith j)* implies *equivalentProperty(i, j)*.

Similarly, a property *f* with domain *S* and range *T* is an *epimorphism* (surjective) iff for every properties *g* and *h* with domain *T* and range *X*, *equivalentProperty(g composedWith f, h composedWith f)* implies *equivalentProperty(g, h)*.

These definitions may be more convenient for reasoning in OWL engines.

## 4. Application of the Constructs: N-Ary Relations and Association Classes

Suppose we wanted to say that Ian Thorpe won a gold medal for the 400 metre freestyle in the 2004 Olympics. This is easily expressed as a 4-ary relation

<Ian Thorpe, 2004, 400m Freestyle, gold>

UML, Object Role modelling, and many dialects of Entity-Relationship modelling have constructs permitting this to be directly expressed (respectively n-ary association, n-ary fact type, n-ary relationship).

Further, it is often convenient to consider that something modelled as a binary relation has an attribute, or participates in a relation. For example, if we represent the fact that a student is enrolled in a course as a 2-relation <Student, Course>, it makes sense to represent the students' grades as also a 2-relation <<Student, Course>, Grade>. Or we might want to represent that a student's enrolment is managed by a Faculty. UML has a facility called association class to model this representation, and Object-Role modelling has a facility called reification of fact types. It is also supported by some dialects of ER modelling.

The two can be combined. We can represent that a person is booked on a flight on a date as a ternary association, and represent the seat allocated as an attribute on that association seen as an association class.

On the other hand, many modelling and knowledge representation tools do not provide one, or indeed either, of these facilities. In particular, OWL provides neither.

It happens, though, that the machinery already introduced provides a good facility for representing both. First, the association class, as in Figure 3. The Figure uses a UML-style notation. The rectangles denote classes, the arrows properties with range at the end with the arrowhead.

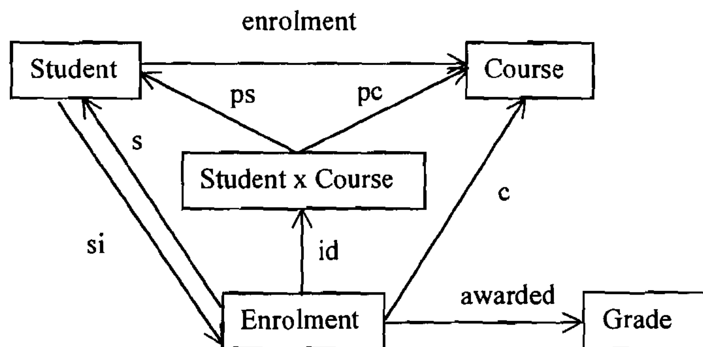


Figure 3: Example of association class in augmented OWL

The original property is *enrolment.* We represent it as the class *Enrolment*, identified by *id* whose range is the Cartesian product *Student x Course* with projections *ps* and *pc*. The property *s* is *equivalentProperty* to *ps composedWith id*, *c* is *equivalentProperty* to *pc composedWith id*, and *si* is *inverseOf s*. We can define *enrolment* as *equivalentProperty* to *c composedWith si*. Finally *Enrolment* is the domain of the property *awarded.*

Note that we need never materialize the Cartesian product nor the projections *ps* and *pc*. It is used solely as a constraint on the triples representing the properties *s* and *c*.

N-ary properties can be represented in a similar way, as illustrated in Figure 4. The notation is the same as Figure 3, with the addition of the arrow with a solid head denoting *Outcome* rdfs:subClassOf *CxExOxM*. To avoid clutter, the projection properties are not shown.

The quaternary property is represented by the class *Outcome*. As a subclass of *CxExOxM*,

*Outcome* inherits the projection properties. *Outcome* can be identified in two different ways. Given *Event*, *Olympiad* and *Competitor*, we know *Medal*. Given *Event*, *Olympiad* and *Medal*, we know *Competitor*. These identifications are represented by the two properties *id1* and *id2*.

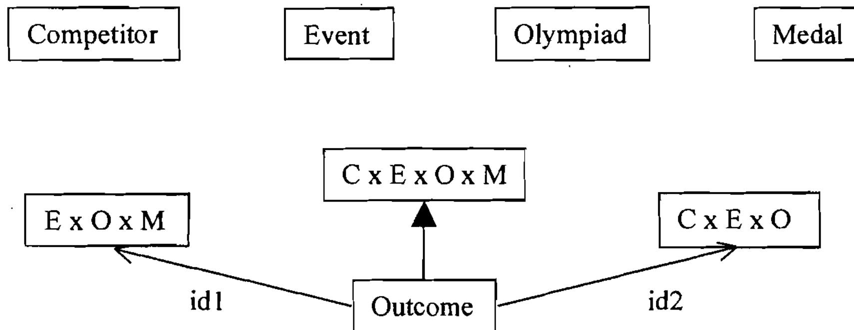


Figure 4: Example of n-ary property

## 5. CORRELATION AMONG CLASSES

Suppose we have two classes, each a subclass of the domain of a property whose range they share. There are often constraints on the ontology involving correlation of instances of the domains as they are associated with instances of the range. For example, we might be a motor vehicle service establishment with an ontology fragment as shown in Figure 5. *WorkItem* is a charge for labor of a particular specialist, while *PartItem* is a charge for parts. In both cases the class *Product* has price, tax rate and other such general properties. We want there to be no instance of *Product* which the object of both *laborFor* and *partFor*. That is, no *WorkItem* refers to a part and no *PartItem* refers to a kind of labor.

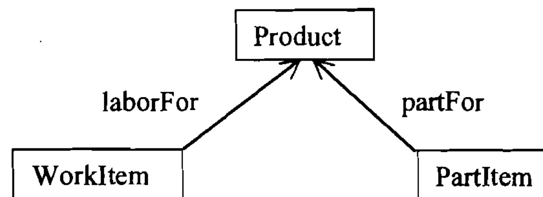


Figure 5: Ontology fragment where range instances not shared

On the other hand, there are applications where we want every instance of the range to be the target of links from both domains, as in the ontology fragment shown in Figure 6. Here, *Skill* is a class of specific skill needed for problem solving. *Problem* is a class of standard problem situations, and *neededFor* links problem situations to the skills needed to solve them. *Staff* is a class of staff members. Each staff member has some skills. We want the constraint that for each skill needed for a problem there is at least one staff member who has that skill,

and that staff members only record skills potentially needed for solving standard problem instances.

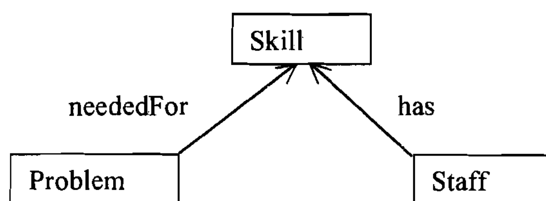


Figure 6: Ontology fragment where all range instances are shared

Finally, we might be interested in the subclass of skills for which there is both a problem and a staff member, not a constraint but what amounts to a query.

We can define these constraints (or create these classes) by intersection of images, in the same sort of way that we can intersect restriction classes. In the first example, if the images of *laborFor* and *partFor* are disjoint, then no instance of *Product* is linked to both a *WorkItem* and a *PartItem*. This can be made a constraint by defining the intersection as a subclass of *Nothing*, the built-in empty class.

That every instance of *Skill* is *neededFor* an instance of *Problem* and some instance of *Staff has* that skill can be specified as a constraint by defining *Skill* as a subclass of the intersection of the images of *neededFor* and *has*.

Finally, the query is a defined class obtained by declaring the intersection of the images to be *equivalentClass* to a desired class name.

### 6. PARAMETERIZED FAMILIES OF PROPERTIES

N-ary associations can be conveniently represented as classes, as in Figure 4. But note that it is possible to derive binary properties from the n-ary, as in Figure 7. The property *competedIn equivalentProperty po composedWith ci*, with *ci inverseOf pc*, is a perfectly normal property, associating every instance of *Competitor* with the instances of *Olympiad* in which they competed in any event. It works because *pc* is a total property, so its inverse is surjective. Any of the pairs of classes can be associated with a derived property of this kind.

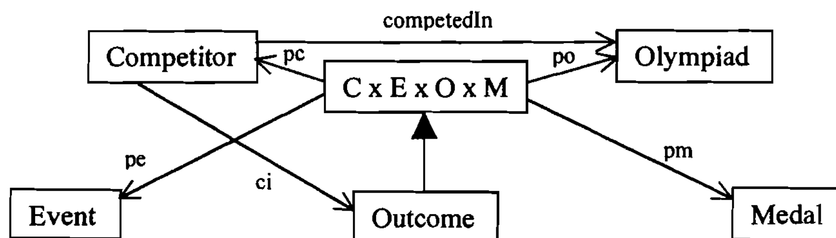


Figure 7. Binary property derived from quaternary association

So n-ary associations can be viewed as properties in a number of ways, but these

associations are complex and often need to be analysed. For example, we might want to restrict attention in Figure 4 to the outcomes in a particular event, say Mens' 400 metre freestyle, so we can see the medals awarded to competitors in Olympiads for just that event. We can do this by creating a restriction class *hasValue* the particular event, in the normal way.

We might then want to look at the result using properties. For example, we might want the Olympiads in which a competitor won a particular medal, say gold. Creating a further restriction class *hasValue* gold and intersecting the two results in an association class as shown in Figure 8 which has only two non-trivial projections, on *Competitor* and *Olympiad.* (The other two are fixed by the restriction classes.)

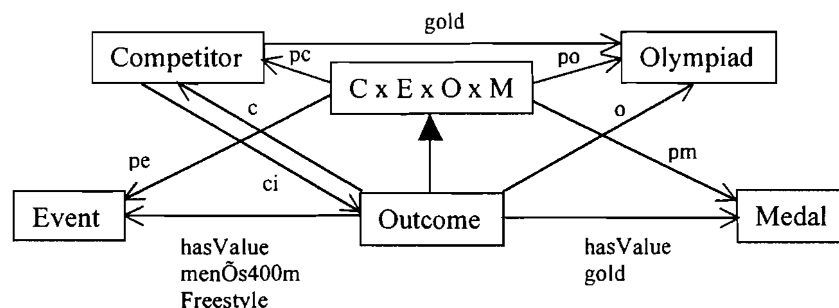


Figure 8 Quaternary association of Figure 4 reduced to a parameterised property

The notation we have been using for the diagram begins to fail us, but Figure 8 is intended to read that the domain of the properties *c* and *o* is the intersection of *Outcome* with the two restriction classes. The property *ci* is *inverseOf c*, and *gold* is *o composedWith ci.* The property *gold* is therefore the property sought, linking instances of *Competitor* to instances of *Olympiad* in which the competitor won a gold medal (for the nominated event).

Notice that we can create a property corresponding to each of the (in this case three) instances of *Medal.* Each of these is a subproperty of the property *competedIn* of Figure 7, indexed by instances of *Medal.* Further, since OWL Full allows objects to be both Individuals and Properties, there is nothing to stop us using the URI of the medal instance to name the subproperty, as has been done in the diagram.

Note that we can use the family of subproperties indexed by *Medal* for the result of any restriction on *Event* in the diagram, since the result is to associate competitors with *Olympiads* in which they have been awarded medals of the kind named by the property. So the subproperty of *competedIn* indexed by the *Medal* instance *gold* can be analysed further into a family of subproperties indexed by instances of *Event.* Of course the naming trick cannot be repeated since the URI notation does not support an algebra of combinations.

## 7. INDEXED FAMILIES OF CLASSES

If indexes families of properties can be useful, so can indexed families of classes. Many ontologies have very large numbers of classes organized into deep hierarchies of subclasses. Probably the largest ontology is the Linnean system for organizing biological species, of which there are many millions. This system is complicated, but has a basic organizing principle that there is a small number of most general superclasses which are indexed by the term *kingdom*. Each kingdom is divided into a number of direct subclasses, all of which are indexed by the term *phylum*. Phyla in turn are subdivided into direct subclasses indexed by *class*, each class by *order*, each order by *family*, each family by *genus* and each genus by *species*. Wikipedia gives the example[1]:

As an example, consider the Linnaean classification for modern humans:

- Kingdom: Animalia (with eukaryotic cells having cell membrane but lacking cell wall, multicellular, heterotrophic)
- Phylum: Chordata (all animals with a notochord)
- Class: Mammalia (vertebrates with mammary glands that in females secrete milk to nourish young, hair, warm-blooded, bears live young)
- Order: Primates (collar bone, eyes face forward, grasping hands with fingers, two types of teeth: incisors and molars)
- Family: Hominidae (upright posture, large brain, stereoscopic vision, flat face, hands and feet have different specializations)
- Genus: Homo (s-curved spine, "man")
- Species: Homo sapiens (high forehead, well-developed chin, skull bones thin)

An individual human is therefore an instance of the increasingly general subclasses of biological organism: Homo sapiens, Homo, Hominidae, Primates, Mammalia, Chordata and Animalia.

Many other systems have this sort of organization. The Standard Industrial Classification system (SIC) maintained by the US Department of Labor consists of about 10,000 classes indexed by the increasingly specific terms *division, major group, industry group* and *industry*. The company Google.com is an instance of the increasingly general subclasses *Information Retrieval Services*; *Computer Programming, Data Processing and Related Services*; *Business Services* and *Services*.

It is usual for a user to navigate among these subclasses by passing from a more specific to a more general class (*rolling up*) or from a more general to a more specific class (*drilling down*). This can be done in OWL either by following the rdfs:subClassOf links among the classes or by following a link from one class to the index object, following links among the

[1] http://en.wikipedia.org/wiki/Linnaean_taxonomy#Example_classification:_humans

index objects, then following a link back to another class, as shown in Figure 9.

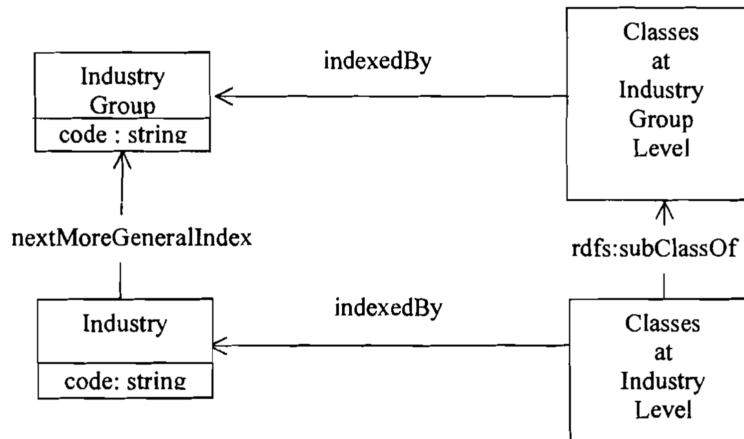


Figure 9 Indexed System of Classes

In Figure 9, the classes *IndustryGroup* and *Industry* have instances which are individuals, and are the domain of a datatype property *code* whose range is xsd:string. The other two classes have instances which are themselves classes (whose instances in turn are individual companies).

The property *nextMoreGeneralIndex* whose domain is *Industry* and range *IndustryGroup* is equivalent to the composition of the subproperty of the inverse of *indexedBy* whose range is *Industry*, *rdfs:subClassOf* and the subproperty of *indexedBy* whose range is *IndustryGroup*.

Some systems have a different organization. For example, SNOMED[2] is a system of hundreds of thousands of concepts used to classify medical records. The concepts are divided into 19 hierarchies. Concepts in each hierarchy are disjoint from concepts in all other hierarchies. An individual record can be an instance of concepts in many of the hierarchies.

This sort of system is called a *faceted system* in the library community [9], and is probably most familiar as the stereotypical oriental noodle restaurant menu: choose one type of noodle, one type of protein ingredient and one type of sauce. Each root concept in SNOMED defines a *facet* containing all its subconcepts.

OWL Full has facilities for specifying indexed collections of classes. Since Class and Individual are not disjoint, it is possible to define a property say *classIndex* whose domain is rdfs:Class and whose range is a class containing the index terms.

OWL DL can be used to represent faceted systems, since an individual can be an instance of several classes, even though representation of the class systems themselves requires OWL Full.

It is common to have a faceted system each facet of which is an indexed system (although

[2] http://www.snomed.org/

SNOMED is not organized this way). The whole field of data warehousing is based on systems of this kind. A fact table (for example sales in USD) is classified by a number of dimensions including time, product and store. Each dimension is an indexed system of subclasses. The most specific classes for time might be days, which are aggregated to weeks, months, quarters and years. The most specific classes for product are bar codes, which are aggregated into increasingly general classes of product. The most specific classes for store are the individual stores, which are generally aggregated into increasingly larger groups of store by geographic region.

Consider the collection of objects consisting of individual athletes completing events in the Olympics. This collection is classified by Olympiad (Beijing will be the $29^{th}$), by gender (men, women, mixed), by event (301 at the Athens Olympics) aggregated by Discipline (Swimming, Track) then Sport (Aquatics, Athletics, 28 in total), by country (202) which can be aggregated by region, and by finishing position, which can be aggregated by medal vs non-medal.

In these systems the indexing properties would be naturally represented as an indexed system of subproperties as described in the preceding section.

## 8. AGGREGATION

If we are going to have individuals like Olympic athletes organized into systems of subclasses, an obvious question is to ask how many individuals there are in a given class. Query languages like SQL support a number of standard aggregation functions:

- Count: the cardinality of a class
- Sum: the sum of the values of a numerically valued datatype property of instances of a class
- Average: the average of the values of a numerically valued datatype property of instances of a class (average = sum/count)
- Max: the maximum of the values of a numerically valued datatype property of instances of a class
- Min: the minimum of the values of a numerically valued datatype property of instances of a class

Not all logical systems can support aggregation. One requirement is that the extents of classes must be finite, and another is that the individuals in the extent of a class satisfy the unique name assumption. OWL satisfies the first requirement but not the second. OWL does have a construct owl:allDifferent, with which the user can declare a specified list of individuals to be different.

The reason OWL does not default the unique names assumption (like SQL does) is that OWL defaults the open world assumption (SQL defaults the closed world assumption).

However, there are ontologies in which the extents of the classes are known definitively. The Olympics ontology is of this kind. The set of athletes who competed in the 2004 Athens Olympics is known for certain, and the list kept by the International Olympic Committee is definitive. Every individual who competed is on that list, and all the individuals are uniquely identified. Also, every event is known, as is every medal won. This is the case because the International Olympic Committee (IOC) commissioned the Athens Organizing Committee for the Olympic Games (AOCOG), the AOCOG registered the athletes, organized the events, created the results in speech acts and kept the official records of the results, then turned the set of records over to the IOC.

The same is true of the classlist my University gives me of students enrolled in a course I am assigned to teach. If a student is not on that classlist, they are not enrolled, even though they may be attending classes. Conversely, a student on that classlist is enrolled in the course, even if they never attend, even if they have died. The list of zip codes in the US is kept definitively by the US Postal Service. The list of member nations of the United Nations is definitively kept by the UN Secretariat. The list of states of the United States is definitively kept by the US Government. All these are examples of collections of institutional facts [10] which are records of decisions made by specific institutions created in order to make such decisions. The closed world assumption can safely be made for all these classes as published by their respective institutions, if not to copies kept by other organizations such as Wikipedia whose records are not necessarily definitive.

It would be convenient for cases where the closed world assumption applies for owl:allDifferent to apply to a class. This would separate the definition of the class and its characteristics from population of its extent. Aggregation functions could then apply to these classes. The count function itself could be defined as a datatype property whose domain was owl:Class. The other aggregation functions from SQL could be defined in a way parallel to the definition of restriction classes.

A restriction class is

| | |
|---|---|
| restriction | (an anonymous class) |
| onProperty | (a property) |
| restriction kind | (maxCardinality, hasValue, allValuesFrom, ...) |
| range | (literal, individual, class) |

where ***restriction*** is a subclass of the domain of the property designated by *onProperty*.

An aggregation is

| | |
|---|---|
| aggregation | (an anonymous datatype property) |
| onClass | (a class, domain of the aggregation) |
| aggregation kind | (sum, average, max, min) |
| of | (a datatype property P) |

where the range of the anonymous aggregation property is the same as the range of P.

It is common to want a collection of aggregations on an indexed system of classes (eg count of gold, silver and bronze medal winners). In SQL this is achieved using a GROUP BY clause. In OWL it would be possible to use a collection of indexing properties on the *onClass* to index the aggregation datatype properties in an aggregation definition.

## 9. SUMMARY

We have proposed a number of enhancements to OWL which can be used to increase the functionality of the language.

Table 1 Summary of enhancements and consequent functionality

| Enhancement | Consequent functionality |
|---|---|
| isOneToOneTo<br>imageOf<br>mono<br>epi | Injection constraint<br>Surjection constraint<br>Identity (single property) |
| imageOf alone | Correlation of classes |
| Cartesian product<br>Property composition | Composite identity<br>n-ary property<br>property class<br>parameterised families of properties |
| Property composition alone | Derived properties<br>Principle of consistent dependency<br>Indexed families of classes |
| allDifferent to a class<br>Aggregation | Aggregations<br>Indexed families of aggregations |

## 10. DISCUSSION AND CONCLUSION

The proposals in this paper need to be considered in the context of a variety of proposals to address weaknesses in the expressivity of OWL made in recent years.

In particular, the concept of role composition is well known in description logics (Baader et al. 2003). Roles in DL are properties in OWL, but DLs including role composition are generally not decidable (e.g. [3]). Therefore only very restricted property composition has been proposed for OWL [5]. So the present proposal for property composition, and its consequences, would be features of OWL Full not included in OWL DL.

On the other hand, property composition is implicit in the logic programming extensions proposed for OWL, including SWRL [6] and the proposals of [7]. These proposals are definitely not decidable [7].

Further, the rule language proposals for OWL do not carry with them the data-oriented