Web Ontology Language: OWL2

CSE 595 – Semantic Web

Instructor: Dr. Paul Fodor

Stony Brook University

http://www3.cs.stonybrook.edu/~pfodor/courses/cse595.html

Lecture Outline

- Requirements for Ontology Languages
- Compatibility of OWL2 with RDF/RDFS
- The OWL Language
  - Ontology Documents
  - Property Types
  - Property and Class Axioms
- OWL2 Profiles

Why a Web Ontology Language?

- RDF and RDF Schema are deliberately very limited
  - RDF is (roughly) limited to binary ground predicates,
  - RDF Schema is (roughly) limited to a subclass hierarchy and a property hierarchy, with domain and range definitions of these properties.
    - They are designed with flexibility in mind

Why a Web Ontology Language?

- We need to express more advanced, more '*expressive*' knowledge
  - For example,
    - that every person has exactly one birth date, or
    - that no person can be both male and female at the same time.
  - The Web Ontology Working Group and the OWL Working Group identified a number of characteristic use cases for the Semantic Web that require much more language features than those that RDF and RDFS have
    - http://www.w3.org/2001/sw/WebOnt/
    - http://www.w3.org/2007/OWL/wiki/OWL_Working_Group

Why a Web Ontology Language?

- The resulting language, OWL2, for the Web Ontology Language, is closely related to a fragment of a family of logics that are specially crafted for representing terminological knowledge: **Description Logics (DL)**
- OWL2 is the second iteration of the OWL language.

Requirements for Ontology Languages

- An explicit formal specification of the concepts in a domain is called an ***ontology***
- Languages that allow us to express ontologies are therefore called ***ontology languages***
- The main requirements for these languages are:
  - a well-defined syntax,
  - a formal semantics,
  - sufficient expressive power,
  - convenience of expression, and
  - efficient reasoning support

Syntax

- A syntax is ***well-defined*** if you can use it to write down everything a language allows you to express in an unambiguous manner
  - A well-defined syntax is not necessarily very user-friendly
    - For instance, the RDF/XML syntax is notoriously hard for people to read.
      - However, this drawback is not very significant because most ontology engineers will use specialized ontology development tools, rather than a text editor, for building ontologies
- OWL2 builds on RDF and RDFS and uses an extension of their syntax

Formal Semantics

- A ***formal semantics*** describes the meaning of a language <u>precisely</u>
  - *Precisely* means that the semantics does not refer to subjective intuitions, nor is it open to different interpretations by different people (or machines).
  - The importance of a formal semantics is well-established in the domain of mathematical logic
    - The combination of formal semantics with a well-defined syntax allows us to ***interpret*** sentences expressed using the syntax: we ***know*** what is meant by the sentence
    - Formal semantics also allows us to ***reason*** about the knowledge expressed in the sentences

Formal Semantics

- For instance, the formal semantics of RDFS allows us to reason about class membership
  - Given

```
    x rdf:type C.
    C rdfs:subClassOf D.
```

  we can infer that **`x`** is an instance of **`D`**.

- The **`rdfs:domain`** and **`rdfs:range`** properties allow similar inferences:

```
    p rdfs:range D.
    x p y.
```

  allows us to infer that **`y rdf:type D`**.

Expressivity

- The expressive power of RDF and RDFS is very limited in some areas.
  - If we build ontologies, we may want to be able to reason about:
    - ***Class Membership***: a more precise description of the conditions under which an instance can be considered to belong to a class would allow for more fine-grained reasoning
      - For instance, if we have declared that certain property-value pairs are a **sufficient** condition for membership in a class **`A`**, then if an instance **`x`** satisfies these conditions, we can conclude that **`x`** must be an instance of **`A`**

Expressivity

- ***Classification***: we would like to use the conditions on class membership to infer relations between the classes themselves
- ***Equivalence between classes***: for example, the class **`Tortoise`** shares all its members with the class **`Land_Turtle`**; they are therefore equivalent
- ***Equality between instances***: we would like to be able to state when two instances are the same: the **`morning_star`** and the **`evening_star`** are names for the same planet **`venus`**; these instances are therefore the same.

Expressivity

- ***Disjointness and Difference***: sometimes we know that two classes do not share any instances (they are ***disjoint***) or that two instances are decidedly not the same thing.
  - For example, **`Winner`** and **`Loser`** are disjoint in a game, and **`roger_federer`** and **`rafael_nadal`** are different individuals.
- ***Boolean Combinations of Classes***: sometimes classes need to be combined in ways that go beyond subclass relations.
  - For instance, we may want to define the class **`Person`** to be the **disjoint union** of the classes **`Female`** and **`Male`**.

Expressivity

- *Local Scope of Properties*: **`rdfs:range`** states that the instances in the range of a property, say **`plays`**, all belong to a certain class.
  - In RDFS we cannot declare range restrictions that differentiate between contexts.
  - For example, we cannot say that tennis players play only tennis, while other people may play badminton
- *Special Characteristics of Properties*: Sometimes it is useful to say that a property is:
  - *transitive*, such as **`greater_than`**
  - *unique*, like **`is_mother_of`**
  - the *inverse* of another property, such as **`eats`** and **`is_eaten_by`**

Expressivity

- ***Cardinality Restrictions***: Sometimes we need to place restrictions on how many distinct values a property may or must take
  - For example, each person has exactly **two** parents,
  - a course has **at least one** TA
- ***Consistency***: Once we can determine relations between classes, we may also want to determine conflicts between their definitions.
  - Suppose we have declared **`Fish`** and **`Mammal`** to be disjoint classes.
    - It is then an error to assert that **`dolphin`** is an instance of both.

Reasoning Support

- Formal semantics is a prerequisite for *reasoning support*
  - Derivations such as the preceding ones can be made mechanically instead of by hand
- Automatic reasoning is important because it allows us to check the correctness of the ontology
  - check the consistency of the ontology
  - check for unintended relations between classes
  - check for unintended classifications of instances
- Checks like these are extremely valuable for designing large ontologies, for cases where multiple authors are involved, and for integrating and sharing ontologies from various sources.

Reasoning Support

- We can provide formal semantics and reasoning support to an ontology language by mapping it to a known logical formalism, and by using automated reasoners that already exist for those formalisms
  - In designing such a formal language one should be aware of the trade-off between **expressive power** and **efficient reasoning** support.
  - Generally speaking, the richer the logical formalism, the less efficient the reasoning support becomes, often crossing the border of decidability (that is, reasoning on such logics is not guaranteed to terminate).
  - Need for a compromise: a language that can be supported by reasonably efficient reasoners, while being sufficiently expressive to represent a large variety of knowledge

OWL2 with RDF/RDFS

- Ideally, OWL2 is an extension of RDF Schema, in the sense that OWL2 adopts the RDFS meaning of classes and properties (**`rdfs:Class`**, **`rdfs:subClassOf`**, etc.) and adds language primitives to support the richer expressiveness required
  - This approach would be consistent with the layered architecture of the Semantic Web
  - Unfortunately, simply extending RDF Schema would work against obtaining expressive power and efficient reasoning
    - RDF Schema has some very powerful modeling primitives. Constructions such as **`rdfs:Class`** (the class of all classes) and **`rdfs:Property`** (the class of all properties) are **very expressive** and would lead to uncontrollable computational properties if the logic underlying OWL2 included these primitives in their generality

Two Semantics

- The full set of requirements for an ontology language seems unobtainable: efficient reasoning support does not exist for a language as expressive as a combination of RDF Schema with a full logic
- These requirements have prompted the successive W3C working groups to split OWL2 into two different sublanguages, each with a different underlying semantics geared toward fulfilling different aspects of the full set of requirements:
  - **OWL2 Full**
  - **OWL2 DL**

OWL2 Full: RDF-Based Semantics

- OWL2 Full uses all the OWL2 language primitives
  - It allows the combination of these primitives in arbitrary ways with RDF and RDF Schema
  - It includes the ability to change the meaning of the predefined (RDF or OWL2) primitives by applying the language primitives to each other
    - In OWL2 Full, we could impose a cardinality constraint on the class of all classes, essentially limiting the number of classes that can be described in any ontology.
    - OWL2 Full is mapped to an RDF-based semantics, so it is structurally and semantically fully upward-compatible with RDF: any legal RDF document is also a legal OWL2 Full document, and any valid RDF Schema inference is also a valid OWL2 Full conclusion
  - The disadvantage is that the language has become so powerful as to be undecidable

OWL2 DL: Direct Semantics

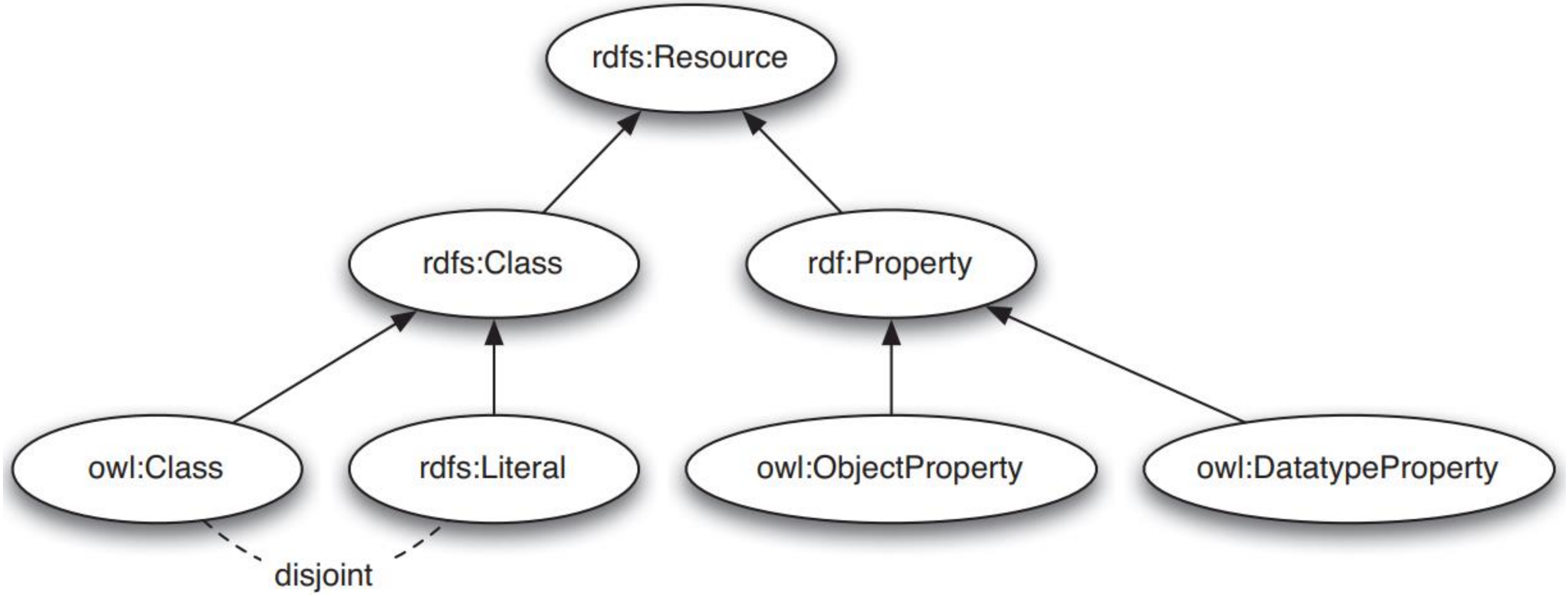
- OWL2 DL is mapped onto a description logic (DL)
  - Description logics are a subset of predicate logic for which efficient reasoning support is possible
  - OWL2 DL restricts the way in which the primitives of OWL2, RDF, and RDFS may be used:
    - OWL2 DL does not allow the application of OWL2's primitives to each other
    - OWL2 DL can only define classes of non-literal resources: all OWL2 DL classes are instances of **`owl:Class`** rather than **`rdfs:Class`**
    - OWL2 DL strictly separates properties for which the range includes non-literal resources from those that relate to literal values: all OWL2 DL properties are instances of either **`owl:ObjectProperty`** or **`owl:DatatypeProperty`** but not both!
    - In OWL2 DL a resource cannot be a class, property, or instance at the same time - they may share the same name (this is called "*punning*") but will always be treated as distinct things by the underlying logic.

OWL2 DL: Direct Semantics

- OWL2 DL can make use of a wide range of existing reasoners such as Pellet, FaCT, RACER, and HermiT
- The disadvantage is that we lose full compatibility with RDF
  - An RDF document will in general have to be extended in some ways and restricted in others before it is a legal OWL2 DL document
- However, every legal OWL2 DL document is a legal RDF document

OWL2 DL: Direct Semantics

- The subclass relationships between some modeling primitives of OWL2 and RDF/RDFS:



The OWL Language

- Vocabulary:
  - The members of classes are commonly called ***individuals*** rather than *instances*
  - When we state that some resource is of a certain type, we call this an ***assertion***

    ```
    roger_federer rdf:type Person.
    ```

    is a ***class assertion*** relating the individual **`roger_federer`** to its class.
  - When we combine classes, properties, and instances, they form ***expressions***

    ```
    _:x rdf:type owl:Class ;
      owl:unionOf ( :Man :Woman ).
    ```

    is a ***class expression*** that specifies the (anonymous) union of the classes **`Man`** and **`Woman`**.

The OWL Language

- If we then relate definitions to one of our classes, we create an ***axiom***

```
Person owl:equivalentClass _:x.
_:x rdf:type owl:Class ;
    owl:unionOf ( :Man :Woman ) .
```

is an equivalent class ***axiom*** that states that the class `Person` is equivalent to the union we introduced above.

  - Class axioms are sometimes called ***restrictions***, as they constrain the set of individuals that can be a member of a class.

The OWL Language

- OWL2 is essentially a language for describing sets of things
  - These sets are called '***classes***.'
  - Any statement we make about a class in OWL2 is meant to differentiate that class from the ***set of all things***

Syntax

- OWL2 builds on RDF and RDF Schema and thus can be expressed using all valid RDF syntaxes
- However, many syntaxes exist for OWL2, each of which has its own benefits and drawbacks:
  - ***Functional-Style Syntax*** closely relates to the formal structure of ontologies
    - It is used in the language specification document, in the definitions of the semantics of OWL2 ontologies, the mappings from and into RDF syntaxes, and the different profiles of OWL2.
    - It is much more compact and readable than many of the other syntaxes.
    - For instance, a class restriction can be written in this syntax as:

```
EquivalentClasses( Person
          ObjectUnionOf( Man Woman ) )
```

Syntax

- ***OWL/XML Syntax*** is an XML syntax for OWL2 that does not follow the RDF conventions, but closely maps onto the functional-style syntax.
  - http://www.w3.org/TR/owl-xml-serialization/
  - The main benefit of this syntax is that it allows us to interact with ontologies using standard off-the-shelf XML authoring tools.
  - The OWL/XML syntax of an equivalent class axiom is:

```
<EquivalentClasses>
  <Class abbreviatedIRI=":Person"/>
  <ObjectUnionOf>
    <Class IRI="#Man"/>
    <Class IRI="#Woman"/>
  </ObjectUnionOf>
</EquivalentClasses>
```

Syntax

- ***Manchester Syntax*****,** originally developed at University of Manchester, this syntax is designed to be as human-readable as possible.
  - It is the syntax used in the user interface of most current ontology editors such as Protégé

```
Class: Person
    EquivalentTo: Man or Woman
```

- In addition to these syntaxes, all RDF syntaxes can be used for OWL (the Turtle syntax is most commonly used)

Ontology Documents

- An OWL2 ontology starts with a collection of assertions for housekeeping purposes.
  - These assertions introduce a base namespace, the ontology itself, its name, possible comments, version control, and inclusion of other ontologies. For example:

```
@prefix : <http://www.semanticwebprimer.org/ontologies/apartments.ttl#> .
@prefix dbpedia-owl: <http://dbpedia.org/ontology/> .
@prefix dbpedia: <http://dbpedia.org/resource/> .
@base <http://www.semanticwebprimer.org/ontologies/apartments.ttl> .

<http://www.semanticwebprimer.org/ontologies/apartments.ttl>
  rdf:type owl:Ontology ;
  rdfs:label "Apartments Ontology"^^xsd:string ;
  rdfs:comment "An example OWL2 ontology"^^xsd:string ;
  owl:versionIRI <http://www.semanticwebprimer.org/ontologies/apartments.ttl#1.0> ;
  owl:imports <http://dbpedia.org/ontology/> ;
  owl:imports <http://dbpedia.org/resource/> .
```

Ontology Documents

- *Imports*: **`owl:imports`** points to other ontologies whose axioms are to be part of the current ontology
  - Our apartments ontology imports <u>all axioms defined in the DBPedia ontology</u>, as well as <u>everything in DBPedia itself</u>.
  - This immediately highlights one of the problems with the **`owl:imports`** : in order to be able to use some of the information in DBPedia, we have to import all 672 million triples described in it

Ontology Documents

- While namespaces are used only for disambiguation, imported ontologies provide definitions that can be used.
- Typically an ontology contains an import statement for every namespace it uses, but it is possible to import additional ontologies
- The **`owl:imports`** property is **transitive**; that is, if ontology O_i imports ontology O_j , and ontology O_j imports ontology O_k, then ontology O_i also imports ontology O_k

Property Types

- OWL2 distinguishes two types of properties: ***object*** properties and ***datatype*** properties
- Object Properties relate individuals to other individuals
  - Example: **`rents`**:

```
rents rdf:type owl:ObjectProperty ;
   rdfs:domain Person ;
   rdfs:range Apartment ;
   rdfs:subPropertyOf livesIn .
```

- Datatype Properties relate individuals to literal values of a certain data type
  - Example: **`age`**:

```
age rdf:type owl:DatatypeProperty ;
   rdfs:range xsd:nonNegativeInteger .
```

Property Types

- Just as in RDF, OWL2 allows one to use XML Schema datatypes for indicating the type of a literal or specifying the range of a datatype property.
  - OWL2 introduces two additional datatypes **`owl:real`** and **`owl:rational`**

Universal Datatype	rdfs:Literal			
Numbers	owl:rational		owl:real	
	xsd:double	xsd:float	xsd:decimal	xsd:integer
	xsd:long	xsd:int	xsd:short	xsd:byte
	xsd:nonNegativeInteger		xsd:nonPositiveInteger	
	xsd:positiveInteger		xsd:negativeInteger	
	xsd:unsignedLong		xsd:unsignedInt	
	xsd:unsignedShort		xsd:unsignedByte	
Strings	rdf:PlainLiteral (RDF plain literals)			
	xsd:string	xsd:NCName	xsd:Name	xsd:NMTOKEN
	xsd:token	xsd:language	xsd:normalizedString	
Boolean Values	xsd:boolean (value space: *true* and *false*)			
Binary Data	xsd:base64Binary		xsd:hexBinary	
IRIs	xsd:anyURI			
Time Instants	xsd:dateTime (optional time zone offset)			
	xsd:dateTimeStamp (required time zone offset)			
XML Literals	rdf:XMLLiteral			

Annotation Properties

- ***Annotation properties*** are properties that do not carry any meaning under the direct semantics of OWL2 DL.
  - That is, they are ignored by a DL reasoner.
  - However, they will be taken into account by RDF Schema and OWL2 Full reasoners.
  - Annotation properties are typically used for adding readable labels, comments, or explanations to OWL2 ontologies, classes, properties, and individuals

Annotation Properties

- Example: **`label`** property to be of type **`owl:AnnotationProperty`** with a range of **`rdf:PlainLiteral`** (a special RDF datatype for natural language text – that is, plain literals can have a language tag)

```
label rdf:type owl:AnnotationProperty ;
  rdfs:range rdf:PlainLiteral ;
  rdfs:subPropertyOf rdf:label .
Apartment :label "Apartment"@en,
  "Appartement"@nl,
  "Apartament"@ro .
```

  - We defined the **`label`** property to be a subproperty of **`rdf:label`**, and then give three labels to the **`Apartment`** class in English, Dutch, and Romanian

Top and Bottom Properties

- All object properties in OWL2 are a subproperty of **`owl:topObjectProperty`**
  - This property is defined as the property that relates ***all*** individuals in the ontology.
- Conversely, **`owl:bottomObjectProperty`** is the property that relates ***no*** individuals.
- Similarly, **`owl:topDataProperty`** relates all individuals to any possible literal value, and **`owl:bottomDataProperty`** relates no individual to any literal value

Transitive Properties

- We know that **`rdfs:subClassOf`** is transitive: every class is a subclass of all superclasses of its direct superclass.
- There are other relations which are transitive as well, such as **`isPartOf`**
- We can define a property as transitive as follows:

```
isPartOf rdf:type owl:ObjectProperty ;
   rdf:type owl:TransitiveProperty .
```

Transitive Properties

- Transitive properties are so-called ***composite*** properties: they can be said to be composed of multiple steps
  - For instance, given:

```
BaronWayApartment isPartOf BaronWayBuilding .
BaronWayKitchen isPartOf BaronWayApartment .
```

a reasoner will infer:

```
BaronWayKitchen isPartOf BaronWayBuilding .
```

    - This last **`isPartOf`** relation is composed of the two preceding property assertions.

Transitive Properties

- Because of this composition, transitive properties are subject to a number of restrictions
  - Composite properties may not occur in the following axioms:
    - Qualified and non-qualified cardinality restrictions on classes;
    - Self restrictions on classes,
    - Disjoint property axioms.
  - They may furthermore not be assigned the following property types:
    - Functional or inverse functional;
    - Irreflexive;
    - Asymmetric.

Symmetric and Asymmetric Properties

- Some properties, such as **`isAdjacentTo`**, are *symmetric*: if **`a isAdjacentTo b`**, the **inverse** holds as well: **`b isAdjacentTo a`**
  - In other words, symmetric properties are equivalent to their inverse

```
isAdjacentTo rdf:type owl:ObjectProperty ;
   rdf:type owl:SymmetricProperty .
```

Symmetric and Asymmetric Properties

- For other properties, this will never be the case - for instance, the **`isCheaperThan`** relation is ***asymmetric***, since nothing can be more expensive than something that they are cheaper than:

```
isCheaperThan rdf:type owl:ObjectProperty;
   rdf:type owl:AsymmetricProperty ;
   rdf:type owl:TransitiveProperty .
```

Functional and Inverse-Functional Properties

- For some properties we know that every individual can always have at most one other individual related via that property
  - For instance, **`hasNumberOfRooms`** is a functional property: one apartment can only have one number of rooms:

```
hasNumberOfRooms rdf:type owl:DatatypeProperty ;
   rdf:type owl:FunctionalProperty .
```

Functional and Inverse-Functional Properties

- If two apartments **`a1`** and **`a2`** are related via **`hasRoom`** to the same room **`r`**, this is not necessarily inconsistent: the individuals will simply be inferred to be the *same*
  - The **`hasRoom`** property is called to be ***inverse-functional***:

```
hasRoom rdf:type owl:ObjectProperty ;
   rdf:type owl:InverseFunctionalProperty.
```

Reflexive and Irreflexive Properties

- *Reflexivity* of a property means that every individual is related via that property to itself
  - For instance, everything **`isPartOf`** itself

```
isPartOf rdf:type owl:ObjectProperty ;
  rdf:type owl:ReflexiveProperty .
```

- *Irreflexivity* means that no individual is related to itself via that property.
  - Most properties with disjoint domain and range are actually irreflexive
    - An example is the **`rents`** property:

```
rents rdf:type owl:ObjectProperty ;
  rdf:type owl:IrreflexiveProperty
```

Property Axioms

- In addition to the property types discussed, we can specify additional characteristics of properties in terms of how they relate to classes and other properties

Property Axioms

- ***Domain and Range***: the way in which OWL2 treats domain and range for properties is exactly the same as in RDF Schema:
  - If more than one **`rdfs:range`** or **`rdfs:domain`** is asserted for a property, the actual range or domain is the ***intersection*** of the classes specified in the property axiom
  - A common misunderstanding is that domain and range work as a constraint on the types of individuals that may be related via a property
  - In fact, domains and ranges can only be used to determine class membership for these individuals
    - Given the definition of **`rents`**, any two individuals **`p`** and **`a`** such that **`p rents a`** will be classified as members of **`Person`** and **`Apartment`** respectively.

Property Axioms

- *Inverse Properties*: the pair **`rents`** and **`isRentedBy`** are *inverse*
  - For instance:

```
isRentedBy rdf:type owl:ObjectProperty;
   owl:inverseOf :rents
```

  means that a reasoner will determine that any two individuals **`p`** and **`m`** that have the relation **`m isRentedBy p`** in addition to a stated relation **`p rents m`**
  - Domain and range are inherited from the inverse property: **`isRentedBy`** has **`Apartment`** as **`domain`** and **`Person`** as **`range`**.
  - In OWL2 DL, only object properties can have an inverse

Property Axioms

- ***Equivalent Properties***: Properties can also be defined as ***equivalent*** - that is, every two individuals related via a property will always be related via its equivalent, and vice versa.
  - Equivalence is a convenient mechanism for mapping elements of different ontologies to each other.
    - For instance:

```
isPartOf rdf:type owl:ObjectProperty ;
  owl:equivalentProperty dbpedia:partOf.
```

Property Axioms

- ***Disjoint Properties***: For some properties we know that no two individuals related via one property can be related via the other: the sets of pairs of individuals for which the properties can hold are disjoint.
  - For examples, the **`rents`** and **`owns`** properties: clearly, you cannot rent something you own

```
rents rdf:type owl:ObjectProperty ;
  rdfs:domain Person ;
  rdfs:range Apartment ;
  owl:disjointProperty owns .
```

  - Under the direct semantics of OWL2 DL, the **`owl:ObjectProperty`** and **`owl:DatatypeProperty`** are disjoint

Property Axioms

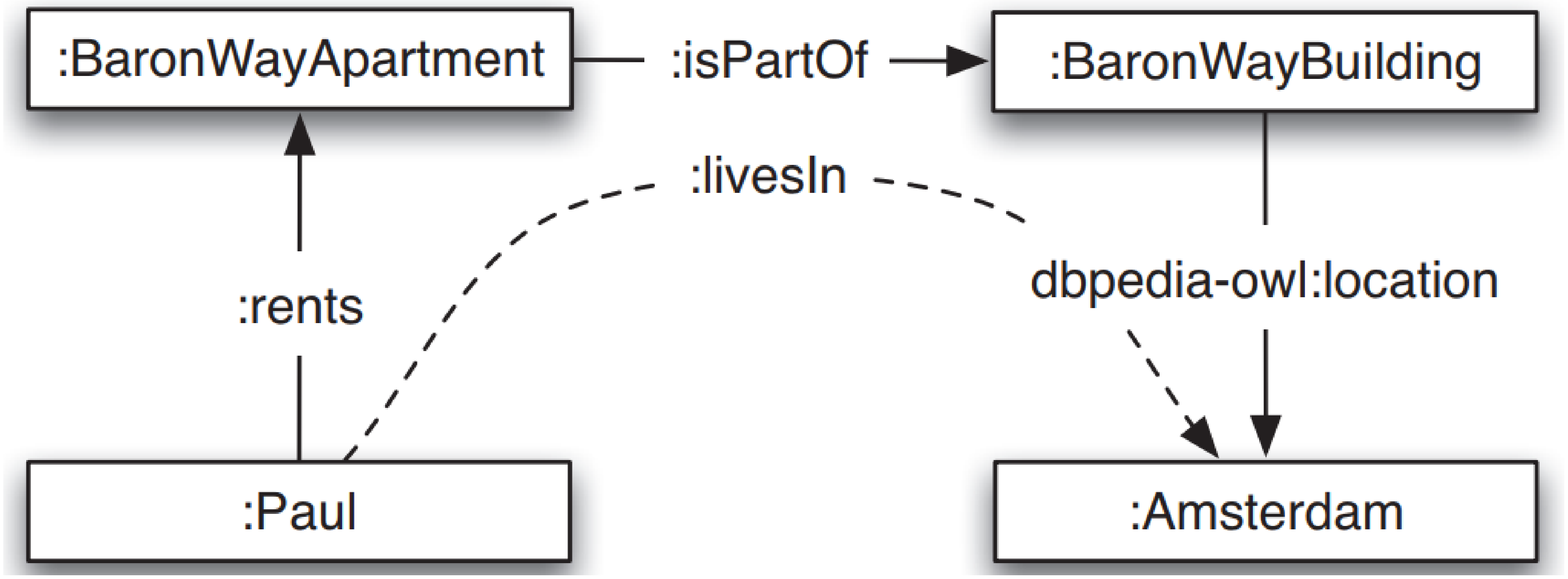
- ***Property Chains***: Sometimes it is useful to specify **shortcuts** along the graph of properties relating various individuals.
  - For instance, if we know that **`Paul`** rents the **`BaronWayApartment`**, and that the **`BaronWayApartment`** **`isPartOf`** the **`BaronWayBuilding`**, for which the **`dbpedia:location`** is **`dbpedia:Amsterdam`**, we infer that **`Paul`** must have a **`livesIn`** relation with **`Amsterdam`**.
  - In OWL2 we can specify the **`livesIn`** property using a property chain axiom:

```
livesIn rdf:type owl:ObjectProperty ;
   owl:propertyChainAxiom
      ( rents isPartOf location ) .
```

The property chain axiom does not make the **`livesIn`** property equivalent to the chain of properties (if we have declared instances of **`livesIn`** we don't infer anonymous resources for the components of the chain); it is rather only inferred from the chain.

Property Axioms

- ***Property Chains***:
  - dotted lines are inferred by the reasoner



Property Axioms

- ***Property Chains***:
  - In OWL2 DL, property chains may only involve object properties, though most reasoners can handle chains that have a datatype property as last step.
  - Property chains are subject to a number of restrictions
    - Just like transitive properties, the superproperty of property chains is composite - it means that they cannot be used in a number of axioms
    - The property chain may not be **recursive**: the superproperty of the chain, its inverse, or one of its subproperties (or their inverse) may not occur in the property chain axiom.

Property Axioms

- ***Property Chains***:
  - OWL2 DL does not allow us to extend the **`livesIn`** property in the following way:

```
livesIn rdf:type owl:ObjectProperty ;
   owl:propertyChainAxiom
     ( rents isPartOf dbpedia-owl:location ) ;
   owl:propertyChainAxiom
     ( livesIn dbpedia-owl:country ) .
```

even though it would allow us to infer that because **`Paul`** lives in **`dbpedia:Amsterdam`**, he must live in **`dbpedia:Netherlands`** as well.

Class Axioms

- Classes are defined by asserting a resource to be of type **`owl:Class`**
  - There are two predefined classes that play an important role in reasoning:
    - **`owl:Thing`** is the most ***general*** class: every possible OWL2 individual is a member of this class, and every instance of **`owl:Class`** is a subclass of **`owl:Thing`**
      - Restrictions on **`owl:Thing`** have very far-reaching consequences: they hold for every class and individual in the ontology
    - **`owl:Nothing`** is the *empty* class: it has no members, and every instance of **`owl:Class`** is a superclass of that class.
      - Inconsistent classes cannot have any members, and are therefore equivalent to **`owl:Nothing`**

Class Axioms

- ***Subclass Relations***:
    - Subclass relations are defined as in RDF Schema
        - For example, we can define a class **`LuxuryApartment`** as follows:

```
LuxuryApartment rdf:type owl:Class ;
   rdfs:subClassOf :Apartment .
```

Class Axioms

- *Class Equivalence*:
  - Equivalence of classes means that every member of a class must also be a member of the equivalent class, and vice versa.
  - In other words, both classes cover exactly the same set of individuals.
  - Class equivalence can be defined using an **`owl:equivalentClass`** property:

```
Apartment owl:equivalentClass dbpedia:Apartment .
```

states that the **`Apartment`** class in our apartment ontology is equivalent to the **`dbpedia:Apartment`** imported from DBPedia.

Class Axioms

- ***Class Equivalence***:
  - Asserting an equivalence relation between classes is equivalent to asserting subclass relations in both directions:

```
Apartment rdfs:subClassOf dbpedia:Apartment .
```

and

```
dbpedia:Apartment rdfs:subClassOf Apartment .
```

Class Axioms

- ***Punning***:
  - The DBPedia apartment definition comes from the **`dbpedia`** namespace instead of from **`dbpedia-owl`**
    - It is not a class, but an individual
  - Compared to our ontology, DBPedia describes apartments at a higher level of abstraction: the classes in the DBPedia ontology are not intended to classify individual entities (such as apartments in Amsterdam), but rather individual ***topics***.
  - Treating individuals as classes is called ***meta-modeling***.
    - Although the direct semantics of OWL2 do not allow for meta-modeling, OWL2 DL circumvents this limitation by a syntactic trick called ***punning***, or 'word play.'

Class Axioms

- ***Punning***:
  - Means that whenever an URI, such as **`dbpedia:Apartment`** appears in a class axiom, it is treated as a class, and when it appears in an individual assertion, it is treated as individual.
  - Punning is allowed in the following situations: class names, individual names, and property names may be freely interchanged.
  - However, object property names and datatype property names may not mix.

Class Axioms

- ***Enumerations***:
  - The most straightforward (though inexpressive and computationally expensive) way to define a class is by explicitly enumerating all individuals it consists of:

```
BaronWayRooms rdf:type owl:Class;
  owl:oneOf (
    BaronWayKitchen
    BaronWayBedroom1
    BaronWayBedroom2
    BaronWayBedroom3
    BaronWayLivingroom
    BaronWayBathroom
    …) .
```

  - This defines the class of all rooms: this type of class definition can be very cumbersome if the list of known members is very long, or even impossible if we do not currently know all individuals

Class Axioms

- ***Disjoint Classes***:
  - Disjointness of classes means that no member of one class can also be a member of the other class.
    - The sets of individuals described by the classes do not overlap.
    - For example, the **`LuxuryApartment`** class is disjoint from **`ColdWaterFlat`** using the **`owl:disjointWith`** property:

```
LuxuryApartment owl:disjointWith ColdWaterFlat.
```

means that no **`LuxuryApartment`** can be a **`ColdWaterFlat`** at the same time.

Class Axioms

- ***Complement***:
  - The ***complement*** **`C`** of a class **`A`** is the class of all things not belonging to **`A`**.
    - In other words, the union of **`A`** and **`C`** is equivalent to **`owl:Thing`**
      - this means that the complement is always a superclass of the disjoint classes of **`A`**.
    - For instance:

```
FurnishedApartment rdfs:subClassOf Apartment .
UnFurnishedApartment rdfs:subClassOf Apartment;
   owl:complementOf FurnishedApartment .
```

states that the class of furnished apartments is the complement of the class of apartments without furnishing

Class Axioms

- ***Complement***:
  - Complementarity is a very powerful modeling construct that should be used with care
    - **`Apartment`** covers both **`FurnishedApartment`** and its complement **`UnfurnishedApartment`**, so **`Apartment`** will be equivalent to **`owl:Thing`**: there cannot be an individual not belonging to a class nor its complement.
  - If we then additionally introduce a class that is disjoint with **`Apartment`**, this class is effectively disjoint with **`owl:Thing`**.
    - If we state:

```
SemiDetached owl:disjointWith Apartment .
```

    - the **`SemiDetached`** class will be empty because the class **`Apartment`** covered both **`FurnishedApartment`** and its complement, so **`Apartment`** was equivalent to **`owl:Thing`**

Class Axioms

- ***Union and Disjoint Union***:
  - We often know for some class that it is equivalent to two or more other classes: every member of the class is a member of at least one of the classes in the union
    - This can be specified using the **`owl:unionOf`** construct

```
Apartment rdf:type owl:Class ;
   owl:unionOf ( ColdWaterFlat
      LuxuryApartment
      PenthouseApartment
      StudioApartment
      BasementApartment
      FurnishedApartment
      UnFurnishedApartment
   ) .
```

Class Axioms

- ***Union and Disjoint Union***:
  - In many cases, the member classes of the union are mutually disjoint.
  - Of course, we can explicitly assert **`owl:disjointWith`** relations between each class, but it is more convenient to state this directly:

```
Apartment rdf:type owl:Class;
   owl:disjointUnionOf (
      FurnishedApartment
      UnFurnishedApartment
   ) .
```

Class Axioms

- ***Intersection***:
  - We can state that a class is exactly the intersection of two or more other classes: every member of the class is a member of each of the classes in the intersection.
    - For example:

```
LuxuryApartment rdf:type owl:Class ;
  owl:intersectionOf ( GoodLocationApartment
    LargeApartment NiceViewApartment
    LuxuryBathroomApartment ) .
```

states that the **LuxuryApartment** class is populated by those individual apartments that have a good location, are of large size, have a nice view, and have a luxury bathroom.

Class Axioms on Properties

- ***Class restriction axioms*** are axioms that **restrict** the set of individuals that may be considered to be members of a class by looking at their properties (they allow us to automatically infer class membership by attaching to an **`owl:Class`** a special type of anonymous class **`owl:Restriction`**)
- ***Universal Restrictions***: A universal restriction on a class **`C`** and property **`p`** stating that for every member of **`C`** all values of **`p`** belong to a certain class
  - The universal restriction can be used to specify a range for a property that is local to the restricted class

Class Axioms on Properties

- ***Universal Restrictions*** are built using **`owl:allValuesFrom`**:

```
:LuxuryBathroomApartment
  rdf:type owl:Class;
  rdfs:subClassOf [ rdf:type owl:Restriction;
    owl:onProperty :hasBathroom ;
    owl:allValuesFrom :LuxuryBathroom  ] .
```

defines the :**`LuxuryBathroomApartment`** class as a subclass of the set of individuals that **only** have instances of :**`LuxuryBathroom`** as value for the :**`hasBathroom`** property.

- An **`owl:allValuesFrom`** restriction merely states that if a member of the restricted class has a value for the property, then that value must be a member of the specified class.
  - The restriction does not require the property to have any value at all: in that case, the restriction is trivially satisfied (vacuously true).
    - In our apartment example, the definition does not require that a luxury bathroom apartment have a bathroom at all

Class Axioms on Properties

- ***Universal Restrictions*** can also be used with datatype properties
  - For instance, to state that the value of a property must be of a certain type or fall within a certain data range

Class Axioms on Properties

- ***Existential Restrictions***: An existential restriction on a class **`C`** and property **`p`** states that for every member of **`C`** there exists at least some value for **`p`** that belongs to a certain class.
  - These restrictions are specified using **`owl:someValuesFrom`**

```
:LuxuryBathroomApartment
   rdf:type owl:Class;
   rdfs:subClassOf [ rdf:type owl:Restriction;
      owl:onProperty :hasBathroom ;
      owl:someValuesFrom :LuxuryBathroom  ] .
```

defines the :**`LuxuryBathroomApartment`** class as a subclass of the set of individuals that have at least one instance of :**`LuxuryBathroom`** as value for the :**`hasBathroom`** property.

Class Axioms on Properties

- ***Necessary and Sufficient Conditions***: Instead of using the **`rdfs:subClassOf`** property to relate our classes to the restrictions, we could also have used an **`owl:equivalentClass`** property to state that the restricted class is **exactly** the class described by the restriction
  - In both cases, if we explicitly assert an individual to be an instance of the **`:LuxuryBathroomApartment`** class, the reasoner will infer that there is at least some (unknown) individual of type **`:LuxuryBathroom`** as value for the **`:hasBathroom`** property.
  - However, the **`rdfs:subClassOf`** restriction states ***necessary*** conditions for class membership, while the **`owl:equivalentClass`** restriction states ***necessary and sufficient*** conditions

Class Axioms on Properties

- ***Necessary and Sufficient Conditions***:
  - For instance, the **existential** restriction will not make a reasoner conclude that every individual that has a `:hasBathroom` relation with an individual of type `:LuxuryBathroom` must be an instance of `:LuxuryBathroomApartment`.
  - The apartment is only a ***subclass*** of the restriction, and we do not have enough information to determine whether the individual is also a member of the class itself.
  - If we make the class ***equivalent*** to the class specified by the restriction, it is clear that **any individual that satisfies the restriction must also be a member of the class**.

Class Axioms on Properties

- ***Value Restrictions*** are used when we want to define a class based on **relations with known individuals**, or **specific values for datatype properties**.
  - For example, we can define **the class of all apartments in Amsterdam**:

```
:AmsterdamApartment
  rdf:type owl:Class;
  owl:equivalentClass [
    rdf:type owl:Restriction;
    owl:onProperty dbpedia-owl:location ;
    owl:hasValue dbpedia:Amsterdam ] .
```

Class Axioms on Properties

- ***Cardinality Restrictions*** constrain the **number** of values a certain property may have for a class

```
:StudioApartment
  rdf:type owl:Class;
  rdfs:subClassOf [ rdf:type owl:Restriction;
    owl:onProperty :hasRoom ;
    owl:cardinality "1"^^xsd:integer ] .
```

specifies that a studio apartment can have **exactly** one value for the **`:hasRoom`** property

Class Axioms on Properties

- ***Cardinality Restrictions***
  - If we additionally specify the class these values need to belong to, the restriction is said to be ***qualified***

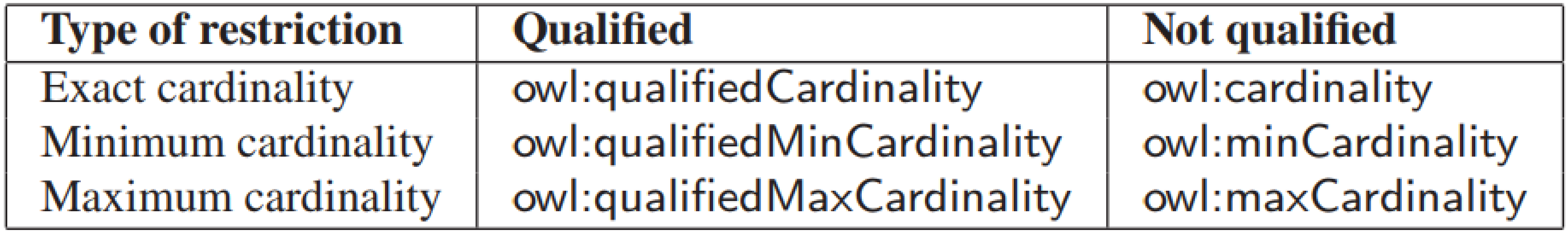
```
:StudioApartment
  rdf:type owl:Class;
  rdfs:subClassOf [ rdf:type owl:Restriction;
    owl:onProperty :hasRoom ;
    owl:qualifiedCardinality "1"^^xsd:integer
    owl:onClass [ owl:unionOf (:LivingRoom
      :Kitchen :Bedroom) ] ] .
```

specifies that a studio apartment can have **exactly** one value for the `:hasRoom` property with the values `:LivingRoom`, `:Kitchen` and `:Bedroom`

  - The qualified restriction still allows for the members of the restricted class to have additional values for the property, provided that these belong to the complement of the qualifier class
  - A qualified cardinality restriction on owl:Thing is equivalent to a restriction without qualifier

Class Axioms on Properties

- ***Cardinality Restrictions***
  - Cardinality restrictions in OWL2:



Type of restriction	**Qualified**	**Not qualified**
Exact cardinality	owl:qualifiedCardinality	owl:cardinality
Minimum cardinality	owl:qualifiedMinCardinality	owl:minCardinality
Maximum cardinality	owl:qualifiedMaxCardinality	owl:maxCardinality

Class Axioms on Properties

- ***Data Range Restrictions and Datatypes***:
  - Universal and existential restrictions on datatype properties allow members of a class to have any value from the specified datatype as value for the property.
  - Sometimes we need more precise definitions on the **range** of values allowed for a property:
    - For instance, the class of people who can rent apartments must be **adults**, or the **minimum** size of apartments

Class Axioms on Properties

- *Data Range Restrictions and Datatypes*:

```
:Adult rdfs:subClassOf dbpedia:Person ;
  rdfs:subClassOf [ rdf:type owl:Restriction ;
  owl:onProperty :hasAge ;
  owl:someValuesFrom [ rdf:type rdfs:Datatype ;
    owl:onDatatype xsd:integer ;
    owl:withRestrictions (
      [ xsd:minInclusive "18"^^xsd:integer ])]].
```

`:Adult` is the subclass of persons that have a value for the **`:hasAge`** that falls within the range of integers equal to or larger than 18.

  - The data range is defined as an anonymous class of type **`rdfs:Datatype`**.

Class Axioms on Properties

- ***Data Range Restrictions and Datatypes***:
  - A new named datatype that we could reuse throughout the ontology:

```
:AdultAge rdf:type rdfs:Datatype ;
  owl:onDatatype xsd:integer ;
  owl:withRestrictions (
    [ xsd:minInclusive "18"^^xsd:integer ]) .
:Adult rdf:type owl:Class ;
  rdfs:subClassOf dbpedia:Person ;
  rdfs:subClassOf [ rdf:type owl:Restriction ;
    owl:onProperty :hasAge ;
    owl:someValuesFrom :AdultAge ] .
```

Class Axioms on Properties

- ***Data Range Restrictions and Datatypes***:
  - OWL2 allows the use of XML Schema to define datatypes.
    - Only datatypes defined using XML Schema **facets** can be used in restrictions.
  - https://www.w3schools.com/xml/schema_facets.asp
  - XSD Restrictions/Facets are used to define acceptable values for XML elements or attributes

```
<xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="120"/>
  </xs:restriction>
</xs:simpleType>.
```

XSD Restrictions/Facets examples

```
<xs:restriction base="xs:string">
  <xs:enumeration value="Audi"/>
  <xs:enumeration value="Golf"/>
  <xs:enumeration value="BMW"/>
</xs:restriction>

<xs:restriction base="xs:string">
  <xs:pattern value="[a-z]"/>
</xs:restriction>

<xs:restriction base="xs:string">
  <xs:pattern value="([a-z])*"/>
</xs:restriction>

<xs:restriction base="xs:string">
  <xs:length value="8"/>
</xs:restriction>
```

XSD Restrictions/Facets examples

Constraint	Description
enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
minLength	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed. Must be greater than zero
whiteSpace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

Class Axioms on Properties

- ***Self Restrictions***:
  - good apartments will sell well

```
:GoodApartment
  rdf:type owl:Class ;
  rdfs:subClassOf [ rdf:type owl:Restriction ;
    owl:onProperty :sells ;
    owl:hasSelf "true"^^xsd:boolean; ] .
```

every instance of **`:GoodApartment`** has a **`:sells`** property with the value **`true`**

Class Axioms on Properties

- ***Keys***:
  - Databases typically use keys to identify records in a table
  - OWL2 allows us to indicate that for certain classes (read: tables) the value of a specific datatype property (or combination of properties) should be regarded as a unique identifier
    - For example, the combination of postcode and street address number will provide a unique identifier for any dwelling in the Netherlands:

```
:postcode rdf:type owl:DatatypeProperty .
:addressNumber rdf:type owl:DatatypeProperty .
:Dwelling
  rdf:type owl:Class ;
  owl:hasKey ( :postcode :addressNumber ) .
```

Class Axioms on Properties

- ***Keys***:
  - The key mechanism allows us to define ***inverse functional datatype properties*** that are local to a class
    - For example: any two individuals of type **`:Dwelling`** that have the same value for the **`:postcode`** and **`:addressNumber`** must be considered to be the same.

Individual Facts

- Statements about individuals are usually called *assertions*
- Class membership and property assertions in OWL2 are stated in the same way as in RDF Schema:

```
:Apartment rdf:type owl:Class .

:BaronWayApartment rdf:type :Apartment ;
   :hasNumberOfRooms "4"^^xsd:integer ;
   :isRentedBy :Paul .
```

- Under the semantics of OWL2 DL, the **`rdf:type`** relations may hold only between two strictly separated levels: that of classes, and that of individuals

Individual Facts

- ***Identity Assertions***:
  - OWL2 has the ***open world assumption***, that is we can never assume that two individuals with different URIs must be different entities -> we might be dealing with a single individual that has multiple names
    - Although we have seen that in some cases we can infer ***identity*** relations automatically, it is often more convenient to state them explicitly:

```
:BaronWayApartment owl:sameAs :PaulsApartment ;
   owl:differentFrom :FranksApartment .
```

Open-world assumption

- ***Open-world assumption***: we cannot conclude some statement x to be false simply because we cannot show x to be true
  - We may not deduce falsity from the absence of truth
  - Question: "Did it rain in Tokyo yesterday?"
  - Answer: "I don't know that it rained , but that's not enough reason to conclude that it didn't rain"
- ***Closed-world assumption*** allow deriving falsity from the inability to derive truth
  - Question: " Was there a big earthquake disaster in Tokyo yesterday? "
  - Answer: " I don't know that there was, but if there had been such a disaster, I'd have heard about it. Therefore I conclude that there wasn't such a disaster"

Unique-name assumption (UNA)

- Unique-name assumption (UNA): when two individuals are known by different names, they are in fact different individuals
- This is an assumption that sometimes works (ex. Product codes) and sometimes doesn't (ex. Social environment: people names)
- OWL does not make the unique-name assumption

Individual Facts

- ***Identity Assertions***:
  - The list of **different individuals** can easily grow quite long.
    - For instance, a small city will contain hundreds of apartments for which we would need to assert pairwise **`owl:differentFrom`** relations
    - We can state this a bit more elegantly using the **`owl:AllDifferent`** construct:

```
_:x rdf:type owl:AllDifferent ;
   owl:members (
      :FranksApartment
      :PaulsApartment
      :PhongsApartment
      :RahulsApartment
      :LumingsApartment
   ) .
```

Individual Facts

- *Negative Assertions*:
  - Sometimes we know something not to be the case and we want to state it in the knowledge
    - For instance, the knowledge that **`:BaronWayApartment`** is not rented by **`:Frank`** may allow us to infer that it is not **`:FranksApartment`**:

```
_:x rdf:type owl:NegativePropertyAssertion ;
   owl:sourceIndividual :BaronWayApartment ;
   owl:assertionProperty :isRentedBy ;
   owl:targetIndividual :Frank .
```

    - If the **`owl:assertionProperty`** points to datatype property, we use **`owl:targetValue`** instead of **`owl:targetIndividual`**

Individual Facts

- ***Negative Assertions***:
  - If we know that an individual is not a **`member`** of a certain class, we can also state this explicitly by asserting it to be a member of that class's complement:

```
:BaronWayApartment rdf:type
   [ owl:complementOf :LuxuryApartment ] .
```

states that **`:BaronWayApartment`** is not a **`:LuxuryApartment`**

OWL2 Profiles

- The OWL2 specification includes a number of ***profiles*** . http://ww.w3.org/TR/owl2-profiles/:
  - some are subsets of the OWL2 DL
  - some are more expressive but do not have the full semantics of OWL2 Full
- Motivation: many **existing ontologies** use only a particular **subset** of the language constructs available in DL and we can achieve a significant increase of reasoner performance by using a **less expressive** language
  - It is very useful in the practice of ontology engineering to have a standard library of logical profiles with tradeoffs between expressiveness and computational complexity

OWL2 Profiles

- These profiles are:
  - restricted by syntax
  - defined by logics that can handle at least some interesting inference service in polynomial time, irrespective to:
    - the number of facts in the ontology, or
    - the size of the ontology

OWL2 Profiles

- ***OWL2 EL* profile**:
  - is an extension of the *EL* description logic
    - polynomial time on ontologies with a large number of class axioms with conjunctions and existential restrictions
    - The most significant difference with OWL2 DL is that it drops the **`owl:allValuesFrom`** restriction, though it does support **`rdfs:range`** restrictions on properties, which can have a similar effect.

OWL2 Profiles

- **`owl:allValuesFrom`** vs. **`rdfs:range`** :

```
:Person
  a owl:Class ;
  rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty :hasParent ;
      owl:allValuesFrom :Person
   ] .
```

With **`allValuesFrom`** restriction, it's possible to say that **`Person`**s have **`Person`**-parents and that dogs have dog-parents, etc.

With domain/range, you cannot separate persons from dogs and other.

```
:hasParent rdfs:range :Person.
:hasParent rdfs:domain :Person.
:hasParent rdfs:range :Dog.
:hasParent rdfs:domain :Dog.
```

OWL2 Profiles

- ***OWL2 EL* profile**:
  - it was designed to cover the expressive power of several existing large-scale ontologies in the health care and life sciences domain
    - SNOMED-CT http://www.snomed.org
    - Gene Ontology http://www.geneontology.org
    - GALEN http://www.openclinical.org/prj_galen.html

OWL2 Profiles

- ***OWL2 RL* profile**:
  - is based on so-called ***Description Logic Programs*** and enables interaction between description logics and rules (it is the largest syntactic fragment of OWL2 DL that is implementable using rules)
    - rules can efficiently be run in parallel, allowing for scalable reasoning implementations
    - rule reasoners can easily disregard the restrictions of OWL DL (such as the separation between classes and individuals)
    - rule implementations of OWL2 RL can implement subsets of OWL Full

OWL2 Profiles

- ***OWL2 RL* profile**:
  - Many of the most scalable reasoners for Semantic Web languages implement OWL2 RL or a very similar language called pD* or OWL-Horst
  - The set of rules that have to be implemented is published as part of the OWL2 RL specification:
    - https://www.w3.org/TR/owl2-profiles/#OWL_2_RL
    - https://www.w3.org/TR/rif-owl-rl/

OWL Tools

- There are two types of tools addressing the two main stages of the ontology lifecycle:
  - *ontology editors* are used to create and edit ontologies
  - *ontology reasoners* are used to query ontologies for implicit knowledge, i.e., they determine whether a statement in question is a logical consequence of an ontology.

OWL Tools

- The currently most widely used OWL editor is Protégé, a free open-source editing framework developed at Stanford University
- Other editors:
  - TopQuadrant's commercial TopBraid Composer
  - SWOOP (open-source)
  - NeOn-Toolkit (open-source)

OWL Tools

- Reasoners for OWL DL differ in terms of coverage of the supported reasoning features
  - Test Suite Status lists to which extent some of the reasoners comply with the test cases

OWL Tools

- General-purpose reasoners aiming at supporting all of OWL DL:
  - Fact++ by the University of Manchester
  - Hermit by Oxford University Computing Laboratory
  - Pellet by Clark & Parsia, LLC
  - RacerPro by Racer Systems

OWL Tools

- Reasoning systems tailored to the tractable profiles of OWL:
  - CEL by Dresden University of Technology supports OWL EL
  - QuOnto by Sapienza Università di Roma supports OWL QL
  - ORACLE 11g supports OWL RL

OWL Tools

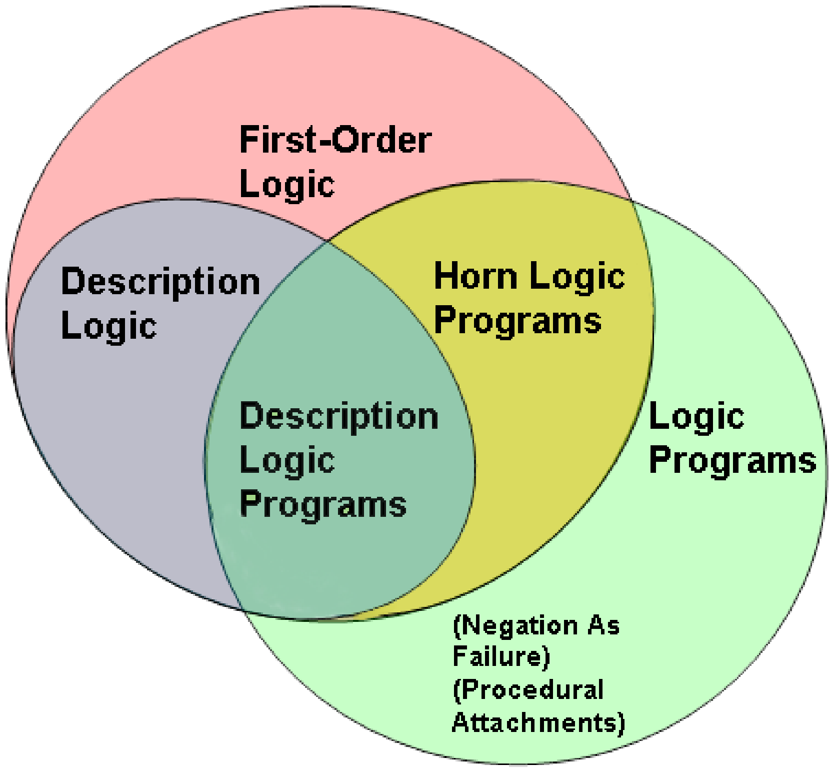
- APIs:
  - OWL API
  - Apache Jena: https://jena.apache.org/
- Other OWL tools can be found at semanticweb.org and in the ESW-Wiki

OWL Versioning Information

```
<http://www.semanticwebprimer.org/ontologies/apartments.ttl>
owl:versionIRI <http://www.semanticwebprimer.org/ontologies/apartments.ttl#1.0> ;
```

- **owl:versionInfo** generally contains a string giving information about the current version, e.g. keywords
- **owl:priorVersion** indicates earlier versions of the current ontology
  - No formal meaning, can be exploited for ontology management
- **owl:backwardCompatibleWith** contains a reference to another ontology
  - All identifiers from the previous version have the same intended interpretations in the new version
  - Thus documents can be safely changed to commit to the new version
- **owl:incompatibleWith** indicates that the containing ontology is a later version of the referenced ontology but is not backward compatible with it

Relation of OWL to other languages



Summary

- OWL2 extends RDF and RDF Schema with a number of very expressive language features, such as cardinality constraints, class equivalence, intersection, and disjunction
- Formal semantics and reasoning support is provided through the correspondence of OWL with logics.
- OWL2 comes in two flavors:
  - OWL2 DL is a language that imposes some restrictions on the combination of OWL2 and RDFS language elements to retain decidability
  - OWL2 Full is a fully compatible extension of RDF Schema with all OWL2 language features, but it is known to be undecidable
- Three profiles, OWL2 EL, OWL2 QL, and OWL2 RL, are syntactic subsets that have desirable computational properties
  - In particular, OWL2 RL is implementable using rule-based technology and has become the de facto standard for expressive reasoning on the Semantic Web.

Summary

- OWL2 has four standard syntaxes:
  - RDF/XML
  - Manchester Syntax
  - OWL/XML
  - Functional Style syntax
- References:
  - http://www.w3.org/TR/owl2-primer/
  - http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/

Summary

- Protégé, the de facto editing environment for OWL ontologies, developed by Stanford University. It has several reasoners built in. http://protege.stanford.edu
- CEL (University of Dresden), an OWL reasoner optimized for the OWL2 EL profile: http://lat.inf.tu-dresden.de/systems/cel/
- HermiT, a fast OWL reasoner for ontologies, developed at Oxford University: http://www.hermit-reasoner.com
- OWLIM, a OWL reasoner for the OWL2 RL profile, developed by Ontotext: http://www.ontotext.com/owlim
- Pellet, one of the most feature-rich OWL reasoners, developed by Clark & Parsia: http://pellet.owldl.com
- TopBraid Composer, an RDF-based editing environment for OWL ontologies, developed by TopQuadrant. It supports SPARQL, connection to triple stores, and inferencing using the OWLIM reasoner http://www.topquadrant.com/products/TB_Composer.html