Knowledge Management through Ontology

Dr. Sudeep Marwaha

Ontology is a knowledge representation technique for the semantic web. Ontologies are not intended just for storing knowledge about the subject domain but can be used by semantic web agents for inference, data integration, decision making etc. W3C semantic web workgroup has promoted Web Ontology Language (OWL) as a standard for creating ontology and it is the central layer in its semantic web architecture. At its core, the semantic Web comprises a philosophy, a set of design principles, collaborative working groups, and a variety of enabling technologies. Some elements of the Semantic Web are expressed as prospective future possibilities that have yet to be implemented or realized. Other elements of the Semantic Web are expressed in formal specifications. Some of these include Resource Description Framework (RDF), a variety of data interchange formats (e.g. RDF/XML), and notations such as RDF Schema (RDFS) and the Web Ontology Language (OWL), all of which are intended to provide a formal description of concepts, terms, and relationships within a given knowledge domain.

Following is an overview of the technologies related to ontologies:
- Resource Description Framework (RDF)
- Resource Description Framework Schema (RDFS)
- Web Ontology Language (OWL)
- Protégé
- SPARQL
- Jena API

Resource Description Framework (RDF)

RDF is the W3C standard for encoding knowledge for the Semantic Web. RDF provides a general, flexible method to decompose any knowledge into small pieces, called triples, with some rules about the semantics (meaning) of those pieces. RDF builds on existing XML and URI (Uniform Resource Identifier) technologies, using a URI to identify every resource, and using URIs to make statements about resources. RDF statements describe a resource (identified by a URI), the resource’s properties, and the values of those properties.

RDF is best thought of in the form of node and arc diagrams:

![RDF Diagram](image_url)

Below is an example of an RDF statement (triple):

```
http://dept.net/articles/rpaper1.htm has a property defined as http://www.w3.org/199/02/22-rdf-syntax-ns#type whose value is Document
```

This English statement in RDF can be divided as:

```
[subject] [predicate] [object]
```
Nodes and Arcs that represent Resources and Properties in RDF model are uniquely identified by Unique Resource Identifier (URI). Fig.  2: RDF Graph using URIs below demonstrates that RDF uses URIs to identify:

- individuals, e.g. creator of a Web page, identified by http://www.example.org/staffid/85740
- things, e.g., a Web page, identified by http://www.example.org/index.html
- properties of those things, e.g., creation date, identified by http://www.example.org/terms/creation-date
- values of those properties, e.g. August 16, 1999 as the value of the creation date property

![RDF Graph using URIs](image)

Once triples are defined graphically, they can be coded in RDF/XML to be accessed programmatically.

RDF/XML syntax used to represent RDF graphs is as follows:

- **rdf:Description** - used to define a Triple, multiple triples having same subject can be defined under one rdf:Description tag.
- **rdf:about** – used to define subject of triple.
- **Properties** are defined by their URI as tag using xml namespace e.g. – creation date property is defined using `<exterms:creation-date>`.
- **Value of property** tag can be plain literal or a resource.
- **rdf:resource** – used to define the value of a property if it is a resource.
- **rdf:datatype** – used to assign data type to literals.
- **rdf:ID** – can be used in place of rdf:about attribute if the resource URI is assigned in terms of RDF document’s base URI.

By creating triples with subjects, predicates, and objects, RDF allows machines to make logical assertions based on the associations between subjects and objects. And since RDF uses URIs to identify resources, each resource is tied to a unique definition available on the Web. However, while RDF provides a model and syntax (the rules that specify the elements of a sentence) for describing resources, it does not specify the semantics (the meaning) of the resources. To truly define semantics, RDFS and OWL are needed.
RDF Schema (RDFS)

RDFS is used to create vocabularies that describe groups of related RDF resources and the relationships between those resources. An RDFS vocabulary defines the allowable properties that can be assigned to RDF resources within a given domain. RDFS also allows creating classes of resources that share common properties.

Using the same triples paradigm defined by RDF, RDFS triples consist of classes, class properties, and values that define the classes and relationships between the resources within a particular domain.

In an RDFS vocabulary, resources are defined as instances of classes. A class is a resource too, and any class can be a subclass of another. This hierarchical semantic information is what allows machines to determine the meanings of resources based on their properties and classes.

RDFS tags are:

- **rdfs:Class**: used to define a class in RDFS.
- **rdfs:subClassOf**: used to assign a class its parent class.
- **rdfs:Property**: used to define a property.
- **rdfs:subPropertyOf**: used to assign a property its parent property.
- **rdfs:domain** and **rdfs:range**: schema properties to describe application specific properties.
- **rdfs:Resource**: RDF Schema defines all the classes as subclass of this class.

The following example shows a sample schema in RDFS:

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <rdfs:Class rdf:ID="Person">
        <rdfs:comment>Person Class</rdfs:comment>
        <rdfs:subClassOf rdf:resource="#Person"/>
    </rdfs:Class>
    <rdfs:Class rdf:ID="Student">
        <rdfs:comment>Student Class</rdfs:comment>
        <rdfs:subClassOf rdf:resource="#Person"/>
    </rdfs:Class>
    <rdfs:Class rdf:ID="faculty">
        <rdfs:comment>Faculty Class</rdfs:comment>
        <rdfs:subClassOf rdf:resource="#Person"/>
    </rdfs:Class>
    <rdfs:Class rdf:ID="Course">
        <rdfs:comment>Course Class</rdfs:comment>
        <rdfs:subClassOf rdf:resource="#Resource"/>
    </rdfs:Class>
    <rdf:Property rdf:ID="faculty">
        <rdfs:comment>Teacher of a course</rdfs:comment>
        <rdfs:domain rdf:resource="#Course"/>
        <rdfs:range rdf:resource="#Faculty"/>
    </rdf:Property>
    <rdf:Property rdf:ID="students">
        <rdfs:comment>List of Students of a course</rdfs:comment>
    </rdf:Property>
</rdf:RDF>
```
2.5 Web Ontology Language (OWL)

Building upon RDF and RDFS, OWL defines the types of relationships that can be expressed in RDF using an XML vocabulary to indicate the hierarchies and relationships between different resources. In fact, this is the very definition of “ontology” in the context of the Semantic Web: a schema that formally defines the hierarchies and relationships between different resources. Semantic Web ontologies consist of taxonomy and a set of inference rules from which machines can make logical conclusions.

Taxonomy in this context is a system of classification for classifying resources into classes and sub-classes based on their relationships and shared properties. Since taxonomies (systems of classification) express the hierarchical relationships that exist between resources, OWL can be used to assign properties to classes of resources and allow their subclasses to inherit the same properties.

All the detailed relationship information defined in OWL ontology allows applications to make logical deductions [6]. It’s important to note that OWL has three sub languages, each with increasing complexity: OWL Lite, OWL DL, and OWL Full. Developers choose which OWL dialect to use based on the level of complexity and level of detail required by their semantic model.

When RDF resource descriptions are associated with an ontology defined somewhere on the Web, intranet, or extranet, it’s possible for machines to retrieve the semantic information associated with each resource. It’s in this way that URIs, XML, RDF, RDFS, and OWL combine to make the Semantic Web a reality as shown in Fig. 3: Semantic Web - Layers.

![Fig. 3: Semantic Web - Layers](image-url)
Besides, inference tools can infer implicit knowledge using predefined rules as specified in the applied logic.

**Need of OWL over RDFS:**

OWL adds several features which enhances semantic expressibility of RDFS. List of additional features that can be defined in owl:

1. **Classes can be defined as Boolean combinations of other classes using the set operators union, intersection, and complement.**

   Set Operators: intersectionOf, unionOf, complementOf

   Classes constructed using the set operations are more like definitions than anything else. The members of the class are completely specified by the set operation. For example:

   ```xml
   <owl:Class rdf:ID="WhiteWine">
     <owl:intersectionOf rdf:parseType="Collection">
       <owl:Class rdf:about="#Wine"/>
       <owl:Restriction>
         <owl:onProperty rdf:resource="#hasColor"/>
         <owl:hasValue rdf:resource="#White"/>
       </owl:Restriction>
     </owl:intersectionOf>
   </owl:Class>
   ``

   The construction above states that WhiteWine is exactly the intersection of the class Wine and the set of things that are white in color. Without such a definition one can know that white wines are wines and white, but not vice-versa. (Note that 'rdf:parseType="Collection"' is a required syntactic element.)

   Similarly, making class Z as union of class X and class Y defines class Z as union of all the instances of class X and class Y.

   And, the complementOf construct selects all individuals from the domain of discourse that do not belong to a certain class. Usually this refers to a very large set of individuals.

2. **Classes can be stated as disjoint.**

   The disjointness of a set of classes can be expressed using the owl:disjointWith constructor. It guarantees that an individual that is a member of one class cannot simultaneously be an instance of a specified other class. The Pasta example below demonstrates multiple disjoint classes.

   ```xml
   <owl:Class rdf:ID="Pasta">
     <rdfs:subClassOf rdf:resource="#EdibleThing"/>
     <owl:disjointWith rdf:resource="#Meat"/>
     <owl:disjointWith rdf:resource="#Fowl"/>
     <owl:disjointWith rdf:resource="#Seafood"/>
     <owl:disjointWith rdf:resource="#Dessert"/>
   </owl:Class>
   ```
This code snippet states that Pasta which is a subclass of EdibleThing is disjoint with Meat, fowl, seafood, dessert, and fruit.

3. It can be stated that the two classes (with different URI) are same, and that two different instances actually represent the same individual.

**equivalentClass, equivalentProperty:** To tie together a set of component ontologies as part of a third it is frequently useful to be able to indicate that a particular class or property in one ontology is equivalent to a class or property in a second ontology.

**sameAs, differentFrom, AllDifferent**

**sameAs:** This mechanism is similar to that for classes, but declares two individuals to be identical. An example would be:

```xml
<Wine rdf:ID="MikesFavoriteWine">
   <owl:sameAs rdf:resource="#StGenevieveTexasWhite" />
</Wine>
```

It states that mike’s favourite wine is “StGenevieveTexasWhite”.

**differentFrom:** This mechanism provides the opposite effect from sameAs. For example:

```xml
<WineSugar rdf:ID="Dry" />
<WineSugar rdf:ID="Sweet">
   <owl:differentFrom rdf:resource="#Dry"/>
</WineSugar>
<WineSugar rdf:ID="OffDry">
   <owl:differentFrom rdf:resource="#Dry"/>
   <owl:differentFrom rdf:resource="#Sweet"/>
</WineSugar>
```

This is one way to assert that these three values are mutually distinct. There will be cases where it is important to ensure such distinct identities. Without these assertions one could describe a wine that was both Dry and Sweet.

**AllDifferent:** This is a convenient mechanism to define a set of mutually distinct individuals. The following asserts that Red, White, and Rose are pairwise distinct.

```xml
<owl:AllDifferent>
   <owl:distinctMembers rdf:parseType="Collection">
      <vin:WineColor rdf:about="#Red" />
      <vin:WineColor rdf:about="#White" />
      <vin:WineColor rdf:about="#Rose" />
   </owl:distinctMembers>
</owl:AllDifferent>
```

4. Cardinality restrictions can be specified for properties.
OWL provides three cardinalities constructs:

- **owl:cardinality**, which permits the specification of exactly the number of elements in a relation.
- **owl:mincardinality**, which permits the specification of the minimum number of elements in a relation.
- **owl:maxcardinality**, which permits the specification of the maximum number of elements in a relation.

5. It can be specified that a property is transitive, symmetric, Functional, inverseOf, InverseFunctionalProperty.

- **Transitive property**: If a property, P, is specified as transitive then for any x, y, and z: P(x,y) and P(y,z) implies P(x,z)
- **Symmetric property**: If a property, P, is tagged as symmetric then for any x and y: P(x,y) iff P(y,x)
- **Functional property**: If a property, P, is tagged as functional then for all x, y, and z: P(x,y) and P(x,z) implies y = z
- **inverseOf**: If a property, P1, is tagged as the owl:inverseOf P2, then for all x and y: P1(x,y) iff P2(y,x)
- **InverseFunctionalProperty**: If a property, P, is tagged as InverseFunctional then for all x, y and z: P(y,x) and P(z,x) implies y = z

Protégé

Protégé is a free, open-source platform that provides a growing user community with a suite of tools to construct domain models and knowledge-based applications with ontologies. At its core, Protégé implements a rich set of knowledge-modeling structures and actions that support the creation, visualization, and manipulation of ontologies in various representation formats. Protégé can be customized to provide domain-friendly support for creating knowledge models and entering data. Ontology describes the concepts and relationships that are important in a particular domain, providing a vocabulary for that domain as well as a computerized specification of the meaning of terms used in the vocabulary. Ontologies range from taxonomies and classifications, database schemas, to fully axiomatized theories. In recent years, ontologies have been adopted in many business and scientific communities as a way to share, reuse and process domain knowledge. Ontologies are now central to many applications such as scientific knowledge portals, information management and integration systems, electronic commerce, and semantic web services.

The Protégé platform supports two main ways of modeling ontologies:

- The Protégé-Frames editor enables users to build and populate ontologies that are frame-based. In this model, ontology consists of a set of classes organized in a subsumption hierarchy to represent a domain's salient concepts, a set of slots associated to classes to describe their properties and relationships, and a set of instances of those classes - individual exemplars of the concepts that hold specific values for their properties.
- The Protégé–OWL editor enables users to build ontologies for the Semantic Web, in particular in the W3C's Web Ontology Language (OWL). OWL ontology may include descriptions of classes, properties and their instances. Given such ontology, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the semantics. These entailments may be based on a single document or multiple distributed documents that have been combined using defined OWL mechanisms.

The Protégé-OWL editor enables users to:
• Load and save OWL and RDF ontologies.
• Edit and visualize classes, properties, and restrictions.
• Define logical class characteristics as OWL expressions.
• Execute reasoners such as description logic classifiers.
• Edit OWL individuals.

Protégé-OWL’s flexible architecture makes it easy to configure and extend the tool. Protégé-OWL is tightly integrated with Jena and has an open source Java API for the development of custom-tailored user interface components or arbitrary Semantic Web services.

Creating Crop Ontology Using Protégé

1. Creating class Cereal.
   We create ontology for crops as our prototype Expert System will be doing the reasoning and inferring on this ontology.
   To create a new class we first select the OWL classes tab on the main editor page. Next we select crops class click on subclass button to add a subclass and name it cereals. On clicking the subclass button a class explorer opens up in which we can edit name of class and other restrictions like class disjoint with newly created class.

   ![Fig. 4. Protégé Class Editor](image)

   The snapshot given above shows the subclass explorer and class editor with a newly created class namely Cereals.

2. Creating Subclass Wheat.
   As we created the main class cereals by making it a subclass of owl: thing, we can make a subclass of any given class by selecting the given class i.e. super class and using the same subclass button and create a new class using the same steps as we took while creating a class cereals.
   Protégé also provides us the drag and drop facility by which we can change the class hierarchy in a convenient way without deleting any existing class. All we need to do
is to drag the desired subclass under the parent class. Following snapshot shows a new class subclass Wheat being added to existing class cereals:

3. Creating Object Property.
Protégé provide us tools like add property; add sub property which helps us in creating object property, data type property and annotation property with equal ease. To create a property we click on properties tab on the main editor window, next we click on add object property button. On clicking add object property button a property editor window opens up which provides us the facility of editing the default property name.

The property editor provides option for explicitly specifying whether the property is functional, Inverse functional, symmetric or transitive. We can also add domain and range for a property in case of object property. Following snapshot shows property explorer and property editor and a new object property isCausedBy is being created with domain as diseases and range as Nematodes, Virus, Bacteria and Fungus.

Fig. 5. Creating a sub-class in Protégé
4. Adding restrictions
To add restriction on a class property we select the class in class explorer and click add restriction button to add restriction on desired property. Here we add a restriction on object-type property isCauseBy for diseases that it can have mincardinality as one i.e., a disease can be caused by atleast one Nematode or Virus or Bacteria or Fungus.

5. Creating Individuals
The following snapshot shows us a view of adding individual to a class. To create individuals of a class we click individual tab on main protégé window and select the class on which individuals are to be added using class browser. Next we select add individual button provided in instance browser and can fill the boxes that appear individual editor to create an individual.

![Fig. 8. Creating individuals of a class](image)

**SPARQL**

An RDF graph is a set of triples; each triple consists of a subject, a predicate and an object. SPARQL is a query language for getting information from such RDF graphs. It provides facilities to:

1. Extract information in the form of URIs, blank nodes and literals.
2. Extract RDF sub graphs.
3. Construct new RDF graphs based on information in the queried graphs.

The SPARQL query language is based on matching graph patterns. The simplest graph pattern is the triple pattern, which is like an RDF triple but with the possibility of a variable in any of the subject, predicate or object positions. Combining these gives a basic graph pattern, where an exact match to a graph is needed to fulfill a pattern.

A simple SPARQL query can be written as:

```sparql
PREFIX info: <http://somewhere/peopleInfo#>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name ?age
WHERE
  {?person vcard:FN ?name .
   OPTIONAL { ?person info:age ?age . }
   FILTER (!bound(?age) || ?age > 24 )
  }
```

The RDF file used for this example is:

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE rdf:RDF [<!ENTITY xsd 'http://www.w3.org/2001/XMLSchema#'>]>
```
<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Rishabh Munjal&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Sarah Khan&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Sumit Munjal&quot;</td>
<td>25</td>
</tr>
</tbody>
</table>
In SPARQL query:

1. **PREFIX** is a keyword that provides shorthand mechanism for writing long URIs using prefixes.
2. **SELECT** clause identifies the variables to appear in the query results.
3. **WHERE** clause specifies the triple patterns to be matched with the RDF/OWL data.
4. Other data modifiers can include:
   - **OPTIONAL** - define additional graph patterns that do not cause solutions to be rejected if they are not matched, but do bind to the graph when they can be matched.
   - **FILTER** - restricts the results of a query by imposing constraints on values of bound variables.
   - **UNION** – provides alternative matches feature to write queries that return whichever of the properties is available.
   - **FROM [NAMED]** – provide mechanism to work with multiple graphs.
   - **LIMIT** – limits the number of solution returned as the result of query.
   - **OFFSET** - causes the solutions generated to start after the specified number of solutions.

**JENA (Semantic Web Framework)**

Jena [Hewlett-Packard Development Company, 2006] is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine. The Jena Framework includes modules like RDF API, ARP, Persistence, Reasoning Subsystem, Ontology Subsystem, SPARQL query language implementation. RDF API has statement and resource centric methods for manipulating RDF model, cascading method calls for more convenient programming, built-in support for RDF containers, enhanced resources, integrated parsers and writers for RDF/XML (ARP), N3 and N-TRIPLES. ARP is Jena's RDF/XML Parser. ARP Jena2 version is compliant with the RDF Core recommendations. The Jena persistence subsystem implements an extension to the Jena Model class that provides persistence for models through use of a back-end database engine. Jena also supports a Fastpath capability for SPARQL queries that dynamically generates SQL queries to perform as much of the SPARQL query as possible within an SQL database engine. Reasoning Subsystem of the Jena includes a generic rule-based inference engine together with configured rule sets for RDFS and for the OWL Lite. The subsystem is designed to be extensible so that it should be possible to plug a range of external inference engines into Jena. Ontology Subsystem supports OWL, DAML+OIL and RDFS. A set of Java abstractions extend the generic RDF Resource and Property classes to model more directly the class and property expressions found in ontologies using these languages, and the relationships between these classes and properties, and the individuals created from them. Jena provides the ARQ query engine which implements the SPARQL query language. The implementation in Jena is coupled to relational database storage so that optimized query is performed over data held in a Jena relational persistence store.