Semantic Web and Ontologies

Marcin Synak, Maciej Dabrowski, and Sebastian Ryszard Kruk

This chapter presents ontologies and their role in the creation of the Semantic Web. Ontologies hold special interest, because they are very closely related to the way we understand the world. They provide common understanding, the very first step to successful communication. In following sections, we will present ontologies, how they are created and used. We will describe available tools for specifying and working with ontologies.

1 What is an Ontology?

The shortest possible answer is: An ontology is a specification of a conceptualization.[1]

Basically, it means that an ontology formally describes concepts and relationships which can exist between them in some community. In other words – an ontology describes a part of the world.

A concept in an ontology can represent a variety of things. A concept can be an object of any sort: person, car, building, can describe an activity or state: swimming, being busy or available, abroad. Can represent abstract concepts like time or value. There is no strict restriction what can express as a concept in our ontology. The only restriction is the real world which our ontology tries to reflect.

A relationship in an ontology represents a way in which two concepts, two things, can be connected to each other. The connection may represent some allegiance: *Dog is best friend of Man, Train needs Rails, characteristics of objects: Children are Young, Apples are Juicy, activity: Policemen chase Criminals*, etc.

The whole idea of ontology may sound similar to the concept of RDF [151]. In fact, every ontology is an RDF graph, but the difference is that the ontology sets rules, establishes facts concerning not single objects but classes of

[1] The Semantic Web Community Portal: `http://semanticweb.org`

objects. For example: *Policemen chase Criminals* could be a part of ontology – because chasing criminals is policemen's job in general. It is an established fact. *Inspector Smith chases Johnny-Sticky Fingers* could be an RDF statement compatible with this ontology. We can recognize that *Inspector Smith* represents class *Policemen* (is an individual of this class) and *Johnny-Sticky Fingers* looks like a *Thief*, who is obviously a *Criminal.* To be more specific, we could say that *Thieves* are a subclass of *Criminals.*

Ontologies have different domains and scopes. Some try to describe more general concepts, some are very specific. Hundreds of ontologies are already listed in on-line ontology libraries [53, 188].

2 Ontology Terms

There are some terms established in the field of ontologies. So far, some of these terms were used without their formal definition. Most of them are pretty straightforward and a reader can subconsciously devise what we mean when we say, for example, that "something belongs to class X". Nevertheless, it is a good idea to clearly state the meaning of words we use:

- A *Class*, sometimes called a *Concept* in the ontology is a way to represent general qualities and properties of a group of objects. If a group of objects have the same traits, that fact should be recognized and a class representing these traits should be created. A good idea is to first recognize the groups of objects and then create classes for them. For example, if height is somehow important for what we do, we could describe that all people whose height is more than 1.80m are grouped in a class called "Tall People".
- A *Subclass*, represents a part of the object group with some traits which are not common for the whole group. For example if we have a class of "Staff Members" which contains all the people working in our company, we could devise a subclass called "Management", grouping those members of our staff which work in a management department.
- An *Individual* , or an object, is a single item in our world, which belongs to some class (or many classes). "Ronald Reagan" could be an individual of "American Presidents" class, as well as "message 2341" could be an individual of "E-mails". Two different individuals represent two different objects. We must keep in mind that our individuals may represent abstract concepts (such as activities) as well as solid objects.
- A *property* is used to describe qualities common to all the individuals of a class. Properties represents relationships in ontologies. When we attach a property to a class it becomes this property's Domain. Class of objects a property points to, is its Range.
- A *property restriction* can be set to further shape properties. Some common property restrictions are cardinality and value restrictions. We may also want to specify that some property is required, etc.

3 Reasoning Over Ontologies

We have already described that an ontology provides a common way of representing knowledge about some domain. An ontology is a way to share a common understanding of information structure. Once we have common understanding, we can try to reason over this information, extract parts which are of our interest and work with them.

Reasoning is a wide term which transgresses the boundaries of the Semantic Web. It is often used in AI terminology as well as psychological works. In the Semantic Web, generally we use the term *reasoning* to describe the process of retrieving information from RDF graph.

There are many types of reasoning in the Semantic Web and one reasoning language will not fit all needs. Some types of reasoning are designed to deal with inconsistencies within the graph, other are used when available information is not complete (fuzzy reasoning). Some reasoning languages are similar in purpose to SQL-type querying languages for retrieval of information from databases.

One type of reasoning is based on using the knowledge about graph structure in form of an ontology. It usually consists of two phases:

- Inferencing phase, when additional information (additional triples) is generated from the RDF graph using a set of inferencing rules. The common example is handling transitive properties, where additional information is explicitly written.
- Querying phase, where a part of graph is retrieved. Query languages for RDF, like RDQL [196] or SPARQL [206] are still in development phase. Querying allows not only to simply match triples to given template. RDF graph can be traversed, so queries like *find me addresses of all friends of John Smith* are possible.

Reasoning/querying languages vary in their capabilities. For example, some languages allow datatype reasoning, other not.

3.1 Existing Reasoning Engines

A short description of several existing tools, implementing different languages, which can be used for reasoning:

- Jena [103] is an open source framework written in Java useful for building Semantic Web applications. It provides tools for manipulating RDF graphs, with help of OWL and RDFS technology. One of Jena's tools is a rule-based inference engine, which can be used for reasoning. Jena also supports RDQL [196], the query language for RDF.
- Sesame [198] is an open source RDF database with support for RDF Schema inferencing and querying. Sesame supports several RDF querying languages like RDQL. The most powerful is SeRQL, the language developed especially for Sesame. Sesame inferencing tools use rule definitions

stored in XML file. There is default inferencing (which takes advantage of constructs like OWL definition of transitive properties, etc.). Users can use their custom inferencer and specify their own rules.
- TRIPLE [148] is an RDF query, inference and transformation language. Its syntax uses constructs from Horn logic1 and F-logic [147]. Existing implementation of TRIPLE is based on extensible Prolog implementation, XSB [231].

4 OWL – Web Ontology Language

In the early days of the Semantic Web research there were many formalisms used to describe ontologies. It made interoperability, excepted from semantic solutions, virtually impossible.

4.1 What is OWL?

OWL, the Web Ontology Language [180, 181], is a W3C recommendation of a language for specifying ontologies. It has been designed to facilitate greater machine interpretability than previous solutions. It provides more extensive vocabulary than plain XML, RDF or RDF Schema[2] and better facilitates expressing semantics than these languages. OWL has its roots in DAML+OIL[3] web ontology language, which concepts, revised and updated, were incorporated into OWL.

Currently, the OWL is the choice for creating ontologies unless there are special reasons to use older languages like DAML+OIL (compatibility, etc.).

The practice with creating OWL ontologies showed that while OWL is very expressive, in some cases it is not convenient to use it. Sometimes we do not use all of OWLs sophisticated concepts. But applications always assume that we could and try to reason over these concepts. Designers of OWL predicted the problem and provided three OWL “species”, subsets of OWL with decreasing expressiveness. They will be described later. However, there are initiatives to provide even simpler languages than OWL Lite, the least expressive of the three.

4.2 OWL Concepts and Language Constructs

To fully understand how OWL works and what can it be useful for, we need to formulate few concepts of OWL. Described terms are frequently used in literature dealing with OWL. Definitions are roughly taken from OWL specifications.

[2] RDF Schema: `http://www.w3.org/TR/rdf-schema/`
[3] DAML+OIL: `http://www.w3.org/TR/daml+oil-reference`

- *Class.* Classes in OWL represent groups of objects with shared characteristics. We can define a class in six ways:
  1. Using the class identifier (URI) – named classes
  2. Using the *enumeration* of all individuals which are the instances of the class
  3. Using a *property restriction* – all individuals which match the restriction form a class
  4. As an *intersection* of two or more classes – individuals common to these classes form a new class
  5. As an *union* of two or more classes – the sum of all individuals belonging to these classes form a new class
  6. As a *complement* of a class – contains all individuals which do not belong to this class
- *Class extension.* In OWL terminology, the class extension is simply a set of all individuals which form the class.
- *Class axioms.* The "truths" about classes. The simplest (and the least informative) class axiom is the class declaration which states the existence of the class. Other OWL language constructs used to define class axioms are:
  `rdfs:subClassOf` – (inherited from RDFS) means that the extension of the class is a subset of extension of the superclass
  `owl:equivalentClass` – means the equivalent class has exactly the same extension
  `owl:disjointWith` – means that the two disjointed classes have no common members in their extensions
- *Property.* Properties are used to describe classes. OWL distinguishes two types of properties:
  - Object properties (`owl:ObjectProperty`), which link individuals to individuals
  - Datatype properties (`owl:DatatypeProperty`), which link individuals to data values
- *Property extension.* Property extension is a concept similar to the class extension. It is a set of all pairs of individuals (subject and object) which can be connected with the property.
- *Property axioms.* Similarly to classes, OWL defines property axioms:
  `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range` (inherited from RDFS) – a subproperty has the extension which is a subset of superproperty extension. Domain states that subjects of the property must belong to the extension of pointed class. Range states the same, but refers to objects.
  `owl:equivalentProperty` and `owl:inverseOf` – two properties are equivalent if they have the same extension. One property is inversion of another if for every pair (X, Y) in its extension there is a pair (Y, X) in the extension of inverted property.
  `owl:FunctionalProperty` and `owl:InverseFunctionalProperty` – global cardinality constraints, functional property can have exactly one object

for a particular subject (e.g. every person has exactly one year of birth). Inverse functional properties can have exactly one subject for a particular object (e.g. there is only one X in “X is a father of John”).
`owl:SymmetricProperty` and `owl:TransitiveProperty` – describe logical characteristics of properties. A property is symmetric if for every (X, Y) pair in the property extension there is a pair (Y, X). Transition means that if there are pairs (X, Y) and (Y, Z) in the property extension, then the pair (X, Z) also belongs to this property extension.

- *Individuals.* Individuals are defined using individual axioms. There are two types of axioms (facts, truths):
  Axioms referring to class membership and property values
  Axioms referring to individual identity (`owl:sameAs`, `owl:differentFrom` - OWL can define that two individuals, two URIs, are actually the same thing or explicitly state that this is not the case).
- *Data types.* OWL uses RDF data types (which refer to XML Schema datatypes) and some of its own constructs like enumerated datatype.

4.3 OWL ‘Species’: OWL Lite, OWL DL and OWL Full

There are three types or ‘species’, as authors call them, of OWL. That includes:

- OWL Full (or simply OWL, the ‘Full’ word has been added to distinct it from the other two OWL languages)
- OWL DL (which is an abbreviation of ‘Description Logic’)
- OWL Lite

All three OWLs use the same syntactic constructions to build ontologies, the differences are in restrictions which are put on OWL DL and OWL Lite. OWL Full takes advantage of all RDF/RDFS constructs. For RDF developers, transition from RDF to OWL Full is natural, since most of RDF data will translate directly into OWL Full ontologies. To be compatible with OWL DL or Lite, RDF data has to be specifically constructed for these ontology languages.

Basically, the main restriction which does not exist in OWL Full (and exists in the other two) is distinction between a class and an individual. In OWL Full classes can be individuals at the same time. This gives the flexibility of RDFS and allows to represent complicated real-world constructs but in the same time OWL Full ontologies are very difficult to reason over.

Next is OWL DL. Description Logic [64] is an existing segment of business applications and OWL DL was created to support it by providing desirable computational properties for reasoning systems.

The list of restrictions which are placed upon OWL constructs is:

- Class definitions must be constructed using `owl:Class` or `owl:Restriction`. Other definitions can not be used independently (only together with

these two). A class can *never* be an individual at the same time (it is possible in RDFS or OWL Full).
- Datatype subproperties can only be constructed from datatype properties and object subproperties from object properties.
- A datatype property can not be inverse/functional.
- Transitive properties can not have specified cardinality.
- In OWL DL we can use annotation properties (to annotate our ontology) only with classes, properties, individuals and ontology headers. Annotation properties can not be datatype or object properties at the same time (they can be in OWL Full), can not have subproperties, can not have range and domain, etc.

Every OWL DL ontology is a valid OWL Full ontology (but not the other way around).

The last, OWL Lite, is, as its name suggests, the least sophisticated (and the least expressive) dialect of OWL. Of course, similarly as in the OWL DL case, every OWL Lite ontology is a valid OWL DL ontology and by transience is a valid OWL Full ontology. OWL Lite was created with software developers in mind, it is easy to implement and use. However, there is a push to create even simpler language for tool developers who would want to support OWL and want to start with something uncomplicated. It is sometimes called OWL Lite- (OWL Lite minus [57]).

OWL Lite has all the restrictions on OWL language constructs and add a few of its own:

- There are no enumerations as class descriptions in OWL Lite
- `owl:allValuesFrom` and `owl:someValuesFrom` used to define classes by property restrictions must point to a class name
- `owl:hasValue` property restrictions can not be used in OWL Lite
- cardinality constraints on properties in OWL Lite can only be set with "0" or "1" value (no multiple cardinality in OWL Lite)
- unions and complements of classes can not be used as class descriptions, intersection use is restricted to named classes and property restrictions
- class equivalence in OWL Lite can be only applied to classes defined with URI and point either to a named class or property restriction
- class disjointness can not be specified using OWL Lite
- domain and range of a property must point to a class name

Examples of OWL constructs which are or are not legal in OWL DL and/or OWL Lite can be found in OWL language reference document [182].

4.4 OWL vs. RDFS – Advantages, Differences

OWL is a build-up on RDFS. It should be considered as a set of ontology building tools which are not provided by plain RDFS. Language constructs of OWL allow to specify such facts like class disjointness or equivalence, defining classes by setting property restrictions or enumerations. At the same time, user has the "freedom" of RDFS like specifying classes as individuals of other (meta)classes (assuming he uses OWL Full).

We could say that OWL constructs are more defined RDFS constructs, with more "finesse". For example `owl:Class` is defined as a subclass of `rdfs:Class`.

RDFS lets us specify a RDF vocabulary. OWL allows us to *ontologize* this vocabulary.

4.5 DAML+OIL

DAML+OIL [51] grows from earlier DAML [211], the DARPA Agent Markup Language and OIL [178], the Ontology Inference Layer. DAML was developed as an extension to XML and RDF. It allowed more sophisticated class definitions than RDFS. OIL was another initiative for providing more complicated classifications which used constructs from frame-based AI.

These two efforts were merged and DAML+OIL language for specifying ontologies was a result.

DAML+OIL introduced such concepts like class equivalence, property uniqueness (concept later broadened in OWL – functional properties), property restrictions, etc. It also redefined the concept of `rdfs:Class`.

DAML+OIL is worth mentioning because it made a way for OWL. OWL is simply a revised and corrected successor of DAML+OIL which incorporates experiences gathered while working with DAML OIL ontologies.

4.6 Ontology Example

Figure 1 shows an example, simple ontology in OWL.

The example ontology describes a class *MaturePerson* which gathers all people who we considered mature (e.g. age 18 and older). A person may be either *Male* or *Female* which are defined as subclasses of *MaturePerson*. An individual of *Male* class can not be an individual of *Female* class at the same time, so class *Male* is disjoint with *Female*.

The example specifies several properties. Every person has properties *hasAge* and *hasFriend*. A person can only have one age but many friends, so *hasAge* is specified as functional property. But being someone's friend also means that this person is also our friend, so *hasFriend* has been defined as a symmetric property. *hasAge* connects an individual to an integer, so it is specified as datatype property. *hasFriend* connects two individuals, so it is an object property.

There are also specified properties *hasWife* and *hasHusband*. A Male can have a wife and a *Female* can have a husband. If we are interested only in monogamous relationships, these two properties are inverse and functional – *I'm a husband of my wife and there is exactly one person which can be my spouse* (at a certain moment in time).

Some concepts were removed from the example to keep it relatively small.

```
<?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#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="MaturePerson"/>
  <owl:Class rdf:about="#Male">
    <owl:disjointWith rdf:resource="#Female"/>
    <rdfs:subClassOf rdf:resource="#MaturePerson"/>
  </owl:Class>
  <owl:Class rdf:about="#Female">
    <owl:disjointWith rdf:resource="#Male"/>
    <rdfs:subClassOf rdf:resource="#MaturePerson"/>
  </owl:Class>
  <owl:SymmetricProperty rdf:ID="hasFriend">
    <rdfs:domain rdf:resource="#MaturePerson"/>
    <rdfs:range rdf:resource="#MaturePerson"/>
    <rdf:type rdf:resource="&owl;#ObjectProperty"/>
  </owl:SymmetricProperty>
  <owl:FunctionalProperty rdf:ID="hasAge">
    <rdfs:domain rdf:resource="#MaturePerson"/>
    <rdfs:range rdf:resource="&xsd;#positiveInteger"/>
    <rdf:type rdf:resource="&owl;#DatatypeProperty"/>
  </owl:FunctionalProperty>
  <owl:InverseFunctionalProperty rdf:about="#hasWife">
    <rdfs:range rdf:resource="#Female"/>
    <rdfs:domain rdf:resource="#Male"/>
    <rdf:type rdf:resource="&owl;#ObjectProperty"/>
    <owl:inverseOf rdf:resource="#hasHusband"/>
  </owl:InverseFunctionalProperty>
  <owl:InverseFunctionalProperty rdf:ID="hasHusband">
    <rdfs:range rdf:resource="#Male"/>
    <rdfs:domain rdf:resource="#Female"/>
    <rdf:type rdf:resource="&owl;#ObjectProperty"/>
    <owl:inverseOf rdf:resource="#hasWife"/>
  </owl:InverseFunctionalProperty>
</rdf:RDF>
```

Fig. 1. Example ontology in OWL

5 Developing Ontologies

Developing an ontology is a job which requires cooperation of both IT specialists and domain experts. The first group has experience in building applications using the technologies of the Semantic Web. The second group knows the field which will be covered with the ontology.

A goal for an ontology is to efficiently and accurately describe a 'part of the world'. This goal can not be achieved without extracting information about this realm first. E.g. if we want to create an ontology to improve postal delivery services, we need to ask people involved what are the parts of their job. For example, we need to know:

- What is the difference between a letter and a parcel?
- How does the fee system work?
- What are responsibilities of a postman? etc.

Some of these things may be obvious but in most cases they are not. For example, most people know what a 'letter' is. But how many know what it means to 'porto' pay for a letter? Specialists on every field have their own dictionary of terms and use them accordingly to their regulations. If such dictionary does not exist, the case is even more challenging, because it must be created first.

When it comes to making two or more groups of people work with different knowledge, experience and a way of seeing things together, there is always a question of tools which could support their interactions. Two IT specialists will probably find their common language quickly, because they understand code, standardized diagrams and so on. It is the same case with two postmen. But how to improve interactions between these two groups? Of course we have traditional ways of communication – letters, telephones or simple face-to-face talk. But more than often we have to make people from different cultures or from different time zones cooperate. A need for a set of tools which could improve that contact is apparent.

The described problem is common when we develop specialized software for specific needs. But it is especially clear when it comes to developing ontologies. An ontology can become 'the ontology' only if a compromise between all the parties interested is achieved. It is not only a problem of developing a specialized piece of software, but of developing a standard.

5.1 Common Steps of an Ontology Development Process

We have decided that an ontology could bring new quality to our project. We have IT specialists, specialists from our example post office, all necessary software and tools. Now, where to start?

Here is some general knowledge about developing ontologies. We know some rules which are always useful, techniques giving good results. However, none of existing techniques of developing an ontology could be described as the best or the easiest. Selection may depend on the domain we create an ontology for and any specific requirements. We may also want to ask developers, what are they most comfortable with.

Nevertheless, we can try to systematize ontology development by dividing the process into phases. Developers of Protegé [212] ontology editor, from Stanford University propose in [160] steps as follows:

1. Determine the ontology domain and scope
   First, as in everything we do, we have to ask ourselves what exactly we want to do. It is no different when creating an ontology. We need to know what we want our ontology to cover, what are we going to use it for, etc. By determining the domain, we restrict our interest to certain field of knowledge. By defining the scope we choose what part of this field is important to us.
   A good practice in determining the ontology domain and scope is creating a set of so called *competency questions*. Competency questions are questions we want our ontology to be able to answer. The example competency questions for postal delivery services could be:
   - Is it possible to send a 50 kg parcel to Zimbabwe till next Friday?
   - Can I insure it?
   - Does the postman in my vicinity deliver letters on Saturday?
   - Does delivering a letter on Saturday cost more?
   - What is the weight/cost ratio for local deliveries?
   - Can I get a return receipt?

   The competency questions are a good starting point of evaluating the ontology after it has been created. They should be considered as a necessary but not sufficient condition.
2. Determine which existing ontologies we will reuse (if any)
   There is a number of existing ontologies and vocabularies covering different fields. Many are free to use, so it is always a good idea to put a bit of work in researching existing solutions and evaluating their usability for our project. It might be possible that our ontology could just refine an existing one instead of defining all the concepts from scratch. We can use some concepts directly or inherit from them. An example could be FOAF (Friend-Of-A-Friend [32]) ontology, which provides a good way to describe human resources in the Web.
   Reusing an existing ontology might also be a requirement. Especially if our system is to interact with other systems based on controlled vocabularies, or are committed to some existing ontologies.
3. Gather important terms
   This step consists of creating a vocabulary of terms from a domain we want to describe with an ontology. Basically, we want to gather words which are being used everyday by people working in our domain. For example, if we wanted to create a bank ontology, we would probably write down things like *bank*, *account*, *client*, *credit*, *balance*, *cash* and so on. In this step we are

not concerned what is the meaning of these words or how concepts they represent are the connected to each other. What is important is to get a comprehensive list of terms which will be a base for further work.

4. Define classes
   This and next step are closely intertwined to each other. It is very hard to define a class structure before defining properties. Typically, we define few classes in the hierarchy and their properties before moving to the next group of classes, etc. Some possible approaches of developing a class hierarchy are:
   - Top-down approach, which starts with creating definition of the most general concepts and then their specialization (creating subclasses); the process is recursive for every class until it we reach the most specific definitions.
   - Bottom-up approach, which goes the other way – first we define the most specific concepts ant then group them into more general concepts by creating common superclass for them.
   - Combination of both when we start with few general (or 'top-level') concepts and few specific (or 'bottom-level') concepts and fill the middle levels consequently.

   Choosing the 'correct' approach may depend on the domain or the way the developer sees it. The combination approach is often the easiest, because middle-level concepts tend to be the most descriptive [160].
   There are two important issues which are often encountered while defining class hierarchy. First is to distinguish between classes and instances. We must decide if some concepts represent a subclass or an instance of a certain class. For example we could muse if a concept of *Letter_to_Denmark* should be described as subclass of *Letter* or its instance. While solving such problems we should decide what level of specification our ontology needs. Are we interested in describing single letters which have an address, sender address, etc. or do we only need to sort them?
   The second problem is to decide which characteristics of concept should be represented in a number of subclasses and which should be put into properties. For example there is a limited number of countries in the world. We could create subclass for every one of them indicating letter destination or simply put *destinationCountry* property in *Letter* class.
5. Define class properties
   Properties (or slots) describe internal structure of concepts. A property may point to simple value such as a string or a date (datatype properties) or a class instance (object properties). We use datatype properties to describe object's physical characteristics (often called 'intrinsic' properties) such as parcel's weight as well as abstract concepts (or 'extrinsic' properties) i.e. *deliveryDate* for letter or name for postman. Object properties are commonly used to represent relationships between individuals. A postman could have the *hasBoss* or *worksAt* object property. Of course we

do not restrict usage of object properties to represent relationships only. For example a postman could have property *workSchedule* pointing to an individual, because schedule is a complex object. Object properties are also often used to represent object's parts, if the object is structured.
Properties are inherited by all subclasses. For example, if *PostWorker* has property *name*, and *Postman* is a subclass of *PostWorker*, it will inherit the *name* property. That is why properties should be attached to the most general class that can have it. But we must keep in mind that 'inheriting' means something different in ontologies that it means, e.g. in object-oriented languages. If a subclass inherits a property from its superclass it only means that this property *can* also be applied to this subclass. It is not a requirement.

6. Define properties restrictions
Property restrictions (or 'slot facets') work together with properties. We use them to further specify usage of the property, its features, allowed values, etc. Types of facets available strongly depend on the ontology language used. Some languages are more expressive than others and have more extensive vocabulary of ready-to-use constructs. The common facets are:
   - *Slot cardinality* specifies how many values a property can have. For example, we can specify that a *PostVan* has exactly one *driver* and can carry no more than 10 *mailBags* (if we treat bags as individuals, i.e. A bag can have *ID*, *destination* etc.). Cardinality is handled differently in different ontology languages. For example, RDFS allows only specifying *single cardinality* (minimum cardinality of 0 or 1). OWL allows specifying *multiple cardinality.* It means that every property can be more precisely described using minimum and maximum cardinality. While describing a fact that every van needs a driver requires system supporting single cardinality only, we need a support for multiple cardinality to represent the number of bags.
   - *Slot value-type* defines what kind of values a property can have. Common value types for datatype properties are *String*, *Number* (or more specific types such as *Integer* or *Float*), *Boolean* and *Enumerated* (a list of specific allowed values such as express or *airMail* for *letterType* property). Object properties have Instance-type value. A list of allowed classes from which instances can come is defined.
   - *Domain and range* are facets of object properties. A list of allowed classes as described above is often called *range* of a slot. A list of classes a property is attached to is called *domain.*[4]
7. Create instances
Creating instances is the last step. We choose a class, create an individual of the chosen class and fill in property values. Instances form the actual semantic description.

[4] A *property domain* should not be confused with *ontology domain*, which should be understood as ontology subject, domain of interest.

5.2 Ontology Development Tools

This subsection describes shortly existing solutions for ontology development. Detailed information could be found at referenced websites.

Ontology Editors

- Protegé [212] originates from Stanford University. It is a free, open source ontology editor, written in Java which allows user create ontologies in RDFS and OWL languages. Protegé allows to manipulate classes using a tree-like structure (creating classes, subclasses, attaching properties, etc.). Protegé itself is a standalone tool and does not support cooperative ontology development. However, Protegé is easily extensible through a plug-in system, so appropriate tools can be added. At the moment, Protegé sets a standard for ontology editors.
- Ontolingua [177] is a distributed collaborative environment for ontology browsing, creation, editing, etc. Ontolingua server is maintained by the Knowledge Systems, AI Laboratory at Stanford University. Part of Ontolingua is the Ontology Editor which allows to create and edit ontologies using the web browser. Access to Ontolingua server is free of charge, but requires registration. Currently there is no way of creating your own server with Ontolingua software. Server maintained by Stanford (and two other at universities in Europe) hosts several ontology development projects.

There are other ontology editors like OILed [171], designed for developing DAML+OIL ontologies or DOE [62] (Differential Ontology Editor), focusing on linguistics-inspired techniques, designed to complement advanced editors like Protegé.

Merging/Mapping Tools

- Chimæra [43] is a part of Ontolingua software package from Stanford University. Chimæra's major function is merging ontologies together and diagnosing individual or multiple ontologies. Chimæra can be used for resolving naming conflicts, reorganizing taxonomies, etc.
- SMART/PROMPT [161] is an algorithm and tool for automated ontology merging and alignment. It may be used to detect conflicts in the ontologies, suggest resolutions and guide a user through the whole process of merging/aligning ontologies. PROMPT (SMART is the former name for PROMPT) is implemented as an extension to Protegé ontology editor.

Acknowledgement

This material is based upon works supported by Enterprise Ireland under Grant No. ILP/05/203 and by Science Foundation Ireland Grant No. SFI/02/CE1/I131.