188.399 Einführung in Semantic Systems

Author:

Max Tiessler

2025-01-08

Contents

1 Introduction

This summary is for the course "Introduction to Semantic Systems" (course code: 188.399) at the Vienna University of Technology (TU Wien) and was created for the Winter Semester 2024 in preparation for the exam. The document covers most of the essential topics discussed in the course, focusing on key concepts and practical applications in semantic systems. While it aims to provide a comprehensive review of the material, some advanced topics might not be explored in full depth.

This summary is available on GitHub. If you find typos, errors, want to add content (such as additional explanations, links, figures, or examples), or wish to contribute, you can do so at the following repository:
`https://github.com/mtiessler/188.399-Introduction-To-Semantic-Systems-Summary-TUW/blob/main/ISS_Summary.pdf`.

If the URL does not work, you can locate it on GitHub using the following details:

- **User**: mtiessler
- **Repo-Name**: 188.399-Introduction-To-Semantic-Systems-Summary-TUW

2 What is an Ontology

Ontology is derived from the Greek words *ontos* (being) and *logos* (word), meaning the study or systematic explanation of existence.

In philosophy, ontology is a branch concerned with categorizing and explaining existence. Aristotle (400 BC) made early attempts to establish universal categories for classifying everything that exists.

2.1 Definition

According to the **Merriam-Webster dictionary**:

- A branch of metaphysics concerned with the nature and relations of being.
- A particular theory about the nature of being and kinds of existents.

In the context of knowledge representation, **Studer (1998)** defines ontology as seen in figure 1. This definition highlights several key aspects:

- **Formal**: Ontologies are machine-readable and interpretable.
- **Explicit**: Concepts, properties, functions, and axioms are clearly defined.
- **Shared**: Ontologies are agreed upon and shared by a community.
- **Conceptualization**: They provide an abstract model of some phenomena in the world.

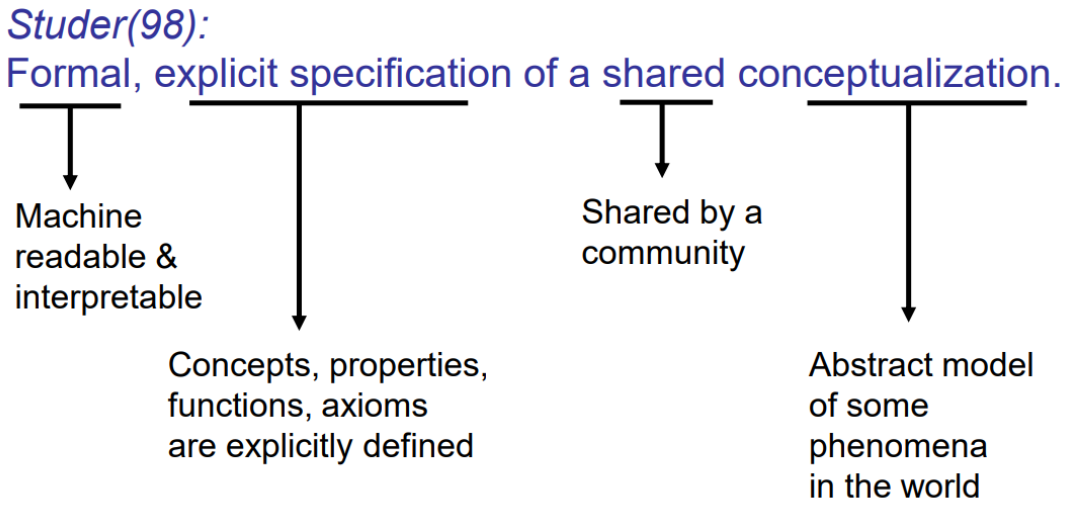

Figure 1: Ontology definition

2.2 Types and Categories

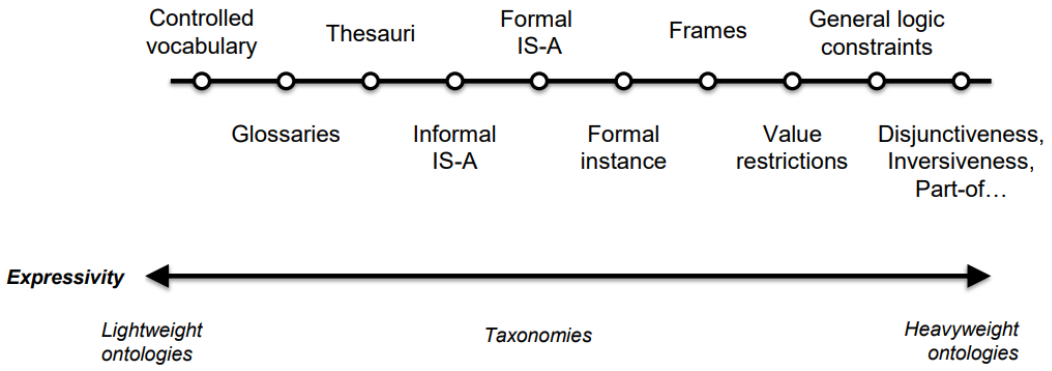

Figure 2: Ontology Types

2.2.1 Lightweight

Lightweight ontologies are simplified frameworks for organizing and defining knowledge. Key types include:

- **Controlled Vocabulary**: A finite list of terms, often used in catalogs.
  - **Example**: A library catalog listing book titles, authors, and genres (e.g., "Fiction," "Science," "Biography").
- **Glossary**: A controlled vocabulary with informal definitions provided in natural language.
  - **Example**: A glossary of medical terms, such as:
    - "Hypertension: High blood pressure."
    - "Cardiology: The branch of medicine dealing with the heart."
- **Thesauri**: A controlled vocabulary where concepts are connected through various relations:
  - **Equivalency**: Synonym relations between concepts.
    - **Example**: "Automobile" and "Car."
  - **Hierarchies**: Subclass and superclass relationships.
    - **Example**: "Dog" is a subclass of "Animal."
  - **Homographs**: Handling of homonyms (terms with identical spellings but different meanings).
    - **Example**: "Bank" (a financial institution) vs. "Bank" (the side of a river).
  - **Associations**: Relations between similar or related concepts.
    - **Example**: "Wine" is related to "Grape."

A popular example of a lightweight ontology is the "Friend of a Friend" (FOAF) ontology, which defines relationships between people in social networks.

2.2.2 Taxonomies

Taxonomies provide a hierarchical system of grouping concepts or entities. The term originates from the Greek words *taxis* (order, arrangement) and *nomos* (law, science). Types of taxonomies include:

- **Informal IS-A Hierarchy**: An explicit hierarchy of classes where subclass relations are not strict. For example, the index of a library.
- **Formal IS-A Hierarchy**: An explicit hierarchy of classes with strict subclass relations.
- **Formal Instance**: A hierarchy that includes explicit class structures, strict subclass relations, and allows *instance-of* relations.

A taxonomy can be defined as a controlled vocabulary organized into a hierarchical structure. As seen in 3.

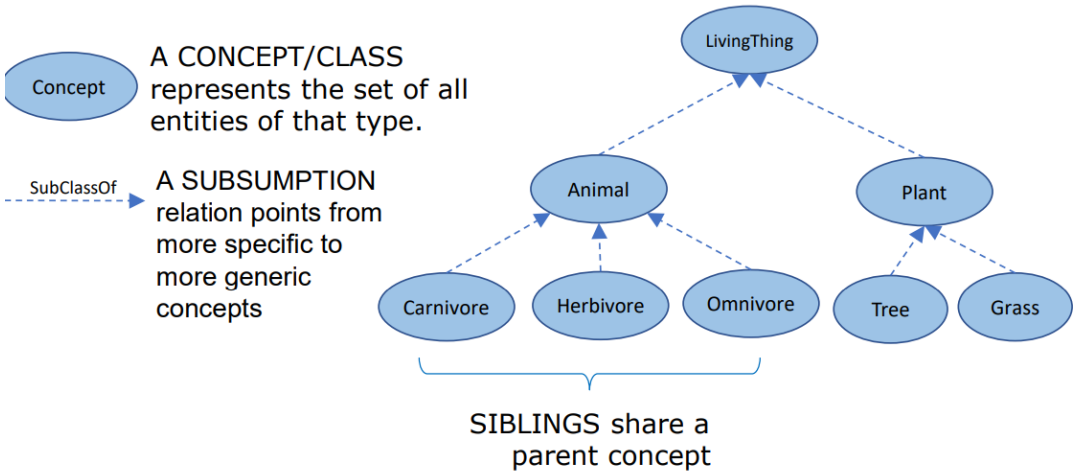

Figure 3: Taxonomy Example

Key concepts in formal taxonomies include:

- **SubClassOf**: A relation pointing from a more specific concept to a more generic concept.
- **Concept/Class**: Represents the set of all entities of a specific type.
- **Subsumption**: A specialization relation pointing from a specific concept to a generic one.

Subsumption/SubClass/Specialization Relation (in Formal IS-A Taxonomies) Semantics (Meaning): Subsumption in formal IS-A taxonomies ensures that if one class is a subclass of another, all instances of the subclass are also instances of the superclass. For example, if *Herbivore* is a subclass of *Animal*, all instances of *Herbivore* are also instances of *Animal*.

Common Mistake: Subsumption is often mistakenly used to represent other types of relations, such as *part-of* (meronymy), which denotes a part-to-whole relationship rather than a class hierarchy.

2.2.3 Heavyweight

Heavyweight ontologies provide a rigorous framework for defining and organizing knowledge with a strong emphasis on logic and formal specifications. Key characteristics include:

- **Rigorous Definition and Organization of Concepts**: Concepts and their relationships are precisely defined, ensuring clarity and reducing ambiguity.
- **Focus on Logic**: Heavyweight ontologies prioritize formally correct deductions and inferences, enabling reliable reasoning and decision-making.
- **Careful Formal Specification**: A formal specification ensures consistency and eliminates contradictions within the ontology.

Heavyweight ontologies are often used in applications requiring advanced reasoning, such as artificial intelligence, semantic web technologies, and complex domain modeling. They aim to create a shared, consistent understanding of knowledge that can be processed by both humans and machines.

Examples of Heavyweight Ontologies

- **Frames**: Frames provide structured representations for defining classes and their properties.
  - **Example**: In a medical ontology, the frame for "Disease" might include properties such as "Symptoms," "Causes," and "Treatment."
- **Value Restrictions**: These enforce constraints on the values that properties can take.
  - **Example**: In a wine ontology, "Wine color" might be restricted to values like "Red," "White," or "Rosé."
- **General Logic Constraints**: These involve the application of logical rules to ensure consistency across the ontology.
  - **Example**: If "Animal" is defined as disjoint from "Plant," no instance can belong to both classes simultaneously.
- **Disjunctiveness**: Ensures that certain concepts are mutually exclusive.
  - **Example**: "Male" and "Female" in a gender ontology might be disjoint concepts.
- **Inversiveness**: Defines inverse relationships between properties.
  - **Example**: If "Parent" is a property, its inverse would be "Child."
- **Part-of Relationships**: These express compositional relationships.
  - **Example**: "Engine" is a part of "Car."

2.3 Ontology

Ontology is a **taxonomy** extended with additional relations and further constraints, providing a more comprehensive framework for modeling knowledge. Relations within an ontology are directed, pointing from a **Domain** concept to a **Range** concept. These relations can also include:

- **Inverse Relations**: Relations that allow bidirectional reasoning. For example, if "Tom eats Jerry," the inverse relation could be "Jerry is eaten by Tom."
- **Disjoint Concepts**: Concepts that describe non-overlapping instance sets, ensuring that instances belong to distinct categories. For example, *Carnivore* and *Herbivore* may be disjoint concepts.

2.3.1 Mechanics

The mechanics of an ontology focus on the logical and structural rules governing the relationships and organization of concepts, as seen in figure 4:

- Establishing the **Domain** and **Range** of relations to clarify which concepts are linked.
- Defining and maintaining **consistency** among concepts, relations, and instances.
- Implementing **constraints**, such as disjointness or cardinality, to refine the ontology's structure and enforce rules.

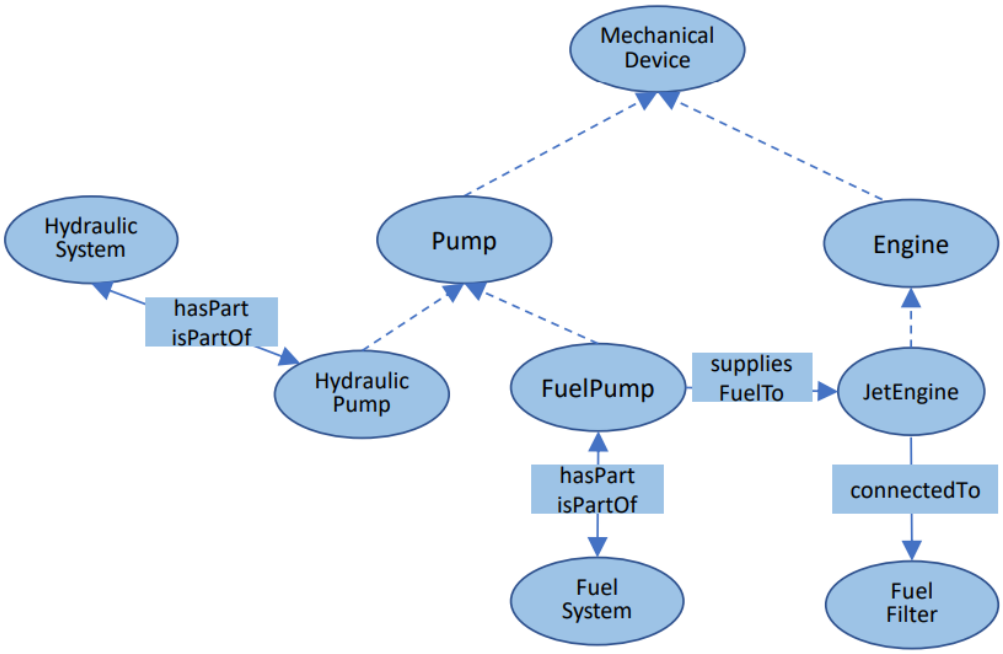

Figure 4: Ontology Mechanics

2.3.2 Instances/Entities

Instances or entities represent individual objects in the universe of discourse. They provide specific examples or metadata tied to the defined concepts in an ontology. This can be seen in 5.

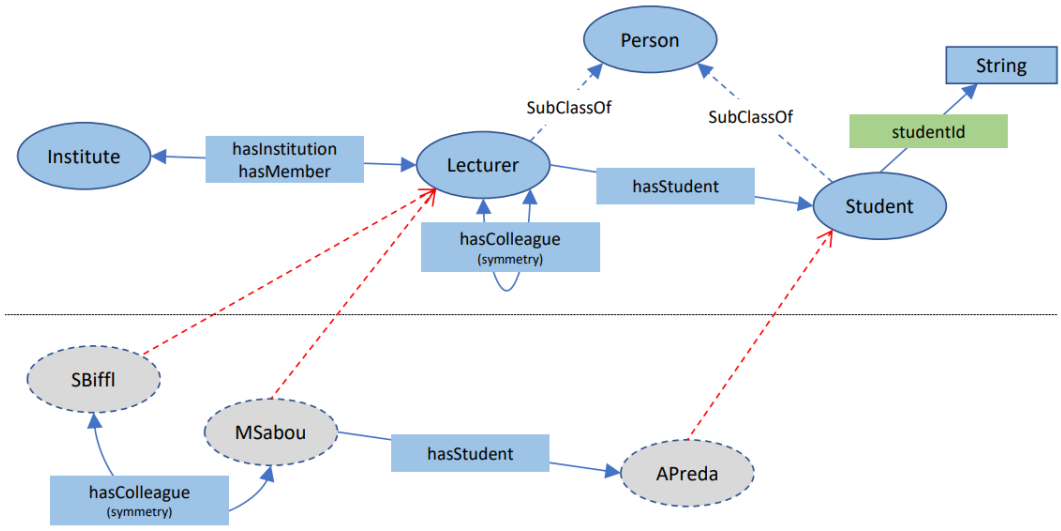

Figure 5: Ontology instances

Key aspects include:

- **Instance Representation**: An instance corresponds to a real-world entity and is associated with one or more concepts. For example:
  - *Tom is a Carnivore.*
  - *Jerry is an Animal.*
  - *Tom eats Jerry.*
- **Typing**: An instance is typed with respect to a concept, meaning that it is classified as being of a particular type. For example, "Tom" is an instance of the concept *Carnivore*.
- **Relations Between Instances**: Instances are connected through defined relations within the ontology, such as "eats" in the example above.

2.4 Description Logics

Description Logics (DL) are a family of formal knowledge representation languages used as the foundation for creating machine-readable ontologies. They are particularly suited for the Semantic Web and are based on a subset of First-Order Logics. DL provides a framework for representing and reasoning about knowledge in a structured and formal way.

2.4.1 Introduction

Description Logics aim to:

- Represent knowledge in a way that is both human-readable and machine-readable.
- Enable computers to reason with data and draw logical conclusions.
- Provide a formal basis for defining and organizing ontologies.

A formal, logic-based language in DL offers:

- **Syntax**: Defines which expressions are considered valid.
- **Semantics**: Provides the meaning of those expressions.
- **Calculus**: Defines how to compute and determine the meaning of expressions.

Semantic Web languages, such as OWL (Web Ontology Language), are built on Description Logics.

2.4.2 General DL Architecture

The general architecture of a Description Logic system (can be seen in 6), involves the following components:

- **Knowledge Base (KB)**: Composed of:
  - **TBox** (Terminological Knowledge): Contains definitions of concepts, attributes, and properties of a domain. For example:

    $$Writer \equiv Person \sqcap \exists author.Book$$

  - **ABox** (Assertional Knowledge): Contains information about specific individuals or entities. For example:

    $$Writer(GeorgeOrwell)$$

- **Inference Engine**: A component that reasons with the knowledge base to infer new information or check consistency.
- **Interface**: The mechanism for users or systems to interact with the ontology.

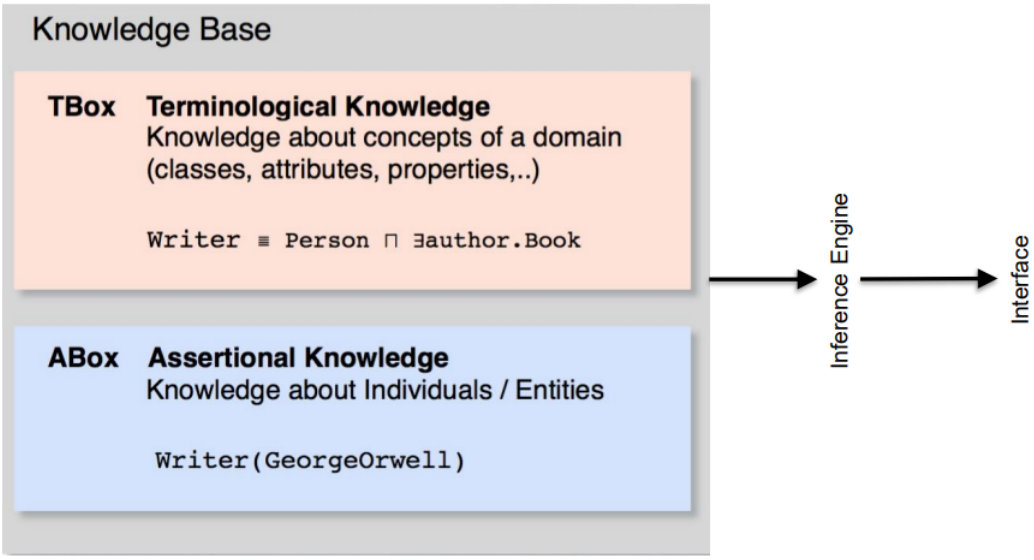

Figure 6: DL architecture

2.4.3 Notation / Syntax

The syntax of DL includes:

- **Atomic Types**:
  - Concept Names: $A, B, ...$
  - Special Concepts:
    - $\top$: Universal concept (all objects in the domain).
    - $\bot$: Bottom concept (empty set).
  - Role Names: $R, S, ...$
- **Constructors**:
  - Negation: $\neg C$
  - Conjunction: $C \sqcap D$
  - Disjunction: $C \sqcup D$
  - Existential Quantifier: $\exists R.C$
  - Universal Quantifier: $\forall R.C$

Examples:

- $Writer \equiv Person \sqcap \exists author.Book$
- $Novel \equiv Prose$
- $Novel \sqsubseteq Book$
- $Herbivore \sqsubseteq \neg\ Carnivore$
- $PetOwner \sqsubseteq Person \sqcap \exists hasPet.Animal$
- $Carnivore \sqsubseteq Animal \sqcap \forall eats.Animal$

2.4.4 Semantics

Description Logics rely on model-theoretic semantics, where an interpretation consists of:

- A **domain of discourse** (Δ): A collection of objects.
- **Interpretative Functions** (I): Functions mapping:
  - Classes (concepts) to sets of objects in the domain.
  - Properties (roles) to sets of pairs of objects.

In a DL, a class description characterizes the individuals that are members of that class, allowing precise reasoning and inference based on defined relationships and constraints.

It can be better seen in 7:

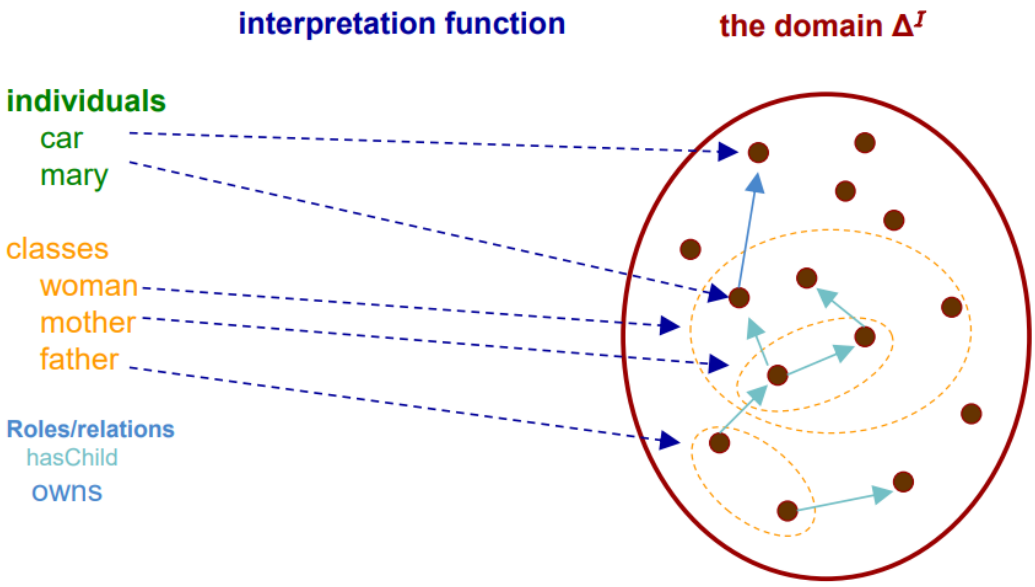

Figure 7: DL semantics

2.5 How to Build an Ontology

Building an ontology involves several steps, each addressing specific aspects of knowledge representation and organization. This process is iterative and is not strictly linear.

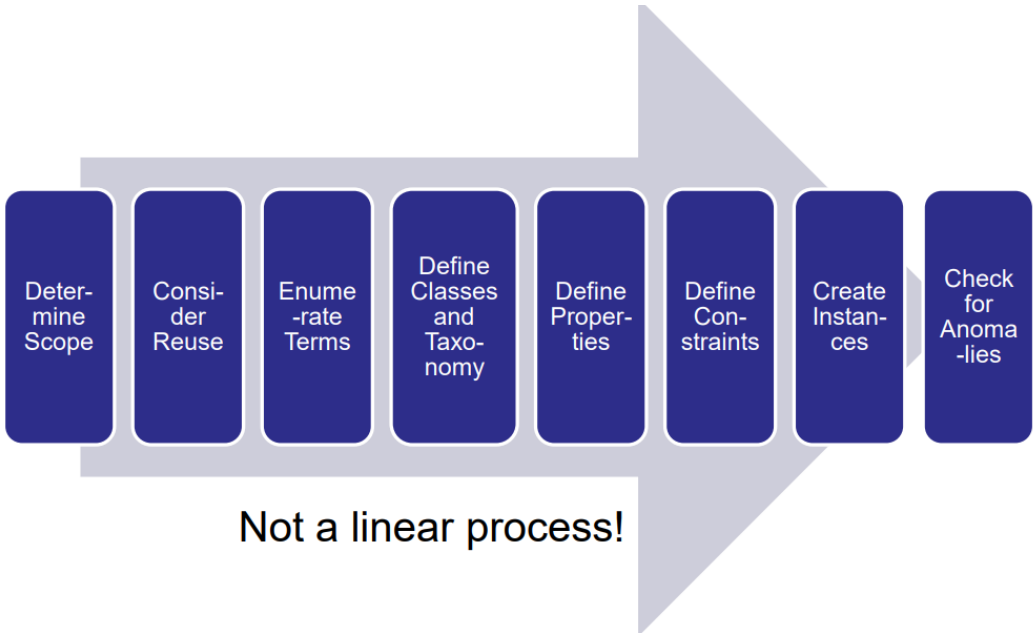

Figure 8: Ontology engineering process

2.5.1 1 - Determine Scope (Competency Questions)

The scope of the ontology should be determined by its intended use and anticipated extensions. Key questions to address include:

- What is the domain that the ontology will cover?
- For what purposes will the ontology be used?
- What types of questions should the ontology answer? (Competency Questions)
- Who will use and maintain the ontology?

Example Competency Questions for a Wine Recommender System:

- Which wine characteristics should I consider when choosing a wine?
- Is Bordeaux a red or white wine?
- Does Cabernet Sauvignon go well with seafood?
- What is the best choice of wine for grilled meat?

2.5.2 2 - Consider Reuse

Reusing existing ontologies saves effort, improves interoperability, and leverages validated ontologies. Sources for reusable ontologies include:

- Protégé Ontology Library (`http://protege.stanford.edu`)
- NCBO Bioportal (`http://bioontology.org`)
- Linked Open Vocabularies (`https://lov.linkeddata.es`)

2.5.3 3 - Enumerate Terms

Write down all relevant terms expected to appear in the ontology. This includes:

- **Nouns**: Basis for class names (e.g., wine, grape, winery).
- **Verbs or Verb Phrases**: Basis for property names (e.g., hasColor, madeFrom).

Examples for a Wine Ontology:

- Terms: wine, grape, winery, region, sugar content.
- Properties: hasColor, locatedIn, madeFrom.

2.5.4 4 - Define Classes and a Taxonomy

Classes represent concepts in the domain and are organized into a taxonomic hierarchy:

- A class is a collection of entities with similar properties (e.g., red wine, winery).
- Subclasses inherit properties from their superclasses.

Example:

- *Red Wine* is a subclass of *Wine*.
- Every instance of *Red Wine* is also an instance of *Wine*.

2.5.5 5 - Define Properties

Properties describe:

- **Attributes** of instances (e.g., color, sugar content).
- **Relationships** between classes (e.g., Wine *hasMaker* Winery).

Properties should:

- Be specified in the highest possible class in the hierarchy.
- Follow domain (parent class) and range (child class) rules.

2.5.6 6 - Define Constraints

Constraints refine the ontology by adding logical rules:

- **Cardinality**: Specifies the number of values a property can have (e.g., each wine has exactly one maker).
- **Existential Restrictions**: Ensures that at least one property exists with a specified value (e.g., wines are located in regions).
- **Universal Restrictions**: Ensures all values of a property are of a certain type (e.g., wines can only be made in wineries).
- **Property Characteristics**: Includes symmetry, transitivity, inverse properties, and functional values.

Examples:

- Wine $\sqsubseteq \geq$1madeFrom.Grape
- Wine $\sqsubseteq \leq$1hasMaker

2.5.7 7 - Create Instances

Instances represent individual entities in the domain:

- The class becomes the direct type of the instance.
- Instances inherit properties and constraints from their class hierarchy.
- Property values assigned to instances must conform to constraints.

Example for a Wine Ontology:

- *Chateau Margaux* is an instance of *Wine*.
- *Chateau Margaux hasMaker Margaux Winery.*

2.5.8 8 - Check Anomalies

Use the formal semantics of the ontology to:

- Validate the model:

  - Check for consistency.
  - Ensure satisfiability of concepts and properties.
- Derive additional knowledge:
  - Automatically classify instances.
  - Infer new relationships or properties.

2.6 Reasoning

Reasoning is one of the key functionalities provided by ontologies due to their formal grounding in logic. It enables the inference of new knowledge based on already declared information. Reasoning is performed using specialized software called reasoners or inference engines.

2.6.1 Capabilities of Reasoning with Ontologies

- **Consistency Checking**: Ensures that the ontology model is free from logical contradictions.
- **Class Inference**: Discovers relationships between classes that were not explicitly declared.
- **Instance Inference**: Determines class membership of instances based on defined relationships and constraints.

2.6.2 Consistency Checking

Consistency checking ensures that the ontology does not contain logical errors:

- Example:
  - Declared: Animal $\sqsubseteq \neg$Plant
  - Declared: Plant(myAlgae)
  - Inferred: Animal(myAlgae)
  - Result: An inconsistency because myAlgae cannot be both an Animal and a Plant.

2.6.3 Satisfiability Checking

Satisfiability checking identifies concepts or classes that cannot possibly have any instances:

- Example:
  - Declared: Algae $\sqsubseteq$ Plant, Animal $\sqsubseteq \neg$Plant, Algae $\sqsubseteq$ Animal
  - Result: The class Algae is unsatisfiable because it cannot belong to both Animal and Plant simultaneously.

2.6.4 Class Inference

Class inference discovers new relationships among classes based on given knowledge:

- Example:
  - Declared: Carnivore $\sqsubseteq$ Animal $\sqcap \forall$eats.Animal
  - Declared: Cat $\sqsubseteq$ Animal $\sqcap \forall$eats.Mouse, Mouse $\sqsubseteq$ Animal
  - Inferred: Cat $\sqsubseteq$ Carnivore

2.6.5 Instance Inference

Instance inference identifies class membership for specific entities:

- Example:
  - Declared: hasPet(Person, Animal), PetOwner $\sqsubseteq$ Person $\sqcap \exists$hasPet.Animal
  - Declared: Person(Pete), Animal(Spike), hasPet(Pete, Spike)
  - Inferred: PetOwner(Pete)

2.6.6 Class + Instance Inference

Combining class and instance inference allows the discovery of both new classes and instance memberships:

- Example:
  - Declared:
    - hasPet(Person, Animal)
    - PetOwner $\sqsubseteq$ Person $\sqcap \exists$hasPet.Animal
    - PetLover $\sqsubseteq$ Person$\sqcap >$ 3hasPet.Animal
  - Declared:
    - Person(Pete), Animal(Tom), Animal(Jerry), Animal(Spike)
    - hasPet(Pete, Spike), hasPet(Pete, Tom), hasPet(Pete, Jerry)
    - $\{Spike\} \sqsubseteq \neg\{Tom, Jerry\}$
  - Inferred:
    - PetOwner(Pete)
    - PetLover(Pete)

Reasoning enables powerful, logic-based conclusions to be drawn from the ontology, enriching both its usability and reliability.

3 Semantic Web, RDF, RDFS, and OWL

3.1 The Semantic Web

The Semantic Web represents an extension of the traditional web, aiming to enable machines to understand and process data in a way that is closer to human reasoning. It focuses on the meaningful interconnection of data to support better knowledge representation, information retrieval, and decision-making.

3.1.1 Key Concepts and Foundations

- **Linked Data:** The Semantic Web is built on the principle of Linked Data, which connects data across different resources using structured formats and qualified links.
  - Links between *things/entities* rather than just documents.
  - Knowledge is *machine-readable*, enabling automated reasoning.
  - Links are qualified to define relationships (e.g., `dbo:city` or `owl:sameAs`).
- **Comparison with the Web of Documents:**
  - The Web of Documents primarily links human-readable pages accessed via browsers.
  - The Web of Data (Semantic Web) links machine-readable entities accessed by programs and agents.
- **Scientific Perspectives:**
  - As emphasized by Sir Tim Berners-Lee: "The more things you have to connect together, the more powerful it is."
  - Decentralized databases underpin this ecosystem, supporting robust knowledge sharing and reasoning.

3.1.2 Semantic Web Language Design Principles

The design of the Semantic Web is governed by specific principles that enable flexibility and interoperability:

AAA Slogan: On the Semantic Web, "Anyone can say Anything about Any topic." This principle highlights the openness of the web, allowing diverse sources to describe and link entities without centralized control.

Non-unique Naming Assumption:

- The same entity may have multiple identifiers (URIs), reflecting its description by different sources.
- Explicit relationships such as `owl:sameAs` are necessary to state that two URIs refer to the same entity.
- Disjointness must also be specified explicitly when entities are distinct.

Open World Assumption (OWA):

- Missing information is not treated as negative information.
- The Semantic Web assumes incomplete knowledge, enabling reasoning over partial data.
- For example:
  - If `likes(PersonA, DrinkB)` is asserted, the OWA does not infer `not likes(PersonA, DrinkC)`.
  - Instead, it concludes `don't know` whether `PersonA` likes `DrinkC`.
- This contrasts with the Closed World Assumption (CWA), where missing information is treated as false.

3.1.3 Benefits of Semantic Web Principles

- Enables a flexible and decentralized approach to knowledge representation.
- Facilitates reasoning and integration of data from disparate sources.
- Supports robust applications in fields like knowledge graphs, data interoperability, and intelligent agents.

3.1.4 Technological Foundations

The Semantic Web leverages existing web technologies to achieve its goals. They are the bottom layer of the Semantic Web Technology stack, which can be seen in 9.

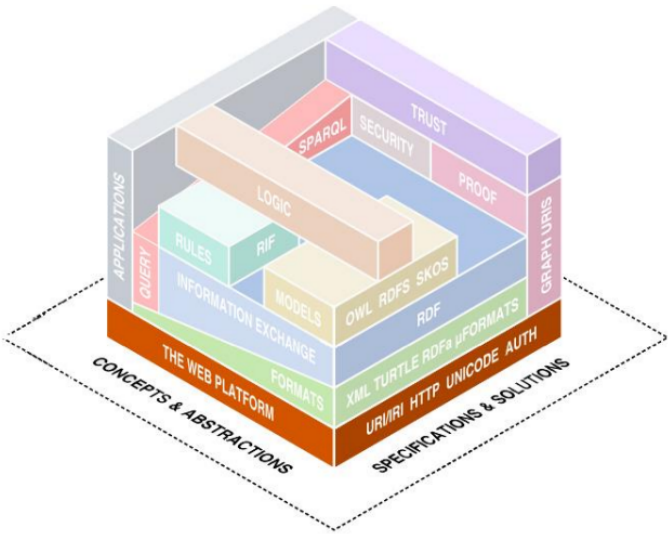

Figure 9: Semantic Web Stack

The technologies used are:

- **TCP/IP:** Facilitates data transfer across networks.
- **HTTP:** Enables client-server communication using a request/response model.

- **HTML and XML:**
  - HTML structures and displays web content.
  - XML provides a flexible markup framework for defining custom data formats.
- **URIs and IRIs:** Identifiers for resources on the web.
  - **URI Types:**
    - Uniform Resource Name (URN): Persistent, location-agnostic identifiers.
    - Uniform Resource Locator (URL): Specifies access mechanisms for resources.
  - **IRI:** An internationalized extension of URI supporting Unicode characters.

3.1.5 Applications and Use Cases

- **Killer Applications:**
  - Intelligent agents retrieving, analyzing, and negotiating information.
  - Scheduling and coordination for multiple stakeholders (e.g., treatments or services).
- **Ontology and Knowledge Representation:**
  - Provides structured models to represent domains of knowledge.
  - Enhances the interoperability and reasoning capabilities of intelligent systems.

3.1.6 Challenges and Evolution

- The Semantic Web is still evolving, requiring:
  - Better adoption of Linked Data principles.
  - Tools for creating, managing, and consuming semantic data.
- Continues to build on the foundational technologies of the traditional web.

3.2 RDF

The Resource Description Framework (RDF) is a W3C standard designed to describe resources on the web in a structured and machine-readable way. It is foundational to the Semantic Web and provides a framework for knowledge representation and sharing across diverse systems.

3.2.1 Overview of RDF

RDF was developed in the late 1990s, with its final W3C recommendation (RDF 1.1) released in 2004. It uses a graph-based data model that formalizes semantics, making it accessible to machines. RDF expresses knowledge as a collection of statements, where each statement is represented as an RDF triple.

An RDF triple consists of:

- **Subject:** The resource being described, identified by a URI.

- **Predicate:** The relationship or property linking the subject to the object, also identified by a URI.
- **Object:** The value or related resource, which can be a URI or a literal.

3.2.2 RDF Graph Model

RDF statements naturally form a directed labeled graph, where:

- Nodes represent resources (subjects or objects).
- Arcs represent predicates, connecting subjects to objects.

For instance, the RDF triple:

```
<http://example.org/person/John>
<http://example.org/property/worksIn>
<http://example.org/resource/ProjectX>
```

is visualized as a graph:

$$\text{John} \xrightarrow{\text{worksIn}} \text{ProjectX}.$$

The object of one statement can act as the subject of another, enabling the integration of multiple graphs.

3.2.3 RDF Serialization Formats

RDF provides multiple serialization formats for different use cases, including machine communication, human readability, and ease of implementation. Common formats include RDF/XML, N-Triples, Turtle, and JSON-LD.

RDF/XML RDF/XML is one of the earliest serialization formats, suitable for inter-machine communication. It uses XML to describe RDF statements. An example:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://example.org/person/John">
    <worksIn rdf:resource="http://example.org/resource/ProjectX" />
  </rdf:Description>
</rdf:RDF>
```

N-Triples N-Triples is a simple, line-based format where each statement is terminated by a period. Example:

```
<http://example.org/person/John>
<http://example.org/property/worksIn>
<http://example.org/resource/ProjectX> .
```

Turtle Turtle (Terse RDF Triple Language) is a human-readable serialization format and a subset of N3. It allows shorthand notations for repeated subjects or predicates. Example:

```
@prefix ex: <http://example.org/> .

ex:John
    ex:worksIn ex:ProjectX ;
    ex:hasSkill "Programming" .
```

Turtle also supports data types and language tags for literals. Example:

```
ex:John ex:age "27"^^xsd:integer ;
         ex:name "John"@en .
```

3.2.4 RDF Literals and Datatypes

RDF supports two types of literals:

- **Plain Literals:** Text values with an optional language tag, e.g., `"Hello World"@en`.
- **Typed Literals:** Text values paired with a datatype URI, e.g., `"27"8sd:integer`.

The datatypes are based on XML Schema, defining value spaces, lexical spaces, and mappings between them.

3.2.5 RDF in Practice

RDF facilitates the integration of knowledge by enabling the merging of multiple graphs. For example, consider two datasets:

- Dataset A states `John worksIn ProjectX`.
- Dataset B states `ProjectX isIn NewYork`.

By combining these graphs, we infer that `John worksIn NewYork`, demonstrating RDF's power to create a web of interconnected knowledge.

3.2.6 Conclusion

RDF is the backbone of the Semantic Web, providing a standard way to describe and link data. Its flexible model and variety of serialization formats ensure its adaptability for various applications, from knowledge graphs to linked data publishing.

3.3 Advanced RDF

Advanced RDF concepts extend the basic principles of RDF to accommodate more complex data structures, unnamed resources, and metadata about statements. These features support nuanced data modeling, enhance flexibility, and address specific challenges in RDF knowledge representation.

3.3.1 Structured Values

RDF enables the representation of structured values, such as addresses, by describing the "aggregated thing" (e.g., an address) as a separate resource. Statements about this new resource capture its individual components, such as street, city, state, and postal code.

Example in Triples Notation:

```
exstaff:85740 exterms:address exaddressid:85740 .
exaddressid:85740 exterms:street "Favoritenstrasse 9-11/188" .
exaddressid:85740 exterms:city "Vienna" .
exaddressid:85740 exterms:state "Vienna" .
exaddressid:85740 exterms:postalCode "1040" .
```

Example in Turtle:

```
exstaff:85740 exterms:address exaddressid:85740 .
exaddressid:85740 exterms:street "Favoritenstrasse 9-11/188" ;
              exterms:city "Vienna" ;
              exterms:state "Vienna" ;
              exterms:postalCode "1040" .
```

3.3.2 Blank Nodes

Blank nodes are unnamed resources represented in RDF. They are useful for describing resources without assigning URIs, grouping related information, or representing n-ary relationships. However, they pose challenges for reasoning and are discouraged in Linked Data, though they remain widely used.

Example Using Blank Nodes:

```
exstaff:85740 exterms:address _:johnaddress .
_:johnaddress exterms:street "1501 Grant Avenue" .
_:johnaddress exterms:city "Bedford" .
_:johnaddress exterms:state "Massachusetts" .
_:johnaddress exterms:postalCode "01730" .
```

3.3.3 Reification

Reification allows making statements about other statements, such as metadata, provenance, or trust. However, it introduces complexity and potential for infinite recursion or cycles.

Standard Reification Example:

```
:Max :says _:s1 .
_:s1 rdf:type rdf:Statement .
_:s1 rdf:subject :Max .
_:s1 rdf:predicate :likes .
_:s1 rdf:object :StarTrek .
```

Alternative approaches to reification include singleton properties, named graphs, and RDF-star (RDF*). Each method has trade-offs in terms of space efficiency, query performance, and system support.

3.3.4 RDF Data Structures: Containers and Collections

RDF supports two primary data structures:

- **Containers:** Open lists that allow the addition of new entries. Examples include `rdf:Bag`, `rdf:Seq`, and `rdf:Alt`.
- **Collections:** Closed lists where entries are fixed and cannot be extended.

Example of an RDF Collection in Turtle:

```
ex:fruits rdf:type rdf:List ;
          rdf:first "Apple" ;
          rdf:rest ( "Banana" "Cherry" rdf:nil ) .
```

3.3.5 RDF Vocabulary

The RDF vocabulary includes predefined classes and properties:

- **Classes:** `rdf:XMLLiteral`, `rdf:Property`, `rdf:Statement`, `rdf:Alt`, `rdf:Bag`, `rdf:Seq`, `rdf:List`.
- **Properties:** `rdf:type`, `rdf:first`, `rdf:rest`, `rdf:value`, `rdf:subject`, `rdf:predicate`, `rdf:object`.

3.3.6 RDF-star and RDF*

RDF-star introduces nested triples as first-class citizens, simplifying the representation of statements about statements. For example:

```
:Alice :says <<:Bob :likes :StarTrek>> .
```

This approach is space-efficient and expressive but is not yet standardized across all triple stores.

3.3.7 Conclusion

Advanced RDF extends the basic RDF model to handle structured data, metadata, and complex relationships. By providing flexible tools for modeling nuanced scenarios, RDF supports a wide range of applications in the Semantic Web and Linked Data ecosystems.

3.4 RDFS (RDF Schema)

RDF Schema (RDFS) is an extension of RDF that introduces a vocabulary for defining the structure and relationships of RDF data. It provides basic semantic constructs to enable inference and organize RDF data into hierarchies, thus enhancing its expressivity.

3.4.1 Purpose of RDFS

RDFS serves two main purposes:

1. **Nomination:**
    - Defines the "types" (i.e., classes) of things about which we make assertions.

- Specifies the properties that can act as predicates in these assertions to capture relationships.

2. **Inference:**
   - Allows reasoning over data to infer additional knowledge that is implicit based on the provided assertions.

3.4.2 Main Constructs of RDFS

RDFS introduces several key constructs to define classes, properties, and their relationships.

Classes:

- `rdfs:Resource`: The superclass of all resources in RDF.
- `rdfs:Class`: Declares a resource as a class (type or category) for other resources.
- `rdfs:Literal`: Represents literal values such as strings and numbers.
- `rdfs:Datatype`: A class of datatypes (e.g., `xsd:string`, `xsd:integer`), which is both an instance of and a subclass of `rdfs:Class`.

Properties:

- `rdfs:subClassOf`: Defines class hierarchies where properties of a superclass are inherited by its subclasses.
- `rdfs:subPropertyOf`: Defines property hierarchies where related resources by a subproperty are also related by its superproperty.
- `rdfs:domain`: Restricts the scope of a property to specific classes.
- `rdfs:range`: Restricts the range of values a property can have.
- `rdfs:comment`: Provides a human-readable description of a resource.
- `rdfs:seeAlso`: Indicates resources that provide additional information.
- `rdfs:isDefinedBy`: Points to a resource that defines the subject.

3.4.3 Example of Class Hierarchies

The following example demonstrates the use of `rdfs:subClassOf` to define hierarchies:

RDFS Example in Turtle:

```
@prefix ex: <http://example.org/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Person a rdfs:Class .
ex:Author rdfs:subClassOf ex:Person .
ex:Reader rdfs:subClassOf ex:Person .
```

```
ex:Frank a ex:Author .
ex:Lynda a ex:Reader .
ex:Frank ex:communicatesTo ex:Lynda .
```

From this data, an RDFS reasoner can infer:

- `ex:Frank` is an instance of `ex:Person`.
- `ex:Lynda` is an instance of `ex:Person`.

3.4.4 Domain and Range Restrictions

The `rdfs:domain` and `rdfs:range` properties restrict the applicability and type of properties.

Example in Turtle:

```
@prefix mo: <http://purl.org/ontology/mo/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

mo:member rdfs:domain mo:MusicGroup .
mo:member rdfs:range foaf:Agent .

mo:TheBeatles mo:member foaf:PaulMcCartney .
```

From this data, the following can be inferred:

- `mo:TheBeatles` is an instance of `mo:MusicGroup`.
- `foaf:PaulMcCartney` is an instance of `foaf:Agent`.

3.4.5 RDFS Vocabulary

The RDFS vocabulary expands RDF with the following classes and properties:

- **Classes:**
  - `rdfs:Resource`, `rdfs:Class`, `rdfs:Literal`, `rdfs:Datatype`, `rdfs:Container`, `rdfs:List`
- **Properties:**
  - `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`, `rdfs:comment`, `rdfs:seeAlso`, `rdfs:isDefinedBy`.

3.4.6 Limitations of RDFS

Despite its utility, RDFS has significant limitations:

- It supports only basic inference and lacks the expressivity for complex reasoning.
- It does not allow for logical expressions beyond membership and basic hierarchies.

3.5 OWL (Web Ontology Language)

The Web Ontology Language (OWL) is a W3C recommendation designed for representing ontologies on the Semantic Web. OWL extends RDF and RDFS, providing a fully-fledged knowledge representation language that incorporates logical constructs and formal semantics. It enables more expressive and complex descriptions of knowledge domains while supporting inferencing to derive implicit knowledge.

3.5.1 Overview and Purpose

OWL addresses the limitations of RDFS by introducing a richer set of constructs and logical operators. Its formal foundation lies in Description Logics (DL), which provide:

- Well-defined semantics for precise reasoning.
- Mechanisms to verify the consistency of knowledge bases.
- Tools to infer implicit knowledge from explicitly defined statements.
- Computationally manageable reasoning processes.

OWL ontologies are typically expressed in RDF syntax, enabling easy integration and sharing on the Semantic Web.

3.5.2 Extending RDFS

OWL extends RDFS in several significant ways, including:

- Logical expressions such as `AND`, `OR`, `NOT`.
- Equality and inequality of classes and individuals.
- Local properties with domain- and range-specific constraints.
- Cardinality constraints, such as specifying the number of relationships.
- Symmetric, transitive, and inverse properties.
- Enumerated classes and required/optional properties.

For example, the following OWL statements specify the symmetry of a property:

```
:hasSibling a owl:ObjectProperty ;
            owl:SymmetricProperty .
```

3.5.3 Classes and Individuals

Classes in OWL define sets of individuals and are referred to as "types," "concepts," or "categories." Membership in a class is determined by logical descriptions rather than names. OWL defines two special classes:

- `owl:Thing`: The universal class that includes all individuals.
- `owl:Nothing`: The empty class containing no individuals.

Defining Classes and Individuals:

```
:Book a owl:Class .
:NineteenEightyFour a :Book .
:FrankSmith a owl:NamedIndividual .
```

3.5.4 Properties

OWL properties are binary relations between individuals or between individuals and data values. They come in two main types:

- **Object Properties:** Relate individuals to other individuals.
- **Datatype Properties:** Relate individuals to data values.

Examples of Object and Datatype Properties:

```
:author a owl:ObjectProperty ;
        rdfs:domain :Book ;
        rdfs:range :Writer .

:publicationYear a owl:DatatypeProperty ;
                 rdfs:domain :Book ;
                 rdfs:range xsd:integer .
```

Object properties can have characteristics such as transitivity, symmetry, and inverses:

```
:hasParent a owl:ObjectProperty ;
           owl:TransitiveProperty .
:hasBrother owl:SymmetricProperty .
:hasSpouse owl:inverseOf :isSpouseOf .
```

3.5.5 Class Hierarchies and Inference

OWL allows the creation of class hierarchies through subclass relationships, supporting reasoning to infer implicit relationships. For example:

```
:Novel a owl:Class ;
       rdfs:subClassOf :Book .

:Book a owl:Class ;
       rdfs:subClassOf :Work .

:Work a owl:Class .
```

From the above definitions, an OWL reasoner can infer that:

- `:Novel` is a subclass of `:Work`.
- Any individual of `:Novel` is also an individual of `:Work`.

OWL also supports the declaration of disjoint classes and equivalence:

```
:Book owl:disjointWith :Writer .
:Writer owl:equivalentClass :Author .
```

3.5.6 Advanced Constructs

OWL supports advanced constructs such as:

- **Cardinality:** Specifies the number of relationships an individual can have. For example:

```
:Person a owl:Class ;
        owl:equivalentClass [
            a owl:Restriction ;
            owl:onProperty :hasChild ;
            owl:minQualifiedCardinality "1"^^xsd:nonNegativeInteger ;
            owl:onClass :Child
        ] .
```

- **Enumerated Classes:** Defines classes by enumerating their members:

```
:PrimaryColors a owl:Class ;
               owl:oneOf (:Red :Blue :Yellow) .
```

- **Property Restrictions:** Enforces constraints on property values:

```
:Person a owl:Class ;
        owl:equivalentClass [
            a owl:Restriction ;
            owl:onProperty :hasAge ;
            owl:allValuesFrom xsd:integer
        ] .
```

3.5.7 Combining TBox and ABox

OWL integrates the TBox (terminology, or schema) and ABox (assertions, or instance data) into a unified framework. For instance:

```
:Book a owl:Class .
:Writer a owl:Class .
:NineteenEightyFour a :Book ;
                    :author :GeorgeOrwell ;
                    :publicationYear 1948 .
```

3.5.8 Inference and Reasoning

OWL reasoners leverage Description Logics to infer new knowledge from existing assertions. For example:

- Given `:Poet rdfs:subClassOf :Writer` and `:Writer owl:equivalentClass :Author`, a reasoner can infer `:Poet rdfs:subClassOf :Author`.

- Individuals can be related or distinguished:

```
:NineteenEightyFour owl:sameAs :ARX012345 .
:ARX012345 owl:differentFrom :ARX0123456 .
```

3.6 Advanced OWL

Advanced OWL concepts extend the foundational constructs to provide a more expressive framework for representing complex relationships, restrictions, and logical constructs in ontologies. These features enable nuanced reasoning and interoperability for knowledge representation on the Semantic Web.

3.6.1 Disjunctive Properties

Disjunctive properties ensure that two individuals cannot be related through both properties simultaneously. OWL provides constructs for declaring disjoint properties.

Example of Disjoint Properties:

```
:hasParent a owl:ObjectProperty ;
           owl:propertyDisjointWith :hasChild .
```

For multiple disjunctive properties, OWL provides a shortcut:

```
[] rdf:type owl:AllDisjointProperties ;
   owl:members (:hasParent :hasChild :hasGrandchild) .
```

3.6.2 Transitive Properties

A transitive property enables reasoning over chains of relationships. If a relationship exists between `A` and `B`, and between `B` and `C`, it can be inferred that the relationship exists between `A` and `C`.

Example of a Transitive Property:

```
:isPublishedBefore a owl:ObjectProperty ;
                   owl:TransitiveProperty ;
                   rdfs:domain owl:Book ;
                   rdfs:range owl:Book .

:AnimalFarm a owl:Book ;
            :isPublishedBefore :NineteenEightyFour .

:BraveNewWorld a owl:Book ;
               :isPublishedBefore :AnimalFarm .
```

From this, an OWL reasoner can infer:

```
:BraveNewWorld :isPublishedBefore :NineteenEightyFour .
```

3.6.3 Closed Classes (Nominals)

OWL allows the definition of closed classes using the `owl:oneOf` construct. This restricts membership in the class to explicitly enumerated individuals.

Example of Closed Classes:

```
:Novel a owl:Class ;
       owl:oneOf (:AnimalFarm :NineteenEightyFour) .

:AnimalFarm a :Novel .
:NineteenEightyFour a :Novel .
```

This states that only `:AnimalFarm` and `:NineteenEightyFour` are members of the `:Novel` class.

3.6.4 Property Restrictions

OWL supports property restrictions to define complex classes based on relationships. Restrictions include:

- `owl:hasValue`: Restricts a property to a fixed value.
- `owl:allValuesFrom`: Specifies that all values of a property must belong to a specific class (universal quantification).
- `owl:someValuesFrom`: Specifies that at least one value of a property must belong to a specific class (existential quantification).
- `owl:cardinality`, `owl:minCardinality`, `owl:maxCardinality`: Restrict the number of relationships a property can have.

Example of Universal Restriction:

```
:VegetarianPizza a owl:Class ;
                 rdfs:subClassOf [
                     a owl:Restriction ;
                     owl:onProperty :hasTopping ;
                     owl:allValuesFrom :VegetarianTopping
                 ] .
```

Example of Existential Restriction:

```
:Reader a owl:Class ;
        rdfs:subClassOf [
            a owl:Restriction ;
            owl:onProperty :reads ;
            owl:someValuesFrom :Book
        ] .
```

Example of Cardinality Restriction:

```
:Tetralogy a owl:Class ;
           rdfs:subClassOf [
               a owl:Restriction ;
               owl:onProperty :hasVolumes ;
               owl:cardinality 4
           ] .
```

3.6.5 Property Characteristics and Relationships

OWL provides constructs for defining advanced property characteristics:

- `owl:SymmetricProperty`: A property is symmetric if the relationship is bidirectional (e.g., `isSiblingOf`).
- `owl:TransitiveProperty`: A property is transitive if it holds over chains of relationships (e.g., `isPartOf`).
- `owl:FunctionalProperty`: A property is functional if an individual can have at most one value for it (e.g., `hasBirthDate`).
- `owl:InverseFunctionalProperty`: A property is inverse functional if its inverse is functional (e.g., `isBirthMotherOf`).
- `owl:AsymmetricProperty`: A property is asymmetric if the inverse relationship cannot hold (e.g., `isLeftOf`).
- `owl:ReflexiveProperty`: A property is reflexive if it always relates an individual to itself (e.g., `isEqualTo`).
- `owl:IrreflexiveProperty`: A property is irreflexive if no individual can relate to itself (e.g., `isParentOf`).

Example of a Symmetric Property:

```
:isSiblingOf a owl:ObjectProperty ;
             owl:SymmetricProperty .
```

Example of an Inverse Property:

```
:hasSpouse a owl:ObjectProperty ;
           owl:inverseOf :isSpouseOf .
```

3.6.6 Combining Restrictions

OWL enables the combination of restrictions to define complex classes. For example:

```
:VegetarianPizza a owl:Class ;
                 rdfs:subClassOf [
                     a owl:Restriction ;
                     owl:onProperty :hasTopping ;
```

```
        owl:someValuesFrom :VegetarianTopping
    ] ;
    rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty :hasTopping ;
        owl:allValuesFrom :VegetarianTopping
    ] .
```

This defines `:VegetarianPizza` as a class where all toppings are vegetarian and at least one topping is specified.

4 SPARQL

5 SHACL

6 Knowledge Graph Creation

6.1 Introduction

6.2 Tabular Data to RDF

6.3 Relational Data to RDF

6.4 Structured Data to RDF

6.5 Text to RDF

6.6 RDF Storage

7 Extras