Unit-3

ONTOLOGY ENGINEERING:

Ontology is the formal specification of terms within a domain and their relationships. It defines a common vocabulary for the sharing of information that can be used by both humans and computers. Ontologies can be in the form of lists of words; taxonomies, database schema, frame languages and logics. The main difference between these forms is their expressive power.

Ontology together with a set of concept instances constitutes a knowledge base. If a program is designed to compare conceptual information across two knowledge bases on the Web, it must know when any two terms are being used to mean the same thing. Ideally, the program must have a way to discover common meanings for whatever knowledge bases it encounters. Typically, an ontology on the Web will combine a taxonomy with a set of inference rules.

Taxonomy is defined as a set of classes of objects and their relationships. These classes, subclasses, and their relationships are important tools for manipulating information. Their relations are described by assigning properties to classes and allowing subclasses to inherit these properties. An ontology then is a taxonomy plus inference.

Ontology inference rules allow manipulation of conceptual information. The most important ontology relationship is the subsumption link (e.g., subtype and supertype link).

When a network of concepts is represented by a tree, it rigorously defines the taxonomy. While ontology can sometimes be modularized as a set of trees, some advocate that all ontology should be taxonomic, but others favor a lattice structure.

Ontology engineering seeks a common vocabulary through a data collection process that includes discussions, interviews, document analysis, and questionnaires.

Existing ontologies on a subject are discovered, assessed, and reused as much as possible to avoid ―reinventing the wheel.‖ As part of this process, ontologies are designed as living objects with a maintenance cycle.

Ontology Applications:

The simplest ontology consists of a simple taxonomy with a single relation. Categories of ontology applications can be grouped as

- **Neutral Authoring:** The author of an object in a single language translates into a different format for use in alternative applications.
- **Ontology as Specification:** Ontology of a given domain is created and used as a basis for specification and development of some software. This approach allows documentation, maintenance, reliability and knowledge (re)use.
- **Common Access to Information:** Information in an inaccessible format becomes intelligible by providing a shared understanding of the terms, or by mapping between sets of terms.
- **Ontology-Based Search:** Ontology is used for searching an information repository.

CONSTRUCTING ONTOLOGY:

Ontology permits sharing common understanding of the structure of information among people and software agents. Since there is no unique model for a particular domain, ontology development is best achieved through an iterative process. Objects and their relationships reflect the basic concepts within an ontology.

An iterative approach for building ontologies starts with a rough first pass through the main processes as follows:

- **First,** set the scope. The development of an ontology should start by defining its domain and scope.

Several basic questions are helpful at this point:
What will the ontology cover?
How will the ontology be used?
What questions does the ontology
answer? Who will use and maintain the
ontology?
The answers may change as we proceed, but they help limit the scope of the model.

- **Second**, evaluate reuse. Check to see if existing ontologies can be refined and extended. Reusing existing ontologies will help to interact with other applications and vocabularies. Many knowledge-representation systems can import and export ontologies directly for reuse.

- **Third,** enumerate terms. It is useful to list all terms, what they address, & what properties they have. Initially, a comprehensive list of terms is useful without regard for overlapping concepts. Nouns can form the basis for class names & verbs can form the basis for property names.

- **Fourth**, define the taxonomy. There are several possible approaches in developing a class hierarchy: a top-down process starts by defining general concepts in the domain. A bottom-up development process starts with the definition of the most specific classes, the levels of the hierarchy, with subsequent grouping of these classes into more general concepts.

- **Fifth,** define properties. The classes alone will not provide enough information to answer questions. We must also describe the internal structure of concepts. While attaching properties to classes one should establish the domain and range. Property constraints (facets) describe or limit the set of possible values for a frame slot.
- **Sixth,** define facets. Up to this point the ontology resembles a RDFS without any primitives from OWL. In this step, the properties add cardinality, values, and characteristics that will enrich their definitions.
- **Seventh,** the slots can have different facets describing the value type, allowed values, the number of the values (cardinality), and other features of the values.

Slot cardinality: the number of values a slot has. Slot value type: the type of values a slot has.
Minimum and maximum value: a range of values for a numeric slot.
Default value: the value a slot has unless explicitly specified otherwise.

- **Eighth,** define instances. The next step is to create individual instances of classes in the hierarchy.

- **Finally,** check for anomalies. The Web-Ontology Language allows the possibility of

detecting inconsistencies within the ontology. Anomalies, such as incompatible domain and range definitions for transitive, symmetric, or inverse properties may occur.

ONTOLOGY DEVELOPMENT TOOLS:

Below is a list of some of the most common editors used for building ontologies:

- **DAG-Edit** provides an interface to browse, query and edit vocabularies with a DAG data structure: **http://www.geneontology.org/#dagedit**
- **Protege 2000** is the most widely used tool for creating ontologies and knowledge bases: **http://protege.stanford.edu/index.shtml**
- **WonderTools** is an index for selecting an ontology-building tool: **http://www.swi.psy.uva.nl/wondertools/**
- **WebOnto** is a Java applet coupled with a Web server that allows users to browse and edit knowledge models: **http://kmi.open.ac.uk/projects/webonto/**

ONTOLOGY "SPOT" EXAMPLE

Portions of the following example for the ―spot‖ ontology were taken from http://www.charlestoncore.org/ont/example/index.html.

The spot ontology consists of ***three owl:Classes (spot, ellipse, and point)*** and ***six rdf:Properties (shape, center, x-position, y-position, x-radius, y-radius).*** Together, these vocabularies can be used to describe a spot.

Classes

The three OWL classes are Spot: A two dimensional (2D) ―spot‖ defined as a closed region on the plane.

Example of SPOT Ontology

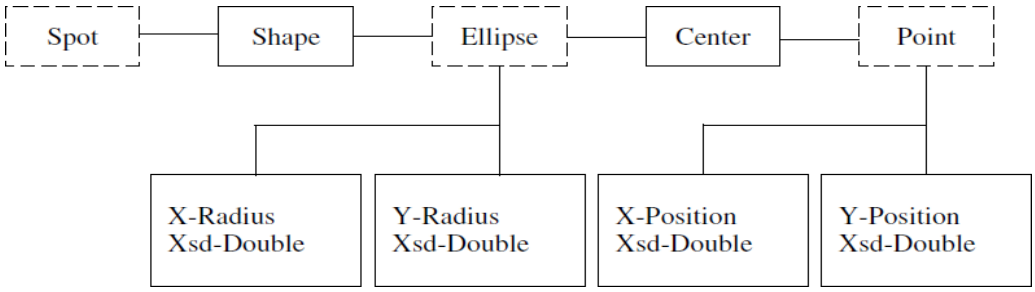


Fig: Example Ontology

Point: A point is defined as a location on a Cartesian plane. It has two attributes; its *x*-position and *y*-position on an implicit coordinate system of the plane.

Ellipse: Ellipse here is defined as a circle stretched along either the *x*- or *y*-axis of a coordinate system. The major and minor axes of an Ellipse parallel the coordinates of the implicit coordinate system.

Properties

The six RDF properties are Shape: A Spot assumes a shape of an Ellipse. Therefore the domain of shape is Spot and the range of Spot is Ellipse.

Center: The center is the center point of the Ellipse. It has a ***rdfs:domain*** of Ellipse and a ***rdfs:range*** of Point.

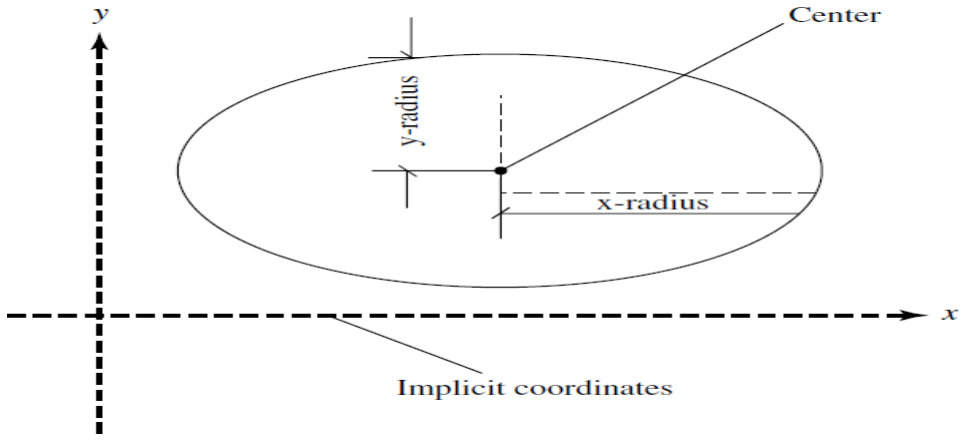


Fig: Ellipse definition

***x*-Position:** An *x*-position is an owl:Datatype property that has a domain of Point. Its value (of type xsd:double) is the distance from the origin on the *x*-axis of the coordinate system.

***y*-Position:** A *y*-position is a owl:Datatype property that has a domain of Point. Its value (of type xsd:double) is the distance from the origin on the *y*-axis of the coordinate system.

***x*-Radius:** *x*-radius is a owl:Datatype property that has a rdfs:domain of Ellipse. It is the radius parallel to the *x*-axis of the coordinate system.

***y*-Radius:** A *y*-radius is a owl:Datatype property that has a rdfs:domain of Ellipse. It is the radius parallel to the *y*-axis of the coordinate system.

The OWL file for this ontology example ***(http://www.charlestoncore.org/ont/example/index.html) is as follows:***

<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE rdf:RDF (...)>
<rdf:RDF xmlns="http:// example#"
xmlns:example="http:// example#" xmlns:rdf=
"http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xml:base="http:// /example">
<owl:Ontology rdf:about="">
<rdfs:isDefinedBy rdf:resource="**http:// example/**" />
<dc:author>**Smith**</dc:author>
<dc:title>**Example Ontology**</dc:title>
<rdfs:comment>**This file defines a partial ontology in OWL**</rdfs:comment>
<owl:versionInfo>**2005**</owl:versionInfo>
</owl:Ontology>
<owl:Class rdf:ID="**Spot**" />
<owl:Class rdf:ID="**Ellipse**" />

```
<owl:Class rdf:ID="Point" />
<owl:ObjectProperty rdf:ID="shape">
<rdfs:domain rdf:resource="#Spot" />
<rdfs:range rdf:resource="#Ellipse" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="center">
<rdfs:domain rdf:resource="#Ellipse" />
<rdfs:range rdf:resource="#Point" />
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="x-radius">
<rdfs:domain rdf:resource="#Ellipse" />
<rdfs:range rdf:resource= "http://www.w3.org/2001/XMLSchema#double"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="y-radius">
<rdfs:domain rdf:resource="#Ellipse" />

<rdfs:range rdf:resource= "http://www.w3.org/2001/XMLSchema#double"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="x-position">
<rdfs:domain rdf:resource="#Point" />
<rdfs:range rdf:resource= "http://www.w3.org/2001/XMLSchema#double"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="y-position">
<rdfs:domain rdf:resource="#Point" />
<rdfs:range rdf:resource= "http://www.w3.org/2001/XMLSchema#double"/>
</owl:DatatypeProperty>
</rdf:RDF>
```

ONTOLOGY METHODS

Several approaches for developing ontologies have been attempted in the last two decades. In 1990, Lenat and Guha proposed the general process steps. In 1995, the first guidelines were proposed on the basis of the Enterprise Ontology and the TOVE (TOronto Virtual Enterprise) project. A few years later, the On-To-Knowledge methodology was developed.

The Cyc Knowledge Base (see http://www.cyc.com/) was designed to accommodate all of human knowledge and contains about 100,000 concept types used in the rules and facts encoded in its knowledge base. ***The method used to build the Cyc consisted of three phases.***

The **first phase** manually codified articles and pieces of knowledge containing common sense knowledge implicit in different sources.

The **second and third phase** consisted of acquiring new common sense knowledge using natural language or machine learning tools.

The **Electronic Dictionary Research (ERD) project** in Japan has developed a dictionary with over 400,000 concepts, with their mappings to both English and Japanese words. Although the EDR project has many more concepts than Cyc, it does not provide as much detail for each one .

WordNet is a hierarchy of 166,000 word form and sense pairs. WordNet does not have as much detail as Cyc or as broad coverage as EDR, but it is the most widely used ontology for natural language processing, largely because it has long been easily accessible over the Internet (see http://www.cogsci.princeton.edu/~wn/).

Cyc has the most detailed axioms and definitions; it is an example of an axiomatized or formal ontology. Both EDR and WordNet are usually considered terminological ontologies. The difference between a terminological ontology and a formal ontology is one of degree: as more axioms are added to a terminological ontology, it may evolve into a formal or axiomatized ontology.

The main concepts in the ontology development include: a top-down approach, in which the most abstract concepts are identified first, and then, specialized into more specific concepts; a bottom-up approach, in which the most specific. Concepts are identified first and then generalized into more abstract concepts; and a middle-out approach, in which the most important concepts are identified first and then generalized and specialized into other concepts.

Methontology was created in the Artificial Intelligence Lab from the Technical University of Madrid (UPM). It was designed to build ontologies either from scratch, reusing other ontologies as they are, or by a process of reengineering them. The Methontology framework enables the construction of ontologies at the knowledge level. It includes the identification of the ontology development process, a life cycle based on evolving prototypes, and particular techniques to carry out each activity. The ontology development process identifies which tasks should be performed when building ontologies (scheduling, control, quality assurance, specification, knowledge acquisition, conceptualization, integration, formalization, implementation, evaluation, maintenance, documentation, and configuration management).

The main phase in the ontology development process using the Methontology approach is the conceptualization phase.

By comparison, the On-To-Knowledge methodology includes the identification of goals that should be achieved by knowledge management tools and is based on an analysis of usage scenarios.

The steps proposed by the methodology are **kickoff:** where ontology requirements are captured and specified, competency questions are identified, potentially reusable ontologies are studied, and a first draft version of the ontology is built; **refinement:** where a mature and application oriented ontology is produced; **evaluation:** where the requirements and competency questions are checked, and the ontology is tested in the application environment; and finally ontology maintenance.

ONTOLOGY SHARING AND MERGING

Knowledge representation is the application of logic and ontology to the task of constructing automated models.

Each of the following three fields contributes to knowledge representation:

- **Logic:** Different implementations support different subsets and variations of logic. Sharing information between implementations can usually be done automatically if the information can be expressed a common subset.

- **Ontology:** Different systems may use different names for the same kinds of objects; or they may use the same names for different kinds.

- **Computation:** Even when the names and definitions are identical, computational or implementation side effects may produce different behaviors indifferent systems. In some implementations, the order of entering rules may have inferences that impact computations. Sometimes, the side effects may cause an endless loop.

Although these three aspects of knowledge representation pose different kinds of problems, they are interdependent. Standardizing the terminology used to classify and find the information is important.

For artificial intelligence, where the emphasis is on computer processing, effort has been directed to precise axioms suitable for extended computation and deduction.

ONTOLOGY LIBRARIES

Scientists should be able to access a global, distributed knowledge base of scientific data that appears to be integrated, locally available, and is easy to search.

Data is obtained by multiple instruments, using various protocols in differing vocabularies using assumptions that may be inconsistent, incomplete, evolving, and distributed. Currently, ***there are existing ontology libraries including***

• DAML ontology library (www.daml.org/ontologies).
• Ontolingua ontology library (www.ksl.stanford.edu/software/ontolingua/).
• Prot´eg´e ontology library (protege.stanford.edu/plugins.html).

Available upper ontologies include
• IEEE Standard Upper Ontology (suo.ieee.org).
• Cyc (www.cyc.com).

Available general ontologies include
• (www.dmoz.org).
• WordNet (www.cogsci.princeton.edu/~wn/).
• Domain-specific ontologies.
• UMLS Semantic Net.
• GO (Gene Ontology) (www.geneontology.org).
• Chemical Markup Language, CML.

ONTOLOGY MATCHING

Ontology provides a vocabulary and specification of the meaning of objects that encompasses several conceptual models: including classifications, databases, and axiom theories. However, in the open Semantic Web environment different ontologies may be defined.

Ontology matching finds correspondences between ontology objects. These include ontology merging, query answering, and data translation. Thus, ontology matching enables data interoperate.

Today ontology matching is still largely labor-intensive and error-prone. As a result, manual matching has become a key bottleneck.

String Matching

String matching can help in processing ontology matching. String matching is used in text processing, information retrieval, and pattern matching. There are many string matching methods including ―edit distance‖ for measuring the similarities of two strings.

Let us consider two strings; S1 and S2.

If we use limited steps of character edit operations (insertions, deletions, and substitutions), S1 can be transformed into S2 in an edit sequence. The edit distance defines the weight of an edit sequence.

The existing ontology files on the Web (e.g., http://www.daml.org/ontologies) show that people usually use similar elements to build ontologies, although the complexity and terminology may be different. This is because there are established names and properties to describe a concept. The value of string matching lies in its utility to estimate the lexical similarity.

However, we also need to consider the real meaning of the words and the

context.

In addition, there are some words that are similar in alphabet form while they have different meaning such as, ―too‖ and ―to.‖ Hence, it is not enough to use only string matching.

ONTOLOGY MAPPING

Ontology mapping enables interoperability among different sources in the Semantic Web. It is required for combing distributed and heterogeneous ontologies.

Ontology mapping transforms the source ontology into the target ontology based on semantic relations. There are three mapping approaches for combing distributed and heterogeneous ontologies:

1. Mapping between local ontologies.
2. Mapping between integrated global ontology and local ontologies.
3. Mapping for ontology merging, integration, or alignment.

Ontology merge, integration, and alignment can be considered as ontology reuse processes.

Ontology merge is the process of generating a single, coherent ontology from two or more existing and different ontologies on the same subject.

Ontology integrationis the **process of generating a single ontology from two or more differing ontologies in different subjects.** Ontology alignment creates links between two original ontologies.

ONTOLOGY MAPPING TOOLS

There are three types of ontology mapping tools and provide an example of each:

For ontology mapping between local ontologies, an example mapping tool is **GLUE**. GLUE is a system that semiautomatically creates ontology mapping using machine learning techniques. Given two ontologies, GLUE finds the most similar concept in the other ontology.

For similarity measurement between two concepts, **GLUE calculates the joint probability distribution of the concepts.** The GLUE uses a multistrategy learning approach for finding joint probability distribution.

For ontology mappings between source ontology and integrated global ontology, an example tool is Learning Source Description (LSD). In LSD, Schema can be viewed as ontologies with restricted relationship types. This process can be considered as ontology mapping between information sources and a global ontology.

For ontology mapping in **ontology merging, alignment, and integration, an example tool is OntoMorph.** OntoMorph provides a powerful rule language for specifying mappings, and facilitates ontology merging and the rapid generation of knowledge base translators. **It combines two syntactic rewriting and semantic rewriting.** Syntactic rewriting is done through pattern-directed rewrite rules for sentence-level transformation based on pattern matching. Semantic rewriting is done through semantic models and logical inference.

LOGIC:

Logic is the study of the principles of reasoning. As such, it constructs formal languages for expressing knowledge, semantics, and automatic reasoners to deduce (infer) conclusions.

Logic forms the foundation of Knowledge-Representation (KR), which has been applied to Artificial Intelligence (AI) in general and the World Wide Web in particular. Logic provides a high-level language for expressing knowledge and has high expressive power. In addition, KR has a well-understood formal semantics for assigning unambiguous meaning to logic statements.

Predicate (or first-order) logic, as a mathematical construct, offers a complete proof system with consequences. Predicate logic is formulated as a set of axioms and rules that can be used to derive a complete set of true statements (or proofs).

As a result, with predicate logic we can track proofs to reach their consequences and also logically analyze hypothetical answers or statements of truth to determine their validity.

Proof systems can be used to automatically derive statements syntactically from premises. Given a set of premises, such systems can analyze the logical consequences that arise within the system.

Both RDF and OWL (DL and Lite) incorporate capabilities to express predicate logic that provide a syntax that fits well with Web languages.

RULE:

Inference Rules

In logic, a rule is a scheme for constructing valid inferences. These schemes establish syntactic relations between a set of formulas called premises and an assertion called a conclusion. New true assertions can be reached from already known ones.

There are two forms of deductively valid argument:

1. modus ponens (Latin for ―the affirming mode‖)
2. modus tollens (the denying mode).

For first-order predicate logic, rules of inference are needed to deal with logical quantifiers.

Related proof systems are formed from a set of rules, which can be chained together to form proofs, or derivations. If premises are left unsatisfied in the derivation, then the derivation is a proof of a conditional statement: ―*if* the premises hold, *then* the conclusion holds.‖

Inference rules may also be stated in this form:

(*1*) some premises; (*2*) a turnstile symbol , which means ―infers,‖ ―proves,‖ or ―concludes‖; and (*3*) a conclusion.

The turnstile symbolizes the executive power.

The implication symbol → indicates *potential* inference and it is a logical operator.

For the Semantic Web, logic can be used by software agents to make decisions and select a path of action. For example, a shopping agent may approve a discount for a customer because of the rule:

RepeatCustomer*(*X*)*→ discount*(*25%*)*

where repeat customers are identified from the company database.

This involves rules of the form ―IF (condition), THEN (conclusion).‖ With only a finite number of comparisons, we are required to reach a conclusion.

Axioms of a theory are assertions that are assumed to be true without proof. In terms of semantics, axioms are valid assertions. Axioms are usually regarded as starting points for applying rules of inference and generating a set of conclusions.

Rules of inference, or *transformation rules*, are rules that one can use to infer a conclusion from a premise to create an argument. A set of rules can be used to infer any valid conclusion if it is complete, while never inferring an invalid conclusion, if it is sound.

Rules can be either conditional or biconditional. **Conditional rules**, or *rules of inference*, are rules that one can use to infer the first type of statement from the second, but where the second cannot be inferred from the first. With **biconditional rules**, in contrast, both inference directions are valid.

Conditional Transformation Rules

We will use letters *p*, *q*, *r*, *s*, etc. as propositional variables.

An argument is Modus ponens if it has the following form (P1 refers to the first premise; P2 to the second premise: C to the conclusion):

(P1) if *p* then *q*
(P2) *p*
(C) *q*

Example:

(P1) If Socrates is human then Socrates is mortal.
(P2) Socrates is human.
(C) Socrates is mortal.

Which can be represented as Modus ponens:

[*(p → q)* A *p*] → [*q*]

An argument is Modus tollens if it has the following form:

(P1) if *p* then *q*
(P2) not-*q*
(C) not-*p*

Example:
(P1) If Socrates is human then Socrates is mortal.
(P2) Socrates is not mortal.
(C) Socrates is not human.

In both cases, the order of the premises is immaterial (e.g., in modus tollens ―not-*q*‖ could come first instead of ―if *p* then *q*‖).

Modus tollens $[(p \rightarrow q) \wedge \neg q] \rightarrow [\neg p]$
An argument is a disjunctive syllogism if it has either of the following forms:
(P1) *p* or *q* (P1) *p* or *q*
(P2) not-*p* (P2) not-*q*
(C) *q* (C) *p*

The order of the premises is immaterial (e.g., ―not-*q*‖ could come first instead of ―*p* or *q*‖).

This argument form derives its name from the fact that its major premise is a ―disjunction,‖ that is, a proposition of the form ―*p* or *q*.‖ The propositions *p* and *q* are called the ―disjuncts‖ of the disjunction ―*p* or *q*.‖

In logic, the disjunction ―*p* or *q*‖ is interpreted as the claim that not both *p* and *q* are false; that is, that at least one of them is true. Thus a disjunction is held to be true even when both its disjuncts are true.

Biconditional Transformation Rules
Biconditional rules, or *rules of replacement*, are rules that one can use to infer the first type of statement from the second, or vice versa.

Double negative elimination is represented as

$[\neg \neg p] \leftrightarrow [p]$

Tautology is represented as
$[p] \leftrightarrow [p \vee p]$

MONOTONIC AND NONMONOTONIC RULES
If a conclusion remains valid after new information becomes available within predicate logic, then we refer to this case as a monotonic rule. If, however, the conclusion may become invalid with the introduction of new knowledge, then the case is called a nonmonotonic rule.

Nonmonotonic rules are useful where information is unavailable. These rules can be overridden by contrary evidence presented by other rules. Priorities are helpful to resolve some conflicts between nonmonotonic rules. The XML-based languages can be used to represent rules.

DESCRIPTIVE LOGIC
Descriptive logic is a family of logic based on knowledge-representation formalisms that is a descendant of semantic networks. It can describe the domain in terms of concepts (classes), roles (properties, relationships), and individuals.

Inference and Classes
We can make inferences about relationships between classes, in particular subsumption between classes. Recall that A subsumes B when it is the case that any instance of B must necessarily be

an instance of A.

INFERENCE ENGINES

An expert system has three levels of organization: a working memory, an inference engine, and a knowledge base. The inference engine is the control of the execution of reasoning rules. This means that it can be used to deduce new knowledge from existing information.

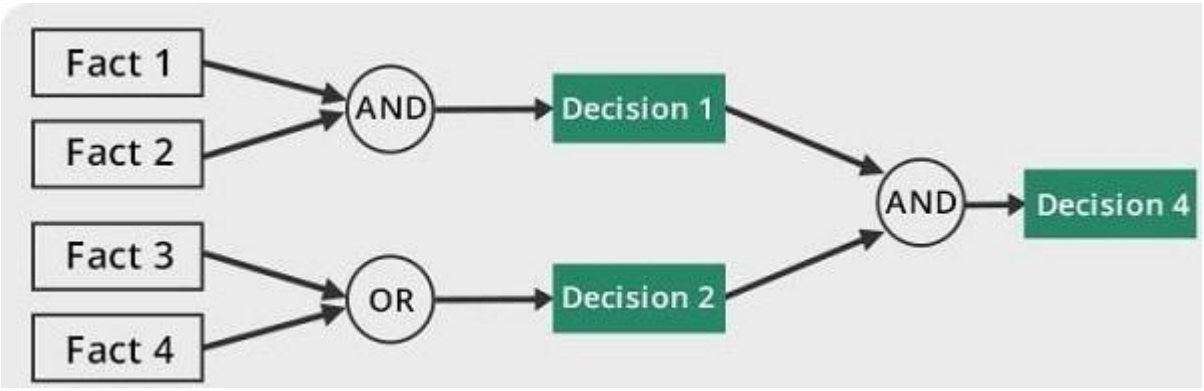
The inference engine is the core of an expert system and acts as the generic control mechanism that applies the axiomatic knowledge from the knowledge base to the task-specific data to reach some conclusion.

Two techniques for drawing inferences are general logic-based inference engines and specialized algorithms. Many realistic Web applications will operate agent-to-agent without human intervention to spot glitches in reasoning. Therefore developers will need to have complete confidence in reasoner otherwise they will cease to trust the results.

Forward chaining

Forward chaining is a method of reasoning in artificial intelligence in which inference rules are applied to existing data to extract additional data until an endpoint (goal) is achieved.

In this type of chaining, the inference engine starts by evaluating existing facts, derivations, and conditions before deducing new information. An endpoint (goal) is achieved through the manipulation of knowledge that exists in the knowledge base.



Properties of forward chaining

- The process uses a down-up approach (bottom to top).
- It starts from an initial state and uses facts to make a conclusion.
- It’s employed in expert systems and production rule system.

Examples of forward chaining

A simple example of forward chaining can be explained in the following sequence.

A

A->B

B

A is the starting point. A->B represents a fact. This fact is used to achieve a decision B.

A practical example will go as follows;

Tom is running (A)

If a person is running, he will sweat (A->B)

Therefore, Tom is sweating. (B)

Advantages

- It can be used to draw multiple conclusions.
- It's more flexible than backward chaining because it does not have a limitation on the data derived from it.

Disadvantages

- The process of forward chaining may be time-consuming.
- Unlike backward chaining, the explanation of facts or observations for this type of chaining is not very clear.The former uses a goal-driven method that arrives at conclusions efficiently.

Backward chaining

Backward chaining is a concept in artificial intelligence that involves backtracking from the endpoint or goal to steps that led to the endpoint. This type of chaining starts from the goal and moves backward to comprehend the steps that were taken to attain this goal.

The backtracking process can also enable a person establish logical steps that can be used to find other important solutions.

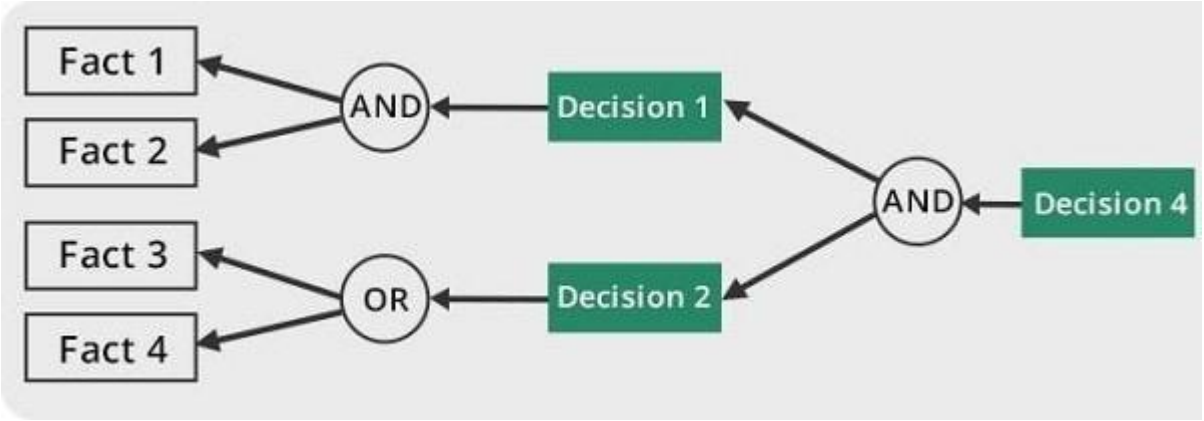


Image Source: Quora

Backward chaining can be used in debugging, diagnostics, and prescription applications.

Properties of backward chaining

- The process uses an up-down approach (top to bottom).
- It's a goal-driven method of reasoning.
- The endpoint (goal) is subdivided into sub-goals to prove the truth of facts.

Example of backward chaining

The information provided in the previous example (forward chaining) can be used to provide a simple explanation of backward chaining. Backward chaining can be explained in the following sequence.

B

A->B

A

B is the goal or endpoint, that is used as the starting point for backward tracking. A is the initial state. A->B is a fact that must be asserted to arrive at the endpoint B.

A practical example of backward chaining will go as follows:

Tom is sweating (B).

If a person is running, he will sweat (A->B).

Tom is running (A).

Advantages

- The result is already known, which makes it easy to deduce inferences.
- It's a quicker method of reasoning than forward chaining because the endpoint is available.
- In this type of chaining, correct solutions can be derived effectively if pre-determined rules are met by the inference engine.

Disadvantages

- The process of reasoning can only start if the endpoint is known.
- It doesn't deduce multiple solutions or answers.
- It only derives data that is needed, which makes it less flexible than forward chaining.

Conclusion

Backward and forward chaining are important methods of reasoning in artificial intelligence. These concepts differ mainly in terms of approach, strategy, technique, speed, and operational direction.

How the Inference Engine Works

In simple rule-based systems, there are two kinds of inference, forward and backward chaining.

Tree Searches

A knowledge base can be represented as a branching network or tree. There is a large number of tree searching algorithms available in the existing literature. However, the two basic approaches are depth-first search and breadth-first search.

The depth-first search algorithm begins at a node that represents either the given data (forward chaining) or the desired goal (backward chaining). It then checks to see if the left-most (or first) node beneath the initial node (call this node A) is a terminal node (i.e., it is proven or a goal). If not, it establishes node A on a list of subgoals outstanding. It then starts with node A and looks at the first node below it, and so on. If there are no more lower level nodes, and a terminal node has not been reached, it starts from the last node on the outstanding list and takes the next route of descent to the right.

Breadth-first search starts by expanding all the nodes one level below the first node. Then it systematically expands each of these nodes until a solution is reached or else the tree is completely expanded. This process finds the shortest path from the initial assertion to a solution. However, such a search in large solution spaces can lead to huge computational costs due to an explosion in the number of nodes at a low level in the tree.

Full First-Order Logic Inference Engines

Using full first-order logic for specifying axioms requires a full-fledged automated theorem

prover. First-order logic is semidecidable and inferencing is computationally intractable for large amounts of data and axioms.

Closed World Machine

The Closed World Machine (CWM) inference engine written in Python by Tim Berners-Lee and Dan Connolly is a popular Semantic Web program. It is a general-purpose data processor for the Semantic Web and is a forward-chaining reasoner that can be used for querying, checking, transforming, and filtering information. Its core language is RDF, extended to include rules.

RDF INFERENCE ENGINE

RDF is a system meant for stating meta-information through triples composed of a subject, a property, and an object. The subject and object can be either a designation like a URL or a set of another triple. Triples form a simple directed graph.

The first triple says that Smith owns a computer and the second says that there is a computer made by Apple. The third drawing, however, is composed of two triples, and it says that Smith owns a computer made by Apple.

Suppose these triples were placed in a database.

In the first query, the question is who owns a computer? The answer is ―Smith.‖ In the second query, the question is What make of computer are defined in the database? The third query, however asks who owns a computer and what is the make of that computer?

The query is a graph containing variables that can be matched with the graph. Should the graph in the database be more extended, it would have to be matched with a subgraph. So, generally for executing an RDF query what has to be done is called ―subgraph matching.‖

Following the data model for RDF the two queries are in fact equal because a sequence of statements is implicitly a conjunction.

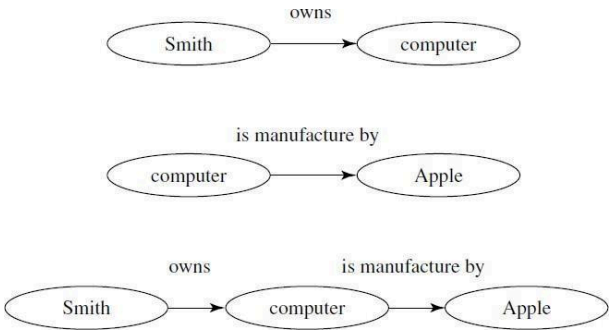


Fig: RDF Statements

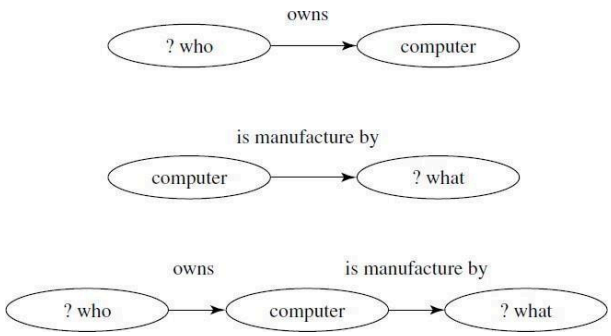


Fig: RDF

Queries Let us make a rule: If X owns a computer, then X must buy software.

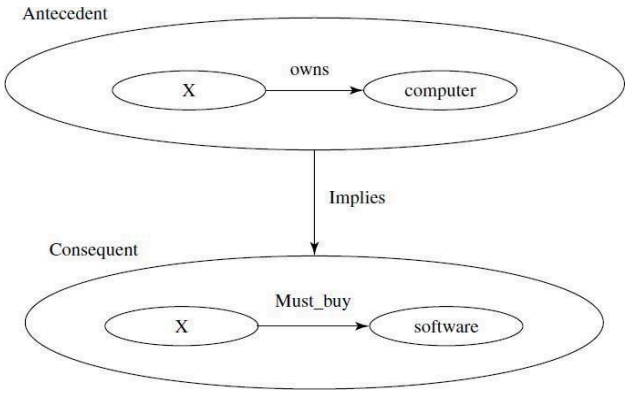
How do we represent such a rule?

The nodes of the rule form a triple set. Here there is one antecedent, but there could be more. There is only one consequent. (Rules with more than one consequent can be reduced to rules with one consequent.)

The desired answer is John must buy software. The query is matched with the consequent of the rule. Now an action has to be taken: The antecedents of the rule have to be added to the database with the variables replaced with the necessary values (substitution). Then the query has to be continued with the antecedent of the rule.

:

The question now is Who owns a computer? This is equal to a query described earlier. A rule subgraph is treated differently from nonrule subgraphs.



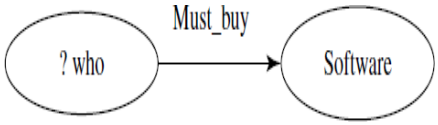


Fig: Graph representation of a rule **Fig: Query that matches with a rule**

A triple can be modeled as a predicate: triple(subject, property, object). A set of triples equals a list of triples and a connected graph is decomposed into a set of triples.

For our example this gives
Triple(John, owns, computer).
Triple(computer, make, Apple).

This sequence is equivalent to:
[Triple(John, owns, computer). Triple(computer,make, Apple).]

From **Figure RDF Queries** the triples are Triple(?who, owns, computer).

Triple(computer, make, ?what).

This sequence is equivalent to:
[Triple(?who, owns, computer). Triple(computer,make, ?what).]

From **Figure Graph representation of a rule** the triple is

Triple([Triple(X, owns, computer)], implies, [Triple(X, must buy, software)]).

From **Figure Query that matches with a rule** the triple is

Triple(?who, must buy, software).

A unification algorithm for RDF can handle subgraph matching and embedded rules by the term ***"subgraph matching with rules."***

The unification algorithm divides the sequence of RDF statements into sets where each set constitutes a connected subgraph. This is called a **tripleset** that is done for the database and for the query. Then the algorithm matches each tripleset of the query with each tripleset of the database.

Agents

Agents are pieces of software that work autonomously and proactively. In most cases, an agent will simply collect and organize information. Agents on the Semantic Web will receive some tasks to perform and seek information from Web resources, while communicating with other Web agents, in order to fulfill its task. Semantic Web agents will utilize metadata, ontologies, and

logic to carry out its tasks.