3

Knowledge Representation and Ontologies

Logic, Ontologies and Semantic Web Languages

Stephan Grimm[1], Pascal Hitzler[2] and Andreas Abecker[1]

[1] FZI Research Center for Information Technologies, University of Karlsruhe, Germany
{grimm,abecker}@fzi.de
[2] Institute AIFB, University of Karlsruhe, Germany,
hitzler@aifb.uni-karlsruhe.de

Summary. In Artificial Intelligence, knowledge representation studies the formalisation of knowledge and its processing within machines. Techniques of automated reasoning allow a computer system to draw conclusions from knowledge represented in a machine-interpretable form. Recently, ontologies have evolved in computer science as computational artefacts to provide computer systems with a conceptual yet computational model of a particular domain of interest. In this way, computer systems can base decisions on reasoning about domain knowledge, similar to humans. This chapter gives an overview on basic knowledge representation aspects and on ontologies as used within computer systems. After introducing ontologies in terms of their appearance, usage and classification, it addresses concrete ontology languages that are particularly important in the context of the Semantic Web. The most recent and predominant ontology languages and formalisms are presented in relation to each other and a selection of them is discussed in more detail.

3.1 Knowledge Representation

As a branch of symbolic Artificial Intelligence, *knowledge representation* and *reasoning* aim at designing computer systems that reason about a machine-interpretable representation of the world, similar to human reasoning. *Knowledge-based systems* have a computational model of some domain of interest in which symbols serve as surrogates for real-world domain artefacts, such as physical objects, events, relationships, etc. [45]. The *domain of interest* can cover any part of the real world or any hypothetical system about which one desires to represent knowledge for computational purposes.

A knowledge-based system maintains a *knowledge base* which stores the symbols of the computational model in form of statements about the domain, and it performs *reasoning* by manipulating these symbols. Applications can base their decisions on domain-relevant questions posed to a knowledge base.

3.1.1 A Motivating Scenario

To illustrate principles of knowledge representation in this chapter, we introduce an example scenario taken from a B2B travelling use case. In this scenario, companies frequently book business trips for their employees, sending them to international meetings and conference events. Such a scenario is a relevant use case for Semantic Web Services, since companies desire to automate the online booking process, while they still want to benefit from the high competition among various travel agencies and no-frills airlines that sell tickets via the Internet. Automation is achieved by computational agents deciding about whether an online offer of some travel agency fits a request for a business trip or not, based on the knowledge they have about the offer and the request. Knowledge represented in this domain of "business trips" is about flights, trains, booking, companies and their employees, cities that are source or destination for a trip, etc.

Knowledge-based systems use a computational representation of such knowledge in form of statements about the domain of interest. Examples of such statements in the business trips domain are "companies book trips for their employees", "flights and train rides are special kinds of trips" or "employees are persons employed at some company". This knowledge can be used to answer questions about the domain of interest. From the given statements, and by means of automated deduction, a knowledge-based system can, e.g., derive that "a person on a flight booked by a company is an employee" or "the company that booked a flight for a person is the person's employer".

In this way, a knowledge-based computational agent can reason about business trips, similar to the way a human would. It could, e.g., tell apart offers for business trips from offers for vacations, or decide whether the destination city for a requested flight is close to the geographical region specified in an offer, or conclude that a participant of a business flight is an employee of the company that booked the flight.

3.1.2 Forms of Representing Knowledge

If we look at current Semantic Web technologies and use cases, knowledge representation appears in different forms, the most prevalent of which are based on semantic networks, rules and logic. Semantic network structures can be found in RDF graph representations [30] or Topic Maps [41], whereas a formalisation of business knowledge often comes in form of rules with some "if-then" reading, e.g., in business rules or logic programming formalisms. Logic is used to realise a precise semantic interpretation for both of the other forms. By providing formal semantics for knowledge representation languages, logic-based formalisms lay the basis for automated deduction. We will investigate these three forms of knowledge representation in the following.

Semantic Networks

Originally, semantic networks stem from the "existential graphs" introduced by Charles Peirce in 1896 to express logical sentences as graphical node-and-link

diagrams [43]. Later on, similar notations have been introduced, such as conceptual graphs [45], all differing slightly in syntax and semantics. Despite these differences, all the semantic network formalisms concentrate on expressing the taxonomic structure of categories of objects and the relations between them. We use a general notion of a semantic network, abstracting from the different concrete notations proposed.

A *semantic network* is a graph whose nodes represent concepts and whose arcs represent relations between these concepts. They provide a structural representation of statements about a domain of interest. In the business trips domain, typical concepts would be "Company", "Employee" or "Flight", while typical relations would be "books", "isEmployedAt" or "participatesIn". Figure. 3.1 shows an example of a semantic network for the business trips domain.

Semantic networks provide a means to abstract from natural language, representing the knowledge that is captured in text in a form more suitable for computation. The knowledge expressed in the network from Fig. 3.1 coincides with the content of the following natural language text.

> *Employees of companies are persons, while both persons and companies are legal entities. Companies book trips for their employees. These trips can be flights or train rides which start and end in cities of Europe or the USA. Companies themselves have locations which can be cities.*
> *The company UbiqBiz books the flight FL4711 from London to New York for Mister X.*

Typically, concepts are chosen to represent the meaning of nouns in such a text, while relations are mapped to verb phrases. The fragment Company —*books*→ Trip is

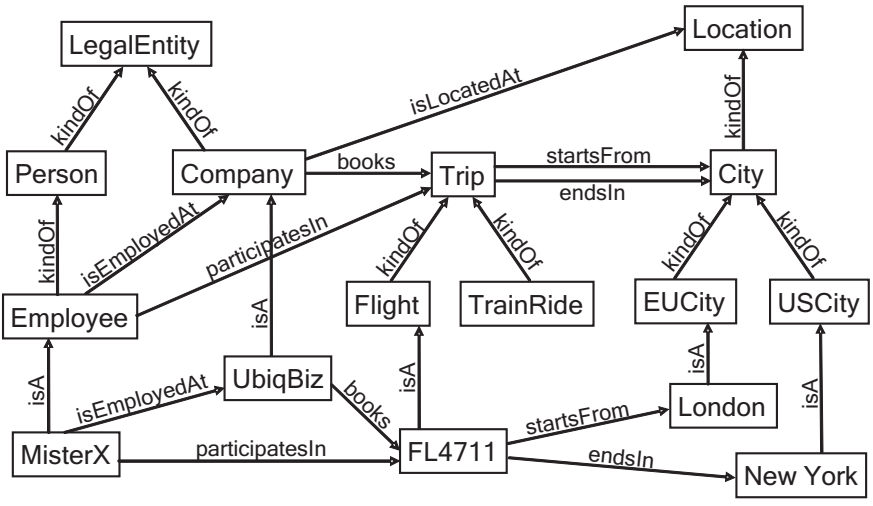


Fig. 3.1. A semantic network for business trips

read as "companies book trips", expressed as a binary relation between two concepts. However, this is not mandatory; the relation $\xrightarrow{books}$ could also be "lifted" to a concept Booking with relations $\xrightarrow{hasActor}$, $\xrightarrow{hasParticipant}$ and $\xrightarrow{hasObject}$ pointing to Company, Employee and Trip, respectively. In this way, its ternary character would be expressed more accurately than in the original network where the information about an employee's involvement in booking is implicit.

In principle, the concepts and relations in a semantic network are generic and could stand for anything relevant in the domain of interest. However, some particular relations for some standard knowledge representation and reasoning cases have evolved.

The semantic network in Fig. 3.1 illustrates the distinction between general concepts, like Employee, and individual concepts, like MisterX . While the latter represent concrete individuals or objects in the domain of interest, the former serve as classes to group together such individuals that have certain properties in common, as e.g. all employees. The particular relation which links individuals to their classes is that of *instantiation*, denoted by $\xrightarrow{isA}$. Thus, MisterX is called an instance of the concept employee. The lower part of the network is concerned with knowledge about individuals, reflecting a particular situation of the employee MisterX participating in a certain flight, while the upper part is concerned with knowledge about general concepts, reflecting various possible situations.

The most prominent type of relation in semantic networks, however, is that of *subsumption*, which we denote by $\xrightarrow{kindOf}$. A subsumption link connects two general concepts and expresses specialisation or generalisation, respectively. In the network in Fig. 3.1, a flight is said to be a special kind of trip, i.e. Trip subsumes Flight. This means that any flight is also a trip; however, there might be other trips which are not flights, such as train rides. Subsumption is associated with the notion of inheritance in that a specialised concept inherits all the properties from its more general parent concepts. For example, from the network one can read that a company can be located in a European city, since $\xrightarrow{locatedAt}$ points from Company to Location while EUCity is a kind of City which is itself a kind of Location . The concept EUCity inherits the property of being a potential location for a company from the concept Location .

Other particular relations that can be found in semantic network notations are, e.g., $\xrightarrow{partOf}$ to denote part-whole relationships, etc.

Semantic networks are closely related to another form of knowledge representation called frame systems. In fact, frame systems and semantic networks can be identical in their expressiveness but use different representation metaphors [43]. While the semantic network metaphor is that of a graph with concept nodes linked by relation arcs, the frame metaphor draws concepts as boxes, i.e. frames, and relations as slots inside frames that can be filled by other frames. Thus, in the frame metaphor the graph turns into nested boxes.

The semantic network form of knowledge representation is especially suitable for capturing the taxonomic structure of categories for domain objects and for expressing general statements about the domain of interest. Inheritance and other relations between such categories can be represented in and derived from subsumption

hierarchies. On the other hand, the representation of concrete individuals or even data values, like numbers or strings, does not fit well the idea of semantic networks.

Rules

Another natural form of expressing knowledge in some domain of interest are *rules* that reflect the notion of consequence. Rules come in the form of IF-THEN constructs and allow to express various kinds of complex statements. Rules can be found in logic programming systems, like the language Prolog [31], deductive databases [34] or business rules systems.

The following is an example of rules expressing knowledge in the business trips domain, specified in their intuitive if-then reading.

(1) IF *something is a flight* THEN *it is also a trip*
(2) IF *some person participates in a trip booked by some company*
THEN *this person is an employee of this company*
(3) FACT *the person MisterX participates in a flight booked by the company UbiqBiz*
(4) IF *a trip's source and destination cities are close to each other*
THEN *the trip is by train*

The IF part is also called the body of a rule, while the THEN part is also called its head. Typically, rule-based knowledge representation systems operate on facts, which are often formalised as a special kind of rule with an empty body. They start from a given set of facts, like rule (3) above, and then apply rules in order to derive new facts, thus "drawing conclusions".

However, the intuitive reading with natural language phrases is not suitable for computation, and therefore such phrases are formalised to predicates and variables over objects of the domain of interest. A formalisation of the above rules in the typical style of rule languages looks as follows.

(1) $\texttt{Trip}(?t) \texttt{ :- } \texttt{Flight}(?t)$
(2) $\texttt{Employee}(?p) \wedge \texttt{isEmployedAt}(?p, ?c) \texttt{ :-}$
$\texttt{Trip}(?t) \wedge \texttt{books}(?c, ?t) \wedge \texttt{Company}(?c) \wedge$
$\texttt{participatesIn}(?p, ?t) \wedge \texttt{Person}(?p)$
(3) $\texttt{Person}(\texttt{MisterX}) \wedge \texttt{participatesIn}(\texttt{MisterX}, \texttt{FL4711}) \wedge$
$\texttt{Flight}(\texttt{FL4711}) \wedge \texttt{books}(\texttt{UbiqBiz}, \texttt{FL4711}) \wedge \texttt{Company}(\texttt{UbiqBiz}) \texttt{ :-}$
(4) $\texttt{TrainRide}(?t) \texttt{ :-}$
$\texttt{Trip}(?t) \wedge \texttt{startsFrom}(?t, ?s) \wedge \texttt{endsIn}(?t, ?d) \wedge \texttt{close}(?s, ?d)$

In most logic programming systems, a rule is read as an inverse implication, starting with the head followed by the body, which is indicated by the symbol $\texttt{:-}$ that resembles a backward arrow. In this formalisation, the intuitive notions from the text, that were concepts and relations in the semantic network case, became predicates linked through variables and constants that identify objects in the domain of interest. Variables start with the symbol ? and take as their values the constants that occur in facts such as (3).

Rule (1) captures inheritance – or subsumption – between trips and flights by stating that "everything that is a flight is also a trip". Rule (2) draws conclusions

about the status of employment for participants of business flights. From the facts (3), these two rules are able to derive the implicit fact that "MisterX is an employee of UbiqBiz".

While the rules (1) and (2) express general domain knowledge, rule (4) can be interpreted as part of some company's travelling policy, stating that trips between close cities shall be conducted by train. In business rules, e.g., rule-based formalisms are used with the motivation to capture complex business knowledge in companies like pricing models or delivery policies.

Rule-based knowledge representation systems are especially suitable for reasoning about concrete instance data, i.e. simple facts of the form `Employee(MisterX)`. Complex sets of rules can efficiently derive implicit facts from explicitly given ones. They are problematic if more complex and general statements about the domain shall be derived which do not fit a rule's head.

Logic

Both forms, semantic networks as well as rules, have been formalised using logic to give them a precise semantics. Without such a precise formalisation they are vague and ambiguous, and thus problematic for computational purposes. From just the graphical representation of the semantic network in Fig. 3.1, e.g., it is not clear whether companies can only book flights for their own employees or for employees of partner companies as well. Neither is it clear from the fragment $\boxed{\text{Company}} \xrightarrow{books} \boxed{\text{Trip}}$ whether every company books trips or just some company. Also for rules, despite their much more formal appearance, the exact meaning remains unclear when, e.g., forms of negation are introduced that allow for potential conflicts between rules. Depending on the choice of procedural evaluation or flavour of formal semantics, different derivation results are being produced.

The most prominent and fundamental logical formalism classically used for knowledge representation is the "first-order predicate calculus", or *first-order logic* for short, and we choose this formalism to present logic as a form of knowledge representation here. First-order logic allows one to describe the domain of interest as consisting of objects, i.e. things that have individual identity, and to construct logical formulas around these objects formed by predicates, functions, variables and logical connectives [43]. We assume that the reader is familiar with the notation of first-order logic from formalisations of various mathematical disciplines.

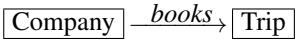
Similar to semantic networks, most statements in natural language can be expressed in terms of logical sentences about objects of the domain of interest with an appropriate choice of predicate and function symbols. Concepts are mapped to unary, relations to binary predicates. We illustrate the use of logic for knowledge representation by axiomatising parts of the semantic network from Fig. 3.1 more precisely.

Subsumption, e.g., can be directly expressed by a logical implication, which is illustrated in the translation of the following fragment.

$$\boxed{\text{Employee}} \xrightarrow{kindOf} \boxed{\text{Person}} \qquad \forall\, x : (Employee(x) \rightarrow Person(x))$$

Due to the universal quantifier, the variable x in the logical formula ranges over all domain objects and its reading is "everything that is an employee is also a person".

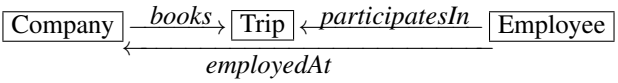
Other parts of the network can be further restricted using logical formulas, as shown in the following example.



$$\forall x, y : (books(x, y) \rightarrow Company(x) \wedge Trip(y))$$
$$\forall x : \exists y : (Trip(x) \rightarrow Company(y) \wedge books(y, x))$$

The graphical representation of the network fragment leaves some details open, while the logical formulas capture the booking relation between companies and trips more precisely. The first formula states that domain and range of the booking relation are companies and trips, respectively, while the second formula makes sure that for every trip there does actually exist a company that booked it.

In particular, more complex restrictions that range over larger fragments of a network graph can be formulated in logic, where the intuitive graphical notation lacks expressivity. As an example, consider the relations between companies, trips and employees in the following fragment.



$$\forall x : \exists y : (Trip(x) \rightarrow Employee(y) \wedge participatesIn(y, x) \wedge books(employer(y), x))$$

The logical formula expresses additional knowledge that is not captured in the graph representation. It states that, for every trip, there must be an employee that participates in this trip while the employer of this participant is the company that booked the flight.

Rules can also be formalised with logic. An IF-THEN rule can be represented as a logical implication with universally quantified variables. For example, a common formalisation of the rule

IF *a trip's source and destination cities are close to each other*
THEN *the trip is by train*

is the translation to the logical formula

$$\forall x, y, z : (Trip(x) \wedge startsFrom(x, y) \wedge endsIn(x, z) \wedge close(y, z) \rightarrow TrainRide(x)).$$

However, the typical rule-based systems do not interpret such a formula in the classical sense of first-order logic but employ different kinds of semantics, which are discussed in Sect. 3.2.

Since a precise axiomatisation of domain knowledge is a prerequisite for processing knowledge within computers in a meaningful way, we focus on logic as the dominant form of knowledge representation. Therefore, we investigate different kinds of logics and formal semantics more closely in a subsequent section.

In the context of the Semantic Web, two particular logical formalisms have gained momentum, reflecting the semantic network and rules forms of knowledge representation. The graph notations of semantic networks have been formalised through *description logics*, which are fragments of first-order logic with typical

Tarskian model-theoretic semantics but restricted to unary and binary predicates to capture the notions of concepts, an relations. On the other hand, rules have been formalised through *logic programming* formalisms with minimal model semantics, focusing on the derivation of simple facts about individual objects. Both description logics and logic programming can be found as underlying formalisms in various knowledge representation languages in the Semantic Web, which are addressed in Sect. 3.4.

3.1.3 Reasoning about Knowledge

The way in which we, as humans, process knowledge is by reasoning, i.e. the process of reaching conclusions. Analogously, a computer processes the knowledge stored in a knowledge base by drawing conclusions from it, i.e. by deriving new statements that follow from the given ones.

The basic operations a knowledge-based system can perform on its knowledge base are typically denoted by `tell` and `ask` [43]. The `tell` operation adds a new statement to the knowledge base, whereas the `ask` operation is used to query what is known. The statements that have been added to a knowledge base via the `tell` operation constitute the *explicit knowledge* a system has about the domain of interest. The ability to process explicit knowledge computationally allows a knowledge-based system to reason over a domain of interest by deriving *implicit* knowledge that follows from what has been told explicitly.

This leads to the notion of logical consequence or *entailment*. A knowledge base *KB* is said to entail a statement α if α "follows" from the knowledge stored in *KB*, which is written as $KB \models \alpha$. A knowledge base entails all the statements that have been added via the `tell` operation plus those that are their logical consequences. As an example, consider the following knowledge base with sentences in first-order logic.

$$\begin{aligned} KB = \{\ & Person(MisterX), participates(MisterX, FL4711), \\ & Flight(FL4711), books(UbiqBiz, FL4711), \\ & \forall x, y, z : (Flight(y) \wedge participates(x, y) \wedge books(z, y) \rightarrow employedAt(x, z)), \\ & \forall x, y : (employedAt(x, y) \rightarrow Company(x) \wedge Employee(y)), \\ & \forall x : (Person(x) \rightarrow \neg Company(x)) \qquad \} \end{aligned}$$

The knowledge base *KB* explicitly states that "*MisterX is a person who participates in the flight FL4711 booked by UbiqBiz*", that "participants of flights are employed at the company that booked the flight", that "the employment relation holds between companies and employees" and that "persons are different from companies". If we ask the question "Is MisterX employed at UbiqBiz?" by saying

$$\texttt{ask}(KB, employedAt(MisterX, UbiqBiz))$$

the answer will be yes. The knowledge base *KB* entails the fact that "MisterX is employed at UbiqBiz", i.e. $KB \models employedAt(MisterX, UbiqBiz)$, although

it was not "told" so explicitly. This follows from its general knowledge about the domain. A further consequence is that "UbiqBiz is a company", i.e. $KB \models Company(UbiqBiz)$, which is reflected by a positive answer to the question

$$\texttt{ask}(KB, Company(UbiqBiz)).$$

This follows from the former consequence together with the fact that "employment holds between companies and employees".

Another important notion related to entailment is that of consistency or *satisfiability*. Intuitively, a knowledge base is consistent or satisfiable if it does not contain contradictory facts. If we would add the fact that "UbiqBiz is a person" to the above knowledge base *KB* by saying

$$\texttt{tell}(KB, Person(UbiqBiz)),$$

it would become unsatisfiable because persons are said to be different from companies. We explicitly said that UbiqBiz is a person while at the same time it can be derived that it is a company.

In general, an unsatisfiable knowledge base is not very useful, since in logical formalisms it would entail any arbitrary fact. The `ask` operation would always return a positive result independent from its parameters, which is clearly not desirable for a knowledge-based system.

The inference procedures implemented in computational reasoners aim at realising the entailment relation between logical statements [43]. They derive implicit statements from a given knowledge base or check whether a particular statement is entailed by a knowledge base.

An inference procedure that only derives entailed statements is called *sound*. Soundness is a desirable feature of an inference procedure, since an unsound inference procedure would potentially draw wrong conclusions. If an inference procedure is able to derive every statement that is entailed by a knowledge base then it is called *complete*. Completeness is also a desirable property, since a complex chain of conclusions might break down if only a single statement in it is missing. Hence, for reasoning in knowledge-based systems we desire sound and complete inference procedures.

3.2 Logic-Based Knowledge-Representation Formalisms

First-order (predicate) logic is the prevalent and single most important knowledge representation formalism. Its importance stems from the fact that basically all current symbolic knowledge representation formalisms can be understood in their relation to first-order logic. Its roots can be traced back to the ancient Greek philosopher Aristotle, and modern first-order predicate logic was created in the 19th century, when the foundations for modern mathematics were laid.

First-order logic captures some of the essence of human reasoning by providing a notion of *logical consequence* as already mentioned. It also provides a notion of *universal truth* in the sense that a logical statement can be universally valid (and thus

called a *tautology*), meaning that it is a statement which is true regardless of any preconditions.

Logical consequence and universal truth can be described in terms of *model-theoretic semantics*. In essence, a model for a logical theory[3] describes a state of affairs which makes the theory true. A tautology is a statement for which all possible states of affairs are models. A logical consequence of a theory is a statement which is true in *all* models of the theory.

How to derive logical consequences from a theory – a process called *deduction* or *inferencing* – is obviously central to the study of logic. Deduction allows to access knowledge which is not explicitly given but implicitly represented by a theory. Valid ways of deriving logical consequences from theories also date back to the Greek philosophers, and have been studied since.

At the heart of this is what has become known as *proof theory*. Proof theory describes syntactic rules which act on theories and allow to derive logical consequences without explicit recurrence to models. The notion of universal truth can thus be reduced to syntactic manipulations. This allows to abstract from model theory and enables deduction by symbol manipulation, and thus by automated means.

Obviously, with the advent of electronic computing devices in the 20th century, the automation of deduction has become an important and influential field of study. The field of automated reasoning is concerned with the development of efficient algorithms for deduction. These algorithms are usually required to be sound, and completeness is a desired feature.

The fact that sound and complete deduction algorithms exist for first-order predicate logic is reflected by the statement that first-order logic is *semi-decidable*. More precisely, semi-decidability of first-order logic means that there exist algorithms which, given a theory and a query statement, terminate with positive answer in finite time whenever the statement is a logical consequence of the theory. Note that for semi-decidability, termination is not required if the statement is *not* a logical consequence of the theory and, indeed, termination (with the correct negative answer) cannot be guaranteed in general for first-order logical theories.

For some kinds of theories, however, sound and complete deduction algorithms exist which always terminate. Such theories are called *decidable*, and they have certain more-or-less obvious advantages, including the following.

- Decidability guarantees that the algorithm always comes back with a correct answer in finite time.[4] Under semi-decidability, an algorithm which runs for a considerable amount of time may still terminate, or may not terminate at all, and thus the user cannot know whether he has waited long enough for an answer. Decidability is particularly important if we want to reason about the question of whether *or not* a given statement is a logical consequence of a theory.

[3] A logical theory denotes a set of logical formulas, seen as the axioms of some theory to be modelled

[4] It should be noted that there are practical limitations to this due to the fact that computing resources are always limited. A theoretically sound, complete and terminating algorithms may thus run into resource limits and terminate without an answer

- Experience shows that practically efficient algorithms are often available for decidable theories due to the effective use of heuristics. Often, this is even the case if worst-case complexity is very high.

3.2.1 Description Logics

Description logics [3] are essentially decidable fragments of first-order logic,[5] and we have just seen why the study of these is important. At the same time, description logics are expressive enough such that they have become a major knowledge representation paradigm, in particular for use within the Semantic Web.

We will describe one of the most important and influential description logics, called $\mathcal{ALC}$. Other description logics are best understood as restrictions or extensions of $\mathcal{ALC}$. We introduce the standard description logic notation and give a formal mapping into standard first-order logic syntax.

The Description Logic $\mathcal{ALC}$

A description logic theory consists of statements about concepts, individuals and their relations. Individuals correspond to constants in first-order logic, and concepts correspond to unary predicates. In terms of semantic networks, description logic concepts correspond to general concepts in semantic networks, while individuals correspond to individual concepts. We deal with concepts first, and will talk about individuals later.

Concepts can be *named concepts* or *anonymous* (*composite*) *concepts*. Named concepts consist simply of a name, say "human", which will be mapped to a unary predicate in first-order logic. Composite concepts are formed from named concepts by use of concept constructors, similar to the formation of complex formulas out of atomic formulas in first-order logic. In $\mathcal{ALC}$, we have the *boolean constructors*

- conjunction $\sqcap$, which is binary
- disjunction $\sqcup$, which is binary
- negation $\neg$, which is unary.

Hence, if C and D are concepts, then $C \sqcap D$, $C \sqcup D$ and $\neg C$ are also concepts. Concept constructors can be nested arbitrarily. The translation of boolean constructors to first-order predicate logic is obvious. To give an example, the statement $C \sqcap \neg D$ translates to the formula $C(x) \wedge \neg D(x)$.

$\mathcal{ALC}$ statements relate named or anonymous concepts by means of one of the following:

- inclusion $\sqsubseteq$
- inverse inclusion $\sqsupseteq$
- equivalence $\equiv$.

[5] To be precise, there do exist some description logics which are not decidable. And there exist some which are not straightforward fragments of first-order logics. But for this general introduction, we will not concern ourselves with these

Their meaning in first-order logic are implication $\rightarrow$ inverse implication $\leftarrow$ and equivalence $\leftrightarrow$. Occurring free variables are universally quantified. To give an example, the statement $C \sqsubseteq D \sqcup \neg E$ translates to $\forall x : (C(x) \rightarrow (D(x) \vee \neg E(x)))$.

$\mathcal{ALC}$ provides two special classes as shortcuts, namely $\bot$ and $\top$. They are defined by means of the equivalences $\bot \equiv C \sqcap \neg C$ and $\top \equiv C \sqcup \neg C$, where C is some arbitrary concept. That is, $\bot$ is the empty concept, and $\top$ is the concept under which everything falls.

$\mathcal{ALC}$ allows the restricted further use of quantifiers by means of the so-called *role restrictions*. A *role* is a named entity which translates to a binary predicate in first-order logic. In the semantic network paradigm, roles are relations between concepts. Given such a role r and a (named or anonymous) concept C, the composite concepts $\forall\, r.C$ and $\exists\, r.C$ can be formed. Role restrictions and boolean constructors can be nested arbitrarily with each other to form anonymous concepts. The composite concept $\forall\, r.C$ translates to $\forall\, y : (r(x,y) \rightarrow C(y))$ in first-order logic, while $\exists\, r.C$ translates to $\exists\, y : (R(x,y) \wedge C(y))$.

An $\mathcal{ALC}$ *TBox*, finally, consists of a set of statements of the form $C \sqsubseteq D$, $C \sqsupseteq D$ or $C \equiv D$, where C and D are named or composite concepts. Obviously, any TBox can be translated to first-order logic, and thus inherits a logical consequence relation from it.

To give some examples for TBox statements from the business trips domain,

$$\mathit{Employee} \sqsubseteq \mathit{Person}$$

encodes the knowledge that every employee is a person, while

$$\mathit{Trip} \sqsubseteq \exists\, \mathit{bookedBy}.(\mathit{Company} \sqcup \mathit{Person})$$

states that every Trip is booked by a company or a person.

We now come to individuals, which correspond to constants in first-order logic. $\mathcal{ALC}$ allows to state that some individuals belong to (named or composite) concepts, e.g. $C(a)$ states that the individual a belongs to concept C. Similarly, a statement $r(a,b)$, where r is a role, means that the individuals a and b stand in relation r. The translation to first-order logic is obvious.

An $\mathcal{ALC}$ *ABox* consists of a set of statements of the form $C(a)$ or $R(a,b)$, where C is a named or anonymous concept, R is a role and a, b are individuals. An $\mathcal{ALC}$ *knowledge base* consists of an $\mathcal{ALC}$ ABox and an $\mathcal{ALC}$ TBox.

Examples for ABox statements are $\mathit{Flight}(\mathit{FL4711})$ and $\mathit{bookedBy}(\mathit{FL4711}, \mathit{UbiqBiz})$, with the obvious meanings.

$\mathcal{ALC}$ allows to define a basic form of knowledge bases. We have already mentioned that it appears to be somewhat akin to semantic networks, but differs in two important respects: $\mathcal{ALC}$ comes with a precise formal semantics via first-order logic, and it is more expressive due to the use of concept constructors.

Nevertheless, $\mathcal{ALC}$ is very restricted in expressiveness in comparison with other knowledge representation formalisms. This is apparent, e.g., by the very restricted kinds of first-order logical statements which are expressible in $\mathcal{ALC}$. In order to meet the requirements of practice, it is therefore necessary to extend expressiveness

of $\mathcal{ALC}$. These extensions are not necessarily of a kind such that a larger fragment of first-order logic is obtained. This is indeed just one of the ways of extending $\mathcal{ALC}$ which we will examine.

Decidability-Preserving Extensions to $\mathcal{ALC}$

We have seen before that decidability is a desirable property, and so the natural question arises, which extensions of $\mathcal{ALC}$ retain its decidability. Indeed, extending $\mathcal{ALC}$ while staying within first-order logic on the one hand, and while retaining decidability on the other, has been one of the driving forces behind description logic research in the recent past. We briefly describe some of these extensions. For a comprehensive treatment of description logics, see [3].

The following additions can be made to $\mathcal{ALC}$ while retaining decidability.[6]

- Roles (i.e. binary predicates) can have additional properties such as being transitive, symmetric or inverse to other roles.
- A role can be described as the inverse of another role.
- Roles can be arranged hierarchically, i.e. a statement such as $r \sqsubseteq s$ is allowed between roles, which translates to $\forall x, y : (r(x, y) \rightarrow s(x, y))$ in first-order logic.
- Individuals can be compared, e.g. by stating explicitly that two individuals are identical ($a = b$), or different ($a \neq b$).
- It is allowed to use the so-called *nominals* in the TBox. Nominals are classes which consist of an enumeration of exactly those elements which are in the class. For example, the statement $C \equiv \{a, b, c\}$ says that the class C contains exactly the elements a, b and c.
- Quantifiers can be generalised to *number restrictions*, which yields anonymous concepts such as $\leq n\, r$ and $\geq n\, r$, where r is a role, and n is a positive integer. The first of these describes the set of all individuals x for which less than or equal to n individuals y are in relation $r(x, y)$ to x. The meaning of the second construction is analogous. Note, e.g., that $\geq 1\, r$ is equivalent to $\exists\, r.\top$.
- Roles such as the ones described so far are also called *abstract roles*. Some description logics additionally allow the use of *concrete roles*, which allow to assign datatype values such as integers or strings to individuals.

$\mathcal{ALC}$, together with the above-mentioned additions, roughly constitutes the description logic $\mathcal{SHOIN}(\mathbf{D})$. The strange acronym comes from a certain agreed-upon standard for naming description logics, where each letter stands for a specific (group of) allowed constructor(s). The $\mathcal{S}$ stands for $\mathcal{ALC}$ together with transitivity for roles. $\mathcal{H}$ stands for role hierarchies. $\mathcal{O}$ and $\mathcal{I}$ stand for nominals and for the use of inverse roles, respectively. $\mathcal{N}$ stands for number restrictions. The $\mathbf{D}$, finally, stands for the use of concrete roles and datatypes.

[6] Some minor restrictions need to be respected, which we do not include here

Non-classical Semantics

$\mathcal{SHOIN}(\mathbf{D})$ is essentially still a decidable fragment of first-order predicate logic.[7] Certain expressive features, however, cannot be conveniently described by means of first-order logic. The study of such expressive features is motivated by Artificial Intelligence applications and has a long history in knowledge representation and reasoning, and most recently corresponding extensions and alterations of description logics are also being developed.

From a very general perspective, such expressive features are obtained by altering the notion of logical consequence. Recall that for first-order predicate logic a statement is a logical consequence of a theory if it is true in *all* models of the theory. Models of the theory, in turn, are interpretations (i.e. states of affairs) which make the theory true. An alternative notion of logical consequence can thus be derived by not selecting *all* interpretations which make the theory true, but only *some*, *more* or simply *other* such interpretations, and by calling those statements logical consequences, which are true in all these selected interpretations.

This endeavour, although it appears to be somewhat dubious at first, provides a general perspective on many expressive features in knowledge representation and reasoning. Important for this is certainly that the corresponding selections of interpretations are clearly defined and meaningful. Often, this selection is done most conveniently by means of additional syntax and, in the following, we will cover some additional expressive features which are most important for the Semantic Web context.

Let us remark that reasoning with expressive features is computationally expensive, and this fact is a well-known obstacle for developments in symbolic Artificial Intelligence. By means of description logics and the fact that they show reasonable scalability despite high worst-case complexities, expressive knowledge representation features become attractive for practical purposes. Of obvious importance is thus the identification of tractable description logics, as done e.g. in [18, 9, 2, 27].

3.2.2 Closed-World Assumption

The Closed-World Assumption (CWA) can be understood as a computational reinterpretation of *negation*. Roughly speaking, it is the assumption that *what cannot be proven is wrong*. Assume, e.g., the statement "if an employee is not booked on a trip at a certain date, then (s)he is available for internal meetings that day", and assume furthermore that there is no knowledge available whether the employee MisterX is booked on a trip on a certain day. Then, under the CWA, we would conclude that MisterX is available for an internal meeting on that particular day.

A CWA perspective is particularly natural from a database point of view. An employee is assumed to be *not* booked on a trip, unless the booking can be found in the database. Thus, the database describes a *closed world*, in which all statements are either the case (if they are explicitly known) or not the case (otherwise).

[7] More precisely, it corresponds to first-order predicate logic with equality. Care needs to be taken with the encoding of number restrictions, and datatypes must be allowed as required

Treating Semantic Web knowledge under CWA, however, is conceptually difficult in some cases. This comes from the open nature of the World Wide Web, where data is constantly added and changing. Thus, if a particular piece of knowledge cannot be retrieved from the Semantic Web, then it cannot safely be assumed to be false: the information may be contained on a web page which has not been included yet, but which will be crawled next. Such a situation should be treated under the *Open-World Assumption* (OWA), which assumes that only such conclusions should be drawn which will remain valid if new information is added.

The semantics of first-order predicate logic – and thus also of description logics – operates under the OWA. If we have no knowledge about whether a person is booked on a flight, then under the OWA we cannot conclude anything on this person's availability for an internal meeting from the example statement given above.

It is safe to assume that knowledge from databases will play a natural role in the realisation of the Semantic Web, and will come alongside knowledge from other sources, like the open web. Restricting knowledge representation to pure OWA or pure CWA settings is thus insufficient: while the basic framework for the open Semantic Web should be based on the OWA, a restricted use of the CWA should be possible at the same time. This integration has become known as *Local Closed World* (LCW) [16], and is currently being researched from several perspectives. We will say more about this in the next section on non-monotonicity.

3.2.3 Non-monotonicity

The original motivation for the study of non-monotonic reasoning comes from the observation that humans tend to *jump to conclusions* when making every day practical and commonsense decisions. If we book a train trip, then we conclude that we will not be arriving by bus, and in case we have to base further decisions on the knowledge, we simply assume the conclusion to be true. However, our knowledge about the real world is never complete. It may turn out, e.g., that there is a large power outage on the day of the trip so that the trains will not run – and as a substitute, we are being transported by bus on short notice.

When *jumping to conclusions*, it may be necessary to withdraw the conclusions if further knowledge becomes available. In the example just given, we withdraw the knowledge about not arriving by bus as soon as we learn about the special circumstances. In this sense, commonsense reasoning is *non-monotonic*.

More formally, a knowledge representation formalism is called *monotonic* if a larger theory implies *more conclusions* or, in other words, if the addition of knowledge never invalidates conclusions drawn before the addition. A knowledge representation formalism is *non-monotonic* if it is not monotonic.

First-order predicate logic – and thus also description logics – are monotonic. Formalisms operating under the CWA are usually non-monotonic: if a database does not contain a booking information for MisterX being on a business trip at a certain date, then it could be concluded that MisterX is available for internal meetings at this date by an appropriate rule; if, however, such a booking information becomes known and is added to the database, then the earlier conclusion must be withdrawn.

The strong relation between CWA and non-monotonicity is well known and has inspired many lines of research in these areas. Historically, there are three major approaches to non-monotonicity, which we briefly list in the following.

Default Logic [42] uses the so-called *default rules* of the form $(\alpha : \beta)/\gamma$ for expressing the following condition for formulas α, β and γ: if α is the case and β is possible, then conclude γ. To give an example, α could be the statement "FL4711 is a trip to a foreign country", β could be the statement "FL4711 is not a train ride", and γ could be the statement "FL4711 is a flight". We further assume that we indeed know that FL4711 is a trip to a foreign country. Without any further knowledge whether FL4711 is a flight or a train ride, we conclude by the default rule that FL4711 is a flight. If we add further knowledge that FL4711 is indeed a train ride, then the conclusion must be withdrawn. In this sense, a default rule is a rule that allows for exceptions.

Circumscription [33] realises non-monotonicity by means of a condition over logical predicates which ensures that in some cases truth or falsity of a statement is enforced although this would not be the case in classical first-order predicate logic. Circumscription is expressed by means of second-order logic (see Sect. 3.2.5), and does not require any extension of syntax.

Autoepistemic Logic [35] employs a modal logic operator to represent that something is *believed* (but not necessarily known).

All three historic approaches are being studied in the context of description logics, and central references are [4], [6] and [14], respectively. It is still an open quest to find out which of these is most suitable for Semantic Web applications. Of particular importance – besides the obvious scalability requirements – is the question how the formalism realises LCW reasoning in a practically useful way.

Historically, the area of non-monotonic reasoning received decisive impulses in the 1980s and 1990s from logic programming research, which we discuss next.

3.2.4 Logic Programming

Logic programming was originally conceived as a way to use (first-order predicate) logic as a programming language. In order to allow for efficient computation, formulas were syntactically restricted to the so-called *Horn clauses*. Additionally, only certain kinds of logical consequences are being considered.

Syntactically, Horn clauses can be understood as rules. For example, the expression $\mathit{Trip}(t) \vee \neg\ \mathit{Flight}(t)$ is a Horn clause, which is semantically equivalent (with respect to FOL) to $\forall t : \mathit{Trip}(t) \leftarrow \mathit{Flight}(t)$. This, in turn, can also be interpreted as the rule $\texttt{Trip}(?t)\ \texttt{:-}\ \texttt{Flight}(?t)$ from page 55.

Note, however, that the semantics of the Horn clause is given by means of first-order logic semantics, whereas logic programming rules are usually understood in a different sense. One of the differences stems from the fact that in a logic programming system only certain types of logical consequences are being

considered, namely ground[8] instances of predicates. In the example, the addition of a fact `Flight(FL4711)` would allow to conclude `Trip(FL4711)` both in FOL and in a logic programming system. A conclusion such as `Trip(FL4711)` $\vee$ $\neg$ `Flight(FL4711)`, however, would be possible only in FOL, and not derivable using logic programming semantics.

The second difference between the semantics concerns the handling of negative information. In the example above, we could be interested in whether the statement `Trip(FL2306)` holds. In FOL, neither truth nor falsity of this statement is derivable. In logic programming, however, the statement would be considered false. The handling of negative information in logic programming in this sense is based on the CWA: as no information on `FL2306` is available, it is considered to be *not* a trip.

Logic programming semantics is thus non-monotonic: just consider adding the single fact `Flight(FL2306)` to the knowledge base, by which `Trip(FL2306)` turns true. This insight triggered substantial research efforts on relating logic programming and non-monotonic reasoning, which led to the introduction of non-monotonic kinds of negation into the logic programming paradigm, see [1].

How to combine logic programming or other rules formalisms with description logics constitutes a recent research issue. Prominent approaches include the creation of hybrid systems by interfacing logic programming systems with description logic systems, as e.g. in [15]. Other approaches simply go back to Horn clauses and add them as FOL statements to description logic knowledge bases [26].

3.2.5 Higher-Order Logic

Another feature which is considered important for knowledge representation in the Semantic Web is what has become known as *metamodelling*. This occurs, e.g., whenever description logic classes should be considered as individual members of other (meta-)classes, or if properties shall be attached to entire classes by means of roles. Logically, this corresponds to using high-order logics, and generally results in the loss of decidability. Decidable fragments, however, can be described, as in [36].

To give an example, consider an international company using a semantics-based knowledge management system for business trips, which requires that different languages spoken within the company are supported by the system. It may thus be necessary to represent the knowledge that the concept *Flight* is called "*Flug*" in German. This could be represented by using a concrete role statement like *germanName*(*Flight*, "*Flug*"). Here, "*Flug*" would be a data value of type string, while the concept *Flight* actually appears syntactically as an individual. Notice that here a data value is directly assigned to a concept rather than to its instances.

3.2.6 Treatment of Inconsistencies

A point of particular importance for the Semantic Web lies in a sensible treatment of inconsistencies in knowledge bases. This comes from the fact that in Semantic

[8] A ground (instance of a) predicate is an atomic formula which does not contain any variable symbols

Web applications it is very often necessary to merge different knowledge bases from different sources, and it can be expected that in many cases some parts of the respective knowledge bases may conflict with each other, resulting in inconsistency. In a classical FOL setting, a single inconsistency causes a knowledge base to be entirely useless. For practical purposes, however, it should be possible to rescue at least some of the knowledge in a constructive way in order to draw meaningful conclusions from the knowledge.

There exist two basic approaches to dealing with inconsistency. The first one is based on the intuition that inconsistencies point to mistakes in modelling, and thus should be repaired. Technically, such repairs can be done by identifying, e.g., maximal consistent subsets of the knowledge base and using those for drawing conclusions, see e.g. [48]. The other approach is based on using the so-called *paraconsistent logics* with an additional truth value which represents contradiction, see e.g. [50].

3.2.7 Uncertainty

Knowledge is often acquired by machine learning techniques. Knowledge base statements obtained this way are usually uncertain, e.g. in a probabilistic sense or in the sense of fuzzy logic. Recent efforts are thus under way to provide methods and tools for the representation and the reasoning with uncertainty in description logics.

To give an example, consider a business trips booking Internet portal which uses a knowledge base for providing personalised content to the user. From the usage patterns of *UbiqBiz* customers the knowledge base knows with a probability of 80% that a *UbiqBiz* customer browsing the portal will be interested in booking a flight, and is thus able to provide appropriate personalised content. As part of a sophisticated personalisation knowledge base, the treatment of such probabilities and other uncertainty values becomes important.

3.3 Ontologies in Information Systems

Recently, the notion of ontologies as computational artefacts has appeared in Artificial Intelligence and Computer Science, while "ontology" originally denotes the study of existence in philosophy. In information systems, ontologies are conceptual models of what "exists" in some domain, brought into machine-interpretable form by means of knowledge representation techniques. In this section we start from a general definition of the notion of ontology and elaborate on its appearance and usage in computer science.

3.3.1 Ontology

In its original meaning in philosophy, *ontology* is a branch of metaphysics and denotes the philosophical investigation of existence. It is concerned with the fundamental questions of "what is being?" and "what kinds of things are there?" [11].

Dating back to Aristotle, the question of "what exists?" lead to studying general categories for all things that exist. Ontological categories provide a means to classify all existing things, and the systematic organisation of such categories allows to analyse the world that is made up by these things in a structured way. In ontology, categories are also referred to as *universals*, and the concrete things that they serve to classify are referred to as *particulars*.

Philosophers have mostly been concerned with general top-level hierarchies of universals that cover the entire physical world. Examples of universals occurring in such top-level hierarchies are most general and abstract concepts like "substance", "physical object", "intangible object", "endurant" or "perdurant". Philosophers have argued about the appropriateness of different such abstract categorisations and about the general properties of everything existing. Transferred to knowledge representation and computer science, information systems can benefit from the idea of ontological categorisation. When applied to a limited domain of interest in the scope of a concrete application scenario, ontology can be restricted to cover a special subset of the world. Examples of ontological categories in the business trips domain are "Person", "Company", "Trip" or "Flight", whereas examples for particular individuals that are classified by these categories are the person "MisterX", the company "UbiqBiz" or the particular flight "FL4711".

In general, the choice of ontological categories and particular objects in some domain of interest determines the things about which knowledge can be represented in a computer system [45]. In this sense, ontology provides the labels for nodes and arcs in a semantic network or the names for predicates and constants in rules or logical formulas that constitute an *ontological vocabulary*. By defining "what exists" it determines the things that can be predicated about. The terms of the ontological vocabulary are then used to represent knowledge, forming statements about the domain.

3.3.2 Ontologies

While "ontology" studies what exists in a domain of interest, "an ontology" as a computational artefact encodes knowledge about this domain in a machine-processable form to make it available to information systems.

Definition of an Ontology

In various application contexts, and within different communities, ontologies have been explored from different points of view, and there exist several definitions of what an ontology is. Within the Semantic Web community the dominating definition of *an ontology* is the following, based on [19].

> An *ontology* is a formal explicit specification of a shared conceptualisation of a domain of interest.

This definition captures several characteristics of an ontology as a specification of domain knowledge, namely the aspects of formality, explicitness, being shared, conceptuality and domain-specificity, which require some explanation.

- *Formality*
  An Ontology is expressed in a knowledge representation language that provides a formal semantics. This ensures that the specification of domain knowledge in an ontology is machine-processable and is being interpreted in a well-defined way. The techniques of knowledge representation help to realise this aspect.
- *Explicitness*
  An ontology states knowledge explicitly to make it accessible for machines. Notions that are not explicitly included in the ontology are not part of the machine-interpretable conceptualisation it captures, although humans might take them for granted by common sense.[9]
- *Being shared*
  An ontology reflects an agreement on a domain conceptualisation among people in a community. The larger the community the more difficult it is to come to an agreement on sharing the same conceptualisation. Thus, an ontology is always limited to a particular group of people in a community, and its construction is associated with a social process of reaching consensus.
- *Conceptuality*
  An ontology specifies knowledge in a conceptual way in terms of symbols that represent concepts and their relations. The concepts and relations in an ontology can be intuitively grasped by humans, as they correspond to the elements in our mental model. (In contrast to this, the weights in a neural network or the probability measures in a Bayesean network would not fit such a conceptual and symbolic approach.) Moreover, an ontology describes a conceptualisation in general terms and does not only capture a particular state of affairs. Instead of making statements about a specific situation involving particular individuals, an ontology tries to cover as many situations as possible, that can potentially occur [21].
- *Domain specificity*
  The specifications in an ontology are limited to knowledge about a particular domain of interest. The narrower the scope of the domain for the ontology, the more an ontology engineer can focus on axiomatising the details in this domain rather than covering a broad range of related topics. In this way, the explicit specification of domain knowledge can be modularised and expressed using several different ontologies with separate domains of interest.

Technically, the principal constituents of an ontology are *concepts*, *relations* and *instances*. Concepts map to the generic nodes in semantic networks, or to unary

[9] Notice that this notion of explicitness is different from the distinction between explicit and implicit knowledge, introduced earlier. Implicit knowledge that can be derived by means of automated deduction does not need to be included in an ontology for a computer system to access it. However, knowledge that is neither explicitly stated nor logically follows from what is stated can by no means be processed within the machine, although it might be obvious to a human. Such knowledge remains implicit in the modeller's mind and is not represented in the machine

predicates in logic, or to concepts as in description logics. They represent the ontological categories that are relevant in the domain of interest. Relations map to arcs in semantic networks, or to binary predicates in logic, or to roles in description logics. They semantically connect concepts, as well as instances, specifying their interrelations. Instances map to individual nodes in semantic networks, or to constants in logic. They represent the named and identifiable concrete objects in the domain of interest, i.e. the particular individuals which are classified by concepts.

These elements constitute an ontological vocabulary for the respective domain of interest. An ontology can be viewed as a set of statements, expressed in terms of this vocabulary, which are also referred to as *axioms*. A simple axiom would, e.g., state that "Mister X is an employee", involving an instance and a concept. A more complex axiom could state that "only employees of a particular company can be on trips booked by this company", imposing a restriction on a relation between two concepts.

Conceptual modelling with ontologies seems to be very similar to modelling in object-oriented software development or to designing entity-relationship diagrams for database schemas. However, there is a subtle twofold difference. First, ontology languages usually provide a richer formal semantics than object-oriented or database-related formalisms. They support encoding of complex axiomatic information due to their logic-based notations. Hence, an ontology specifies a semantically rich axiomatisation of domain knowledge rather than a mere data or object model. Second, ontologies are usually developed for a different purpose than object-oriented models or entity-relationship diagrams. While the latter mostly describe components of an information system to be executed on a machine and a schema for data storage, respectively, an ontology captures domain knowledge as such and allows to reason about it.

In summary, an ontology used in an information system is a conceptual yet executable model of an application domain. It is made machine-interpretable by means of knowledge representation techniques and can therefore be used by applications to base decisions on reasoning about domain knowledge.

Appearance of Ontologies

When engineered for or processed by information systems, ontologies appear in different forms related to the forms of knowledge representation which we discussed. A knowledge engineer views an ontology by means of some graphical or formal visualisation, while for storage or transfer it is encoded in an ontology language with some machine-processable serialisation format. A reasoner, in turn, interprets an ontology as a set of axioms that constitute a logical theory. We illustrate these different forms of appearance in ontology engineering, machine-processing and reasoning by an example.

Our business trips scenario, introduced earlier, involves several domains of interest. On the one hand, reasoning about business trips requires knowledge about travelling infrastructure for trains, flights and rental cars, while on the other hand it

involves financial knowledge about prices, different currencies and methods of payment when it comes to comparing different offers. Yet another related domain is that of geographic knowledge about locations of sources and destinations for trips, which we pick up as an example to illustrate appearance of ontologies. All these different domains of interest can be thought of as being captured by a modularised set of ontologies to which an information system in the business trips scenario can have access.

A geographic ontology suitable for a business trips booking system encodes countries and continents with their geographic regions, as well as geographic features like rivers, roads, rail tracks or cities. It relates geographic features to their regions, stating, e.g., that a city occupies a certain region, and it defines containment between such regions; the geographic region of a European city is, e.g., contained in that of Europe. Besides these general geographic concepts and their relations, such an ontology also determines concrete instances, such as particular cities, countries and continents, and relates them appropriately.

To a knowledge engineer an ontology is often visualised as some form of semantic network. Figure 3.2 shows the graphical visualisation of an example geographic ontology.

As common to most ontology development environments,[10] the visualisation in Fig. 3.2 presents to the knowledge engineer a taxonomy, i.e. a subsumption hierarchy, of the concepts in the ontology, which is indicated by $\xrightarrow{isa}$ links. The two taxonomies exposed in the graph are those for *GeographicRegion* with subconcepts

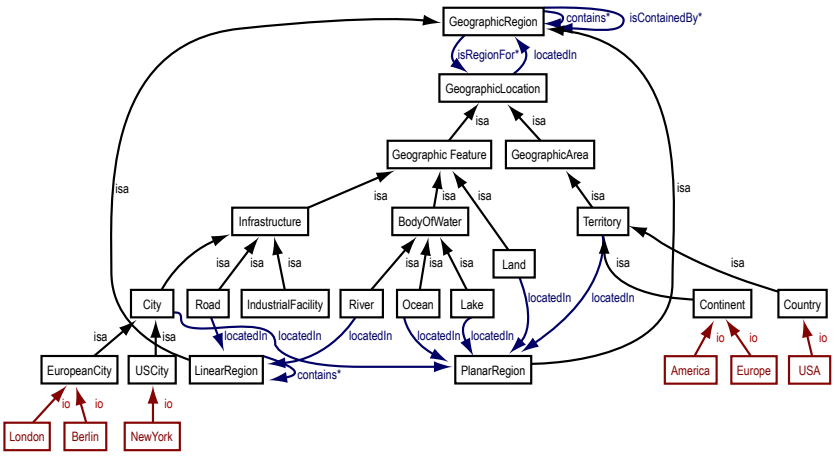


Fig. 3.2. A graphical visualisation for a geographic ontology

[10] The ontology graph in Fig. 3.2 has been produced with the OntoViz-plugin for the Protégé environment (`http://protege.stanford.edu/plugins/owl/`)

for linear and planar regions, and for *GeographicLocation* with subconcepts for geographic features, like cities or rivers, and geographic areas, like continents or countries. In the visualisation, the knowledge engineer can also see conceptual relations as arcs pointing from their domain concept to their range concept. By the relation $\xrightarrow{\textit{locatedIn}}$ between *GeographicLocation* and *GeographicRegion* a location, such as a city or a country, is associated to some region in which it is actually located. A *Road* or *River* is further restricted to be located in a *LinearRegion*, whereas a *City* or *Lake* is located in a *PlanarRegion* encompassing a surface area. The graph also shows some concrete cities and countries, modelled as instances of their respective concepts, which here serve as representatives for all the particular geographic places such an ontology would be populated with.

Not all the information in an ontology can easily be visualised in a graph as the one shown in Fig. 3.2. For some more detailed information, such as complex axioms and restrictions on concepts, there does not exist to date any appropriate visualisation paradigm other than exposing such fragments of the ontology in a formal language. Therefore, ontology engineering environments usually provide extra means for displaying and editing such complex axiomatic information, using a special-purpose ontology language or logical formal notation. When the environment exports the ontology for storage on a disk or for transfer over the wire, all of its information is expressed in the ontology language supported by the tool. Hence, the way an ontology appears to a developer of an ontology editor, storage facility or reasoner is in the form of ontology language constructs in some serialisation format suitable for machine processing.

There are various ontology languages, based on different knowledge representation formalisms, and we investigate the most prevalent of them in Sect. 3.4. For illustrating a fragment of our example geographic ontology, we choose the OWL[11] ontology language. The following listing displays a part of the ontology encoded in the OWL RDF serialisation format.

```
...
<owl:Class rdf:ID="City">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#locatedIn"/>
            <owl:allValuesFrom rdf:resource="#PlanarRegion"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="#Infrastructure"/>
    <owl:disjointWith rdf:resource="#Road"/>
    <owl:disjointWith rdf:resource="#IndustrialFacility"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="locatedIn">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:domain rdf:resource="#GeographicLocation"/>
    <rdfs:range rdf:resource="#GeographicRegion"/>
    <owl:inverseOf rdf:resource="#isRegionFor"/>
</owl:ObjectProperty>
<EuropeanCity rdf:ID="London"/>
...
```

[11] The DL-based Web Ontology Language (OWL) is popular in the Semantic Web context, and it is described in Sect. 3.4 among other languages

The listing shows an excerpt of the geographic ontology as it is serialised and parsed by tools and transferred over the network. It exhibits the specification of OWL classes (concepts), properties (relations) and individuals (instances), all expressed by tags and attributes of a customised XML serialisation. The *City* concept is defined as a subconcept of *Infrastructure* with the restriction that the relation $\xrightarrow{locatedIn}$ can only have instances of *PlanarRegion* as values. The relation $\xrightarrow{locatedIn}$ is defined as functional (having a unique value) and as being inverse to $\xrightarrow{isRegionFor}$, with proper domain and range concepts. *London* is introduced as an instance of *EuropeanCity*.

As ontology languages like OWL are based on logical formalisms, the formal semantics of the language precisely defines the meaning of an ontology in terms of logic. To a reasoner, therefore, an ontology appears as a set of logical formulas that express the axioms of a logical theory. It can verify whether these axioms are consistent or derive logical consequences. This form of appearance of an ontology is free of syntactical or graphical additions or ambiguities and reflects the pure knowledge representation aspect.

We use the description logic notation for OWL to exemplify some of the axioms in our example geographical ontology in their logical form. The following DL formulas constitute the definition of a European city.

$$
\begin{array}{rcl}
\exists\, locatedIn.\top & \sqsubseteq & GeographicLocation \\
\top & \sqsubseteq & \forall\, locatedIn.GeographicRegion \\
\exists\, contains.\top & \sqsubseteq & GeographicRegion \\
\top & \sqsubseteq & \forall\, contains.GeographicRegion \\
GeographicLocation & \sqsubseteq & =1\, locatedIn \\
Continent & \sqsubseteq & GeographicLocation \\
Continent\ (Europe) & & \\
PlanarRegion & \sqsubseteq & GeographicRegion \\
City & \sqsubseteq & GeographicLocation \sqcap \forall\, locatedIn.PlanarRegion \\
EuropeanCity & \equiv & City \sqcap \forall\, locatedIn.\exists\, contains^{-}.\exists\, locatedIn^{-}.\{Europe\}
\end{array}
$$

The last, quite sophisticated formula defines the concept of a European city by its geographical region being contained in the geographical region of the European continent. It has the following translation to first-order logic.

$$
\begin{array}{l}
\forall\, x : (EuropeanCity(x) \leftrightarrow \\
\quad City(x) \wedge \forall\, y : (locatedIn(x,y) \rightarrow \exists\, z : (contains(z,y) \wedge locatedIn(Europe,z))))
\end{array}
$$

In prose, its reading is as follows: "European cities are cities for which all geographic regions they are located in are contained in some geographic region in which Europe is located." This allows a knowledge-based system to decide whether a city is European by reasoning over containment of geographic regions.

In this logical form, an ontology is the set of axioms that constitutes the explicit knowledge represented about its domain of interest. By means of automated deduction, implicit knowledge of the same form can be derived but is not part of the ontology's explicit specification.

3.3.3 Usage of Ontologies

Often, an ontology is distinguished from a knowledge base in that it is supposed to describe knowledge on a schema level, i.e. in terms of conceptual taxonomies and general statements, whereas the more data-intensive knowledge base is thought of containing instance information on particular situations. We take a different perspective and perceive the relation between an ontology and a knowledge base as the connection between an epistemological specification of domain knowledge and a technical tool for reasoning. From this point of view, an ontology is a piece of knowledge that can be used by a knowledge-based application among other pieces of knowledge, e.g. other ontologies or meta data. To properly cover its domain of interest, it can make use of both schema level and instance level information. Whenever the knowledge-based system needs to consult the ontology, it loads (parts of) its specification into a knowledge base, most likely together with other pieces of knowledge, to take it into account for reasoning. The business trips booking system, e.g., would probably make combined use of a geographical ontology, a financial one, and one for public transportation, when comparing offers for trips, loading all relevant domain knowledge in its knowledge base. In this sense, a knowledge-based application uses an ontology via its knowledge base.

The computational domain model of an ontology can be used for various purposes, depending on the application scenario. We distinguish the different cases of usage on diverse levels, as follows.

- Level of knowledge connectivity
  An application can view an ontology as its single and isolated source of knowledge in a stand-alone fashion. This is the way an expert system maintains a highly specialised knowledge base to answer questions in its domain of interest, simulating expert knowledge.

  In contrast to this, an ontology can also be viewed in relation to other sources of knowledge, such as other ontologies or meta data that is aligned to the ontology's conceptual model. In an information integration scenario, e.g., an ontology supports interoperability among different systems on the knowledge or data level, providing a basic domain vocabulary.
- Level of knowledge abstraction
  On the one hand, an application can process an ontology on the schema level of knowledge about categories. Examples for this are applications which need to automatically classify user-defined concepts in an existing taxonomy or which build upon answers to general domain questions.

  On the other hand, an ontology can be used as a schema for data-intensive instance retrieval on large knowledge or databases.
- Level of automation in knowledge processing
  An application can make intensive use of automated reasoning techniques in order to derive implicit knowledge from the axioms in an ontology, answering sophisticated domain questions.

At the same time, ontologies can also be used for documentation and reference purposes, targeting humans to read their specifications rather than machines. This way, the documentation of domain models benefits from precise specification through the formal semantics of ontology languages.

In Artificial Intelligence research, some typical types of applications have evolved that make use of ontologies in different ways. We list some of them as examples of how applications can leverage the formalised conceptual domain models that ontologies provide.

- Information integration
  A promising field of application for ontologies is their use for integrating heterogeneous information sources on the schema level. Often, different databases store the same kind of information but adhere to different data models. An ontology can be used to mediate between database schemas, allowing to integrate information from differently organised sources and to interpret data from one source under the schema of another.

  Our example geographic ontology could be used to integrate geographic databases with different schemas; for example, one relating cities directly to their countries as different entities and another modelling a single entity for geographic places which have the property of being either a city or a country. In either schema, the local entities and relations can be mapped to the respective notions of *City*, *Country*, *GeographicRegion* and $\xrightarrow{locatedIn}$ in the ontology, realising unified querying and reasoning over both information sources.
- Information retrieval
  Motivated by the success and key role of Google[12] in the World Wide Web, information retrieval on web documents is a major field of application for ontologies. The idea behind ontology-based information retrieval is to increase the precision of retrieval results by taking into account the semantic information contained in queries and documents, lifting keywords to ontological concepts and relations.

  When interpreted according to our example geographic ontology, a query like "capital of Germany" would yield documents that are about Berlin, the capital of Germany. Some of the false positive matches that keyword-based retrieval systems typically produce, such as documents about the German venture capital market, can be filtered out this way.
- Semantically enhanced content management
  In many areas of computation, the data that is actually computed is annotated with meta data for various purposes. Ontologies provide the domain-specific vocabulary for annotating data with meta data. The formality of ontology languages allows for an automated processing of this meta data and their grounding in knowledge representation facilitates machine-interpretability.

  The geographic concepts and relations provided by our example ontology could be used to annotate manifold geographic content, such as geographic books and articles in an electronic library to better find and archive them or 3D-models

[12] `http://www.google.com`

of geographic sites in surveying and mapping, in order to better group and relate them, providing easier access to their content.

- Knowledge management and community portals
  In companies or other organised associations, or in communities of practice, individual knowledge can be viewed as a strategic resource that is desirable to be shared and systematically maintained, which is referred to as *knowledge management*. Ontologies provide a means to unify knowledge management efforts under a shared conceptual domain model, connecting technical systems for navigating, storing, searching and exchanging community knowledge.

  Our example ontology could serve as the backbone for a geographic knowledge portal in the Internet, through which land surveying offices, urban planning institutions and other interested community members provide access to geography-related resources.
- Expert systems
  In various domains, such as medical diagnosis or legal advice in case-law, it is desirable to simulate a domain expert who can be asked sophisticated questions. In an expert system, this is achieved by incorporating a thoroughly developed domain ontology that formalises expert knowledge. Domain-specific questions can then be answered by reasoning over such highly specialised knowledge.

  An expert system for the geographical domain could answer questions like "Which is the German city closest to the French border? "or "Through which cities does the river Rhein flow? ".

3.3.4 Types of Ontologies

Since the beginning of ontology research in Computer Science, ontologies have been considered as a means to foster reuse within knowledge-based system engineering, and it turned out that different types of ontologies exhibit a different potential for reuse.

A categorisation of ontologies can be made according to their subject of conceptualisation. The most prominent insights in this respect have been published in [20] and are summarised in Fig. 3.3.

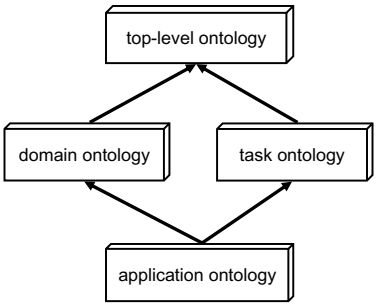


Fig. 3.3. Types of ontologies

The categorisation in Fig. 3.3 distinguishes the following types of ontologies.

- Top-level ontologies
  Top-level ontologies – also called upper ontologies or foundational ontologies – attempt to describe very abstract and general concepts that can be shared across many domains and applications. They borrow from philosophical notions, describing top-level concepts for all things that exist, such as "physical object" or "abstract object", as well as generic notions of common-sense knowledge about phenomena as time, space, processes, etc. They are usually well thought out and extensively axiomatised. Due to their generality, they are typically not directly used in applications but for other ontologies to be aligned to. Prominent examples for top-level ontologies are DOLCE [17] and SUMO [39].
- Domain ontologies and task ontologies
  These types of ontologies capture the knowledge within a specific domain of discourse, such as medicine or geography, or the knowledge about a particular task, such as diagnosing or configuring. In this sense, they have a much narrower and more specific scope than top-level ontologies. In the ideal case, the conceptualisation in a domain ontology is kept strictly task independent, while the notions in a task ontology are described neutrally with respect to a domain. Much work has been done in the development of domain ontologies in medicine, genetics, geographic and environment information, tourism, as well as cultural heritage and museum exhibits. Task ontologies have been devised, e.g., for scheduling and planning tasks, monitoring in a scientific domain, intelligent computer-based tutoring, missile tracking, execution of clinical guidelines, etc.
- Application ontologies
  Further narrowing the scope, application ontologies provide the specific vocabulary required to describe a certain task enactment in a particular application context. They typically make use of both domain and task ontologies, and describe, e.g., the role that some domain entity plays in a specific task. For example, a particular physical entity in some engineering domain may play the role of a replaceable unit in a machine diagnosis and maintenance task, and at the same time play the role of a spare resource in a configuration or production process.

Altogether, we can say that the lattice indicated in Fig. 3.3 represents an inclusion hierarchy: the lower ontologies inherit and specialise concepts and relations from the upper ones. The lower ontologies are more specific and have thus a narrower application scope, whereas the upper ones have a broader potential for reuse.

3.3.5 Ontologies in the Semantic Web

In the context of the Semantic Web, ontologies play a particularly important key role. The idea of the Semantic Web is to annotate web content by machine-interpretable meta data such that computers are able to process this content on a semantic level. Ontologies provide the domain vocabulary in terms of which semantic annotation

is formulated. Meta statements about web content in such annotations refer to a commonly used domain model by including the concepts, relations and instances of a domain ontology. The formality of ontology languages allows to reason about semantic annotation from different sources, connected to background knowledge in the domain of interest. There are a couple of characteristics of the web which affect the use of ontologies for semantic annotation.

One aspect is the natural distributedness of content in the Semantic Web. The knowledge captured in semantic annotation and ontologies is not locally available at a single node but spread over different sites. This poses additional constraints on the use of ontologies in the Semantic Web, taking into account distributedness of knowledge. To avoid the need to transfer relevant knowledge to a central location, there should be techniques that allow for a modularisation of the reasoning process by handling partial results that are computed locally, based on a subset of all relevant information. This issue is addressed by current research on *distributed reasoning*.

Another related aspect is that content on the web is created in an evolutionary manner and maintained in a decentralised way. There is no central control over semantic annotation and ontologies that evolve in the Semantic Web, and information in one ontology can conflict with information in another one. To deal with conflicting pieces of knowledge, there should be techniques that resolve such situations by, e.g., preferring one or another consistent sub view, similar to how humans would do. Such techniques are subject to investigation in current research on *paraconsistent reasoning*, as mentioned in Sect. 3.2.6.

There is an extra chapter dedicated to the topic of semantic annotation, namely Chap. 5, in which the usage of ontologies for annotating web content with meta data in the Semantic Web context is further elaborated on.

3.4 Ontology Languages

To make ontologies available to information systems, various concrete ontology languages have been designed and proposed for standardisation. In this section, we give an overview of the most prevalent ontology languages that are important in the context of the Semantic Web, and present some of them in detail.

3.4.1 Hierarchy of Languages for the Semantic Web

In the light of widespread impact and industrial usability, the standardisation of ontology languages is of great importance to the Semantic Web community. Various different aspects are considered for language standardisation, such as issues of the underlying knowledge representation formalism in terms of expressiveness and computational properties, web-related features like global unique identification and XML serialisation syntax, or usability add-ons like the inclusion of strings and numbers or non-functional meta data. The influence of different research and user communities with manifold requirements have resulted in a complex landscape of a multitude of

languages backed by different past and ongoing standardisation efforts. Which languages are best suited for what purpose, how they can be efficiently implemented and realised in a user-friendly way, or technically and semantically made interoperable is still an open topic stimulating lively discussions in current research.

In Fig. 3.4 we make an attempt to sketch this landscape of languages, giving an overview of the most important ontology languages with respect to current trends in the Semantic Web. Since some languages build on others and on formerly achieved standards, this landscape can be perceived as a hierarchy of languages for the Semantic Web. However, besides a hierarchical structure with some languages being clearly layered on top of others, there are also parallel branches and cross-relations between languages and formalisms.[13]

One of the major distinctions of Semantic Web languages is by the knowledge representation paradigm they follow. On the left-hand side in Fig. 3.4 there is the description logic family of languages that build on various DL dialects and their rule-extensions. They adhere to the classical model-theoretic semantics of first-order predicate logic and to the open-world assumption. On the right-hand side there is the family of logic programming languages that build on rules with

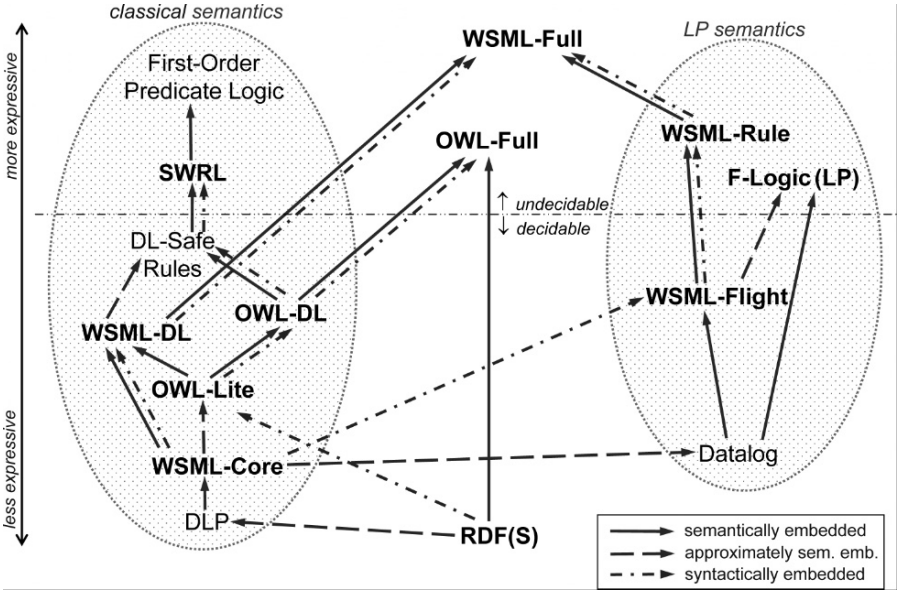


Fig. 3.4. An overview of Semantic Web languages

[13] This figure shall convey a rough intuition about the relationships between major languages with respect to their underlying knowledge representation formalisms and paradigms. It therefore abstracts from certain language details and is necessarily imprecise and vague in some aspects

negation-as-failure. They typically follow a semantics of minimal or preferred models and adhere to the closed-world assumption. There are also languages in between these two main strands, which cannot be clearly assigned to either paradigm. These have been designed with a focus set on aspects other than a logically clear semantics, or are attempts to combine features from both worlds, while the pure DL and LP family languages have well understood properties in terms of computability and inferential behaviour.

Languages that are placed near to the top in Fig. 3.4 are more expressive than languages that are placed close to the bottom, meaning that they allow for expressing more complex knowledge and for richer inferencing through more sophisticated logical consequences than less expressive languages do. Accordingly, high expressivity of a language is traded for higher computational complexity of decision procedures for reasoning. Within recent standardisation efforts, it is considered highly desirable to at least maintain decidability as a design goal for a Semantic Web ontology language, and Fig. 3.4 shows a boundary for decidability, above of which languages do not meet this goal.

Three different kinds of arrows in Fig. 3.4 express a relationship of embedment between languages. A solid arrow denotes complete semantic containedness of a less expressive language in a more expressive one, meaning that anything that can be expressed in the former can also be expressed in the latter by means of a direct mapping of languages constructs. A dashed arrow denotes a weaker form of embedding, where not all the features of the less expressive language do completely fit the more expressive target language, meaning that the former is in principle (approximately) covered by the latter, apart from moderate deficiencies in some language constructs and their semantic interpretation. A dash-dotted arrow denotes a syntactic embedding such that the language constructs of the (syntactically) less expressive language can be directly used in the more expressive one, although they may semantically be interpreted in a different way.

An early initiative to standardise a language for semantic annotation of web resources by the World Wide Web consortium (W3C) resulted in *RDF* and *RDFS*, which form now a well established and widely accepted standard for encoding meta data. The *RDF(S)* language is described in more detail in Sect. 3.4.2. It can be used to express class-membership of resources and subsumption between classes but its peculiar semantics does fit neither the classical nor the LP-style. If semantically restricted to a first-order setting, RDF(S) can be mapped to a formalism named description logic programs (*DLP*) [18], which is sometimes used to interoperate between DL and LP by reducing expressiveness to their intersection.

On top of RDF(S), W3C standardisation efforts have produced the *OWL* family of languages for describing ontologies in the Semantic Web, which comes in several flavours with increasing expressiveness. Only the most expressive language variant, namely *OWL-Full,* has a semantically proper layering on top of RDF(S), allowing for features of metamodelling and reification. The less expressive variants *OWL-Lite* and *OWL-DL* map to certain description logic dialects and fit the classical semantics as subsets of *first-order logic*. Besides the class membership and subsumption relations inherited from RDF(S), OWL offers the construction of complex classes from

simpler ones by means of DL-style concepts constructors. Among ongoing standardisation efforts, OWL-DL is currently the most prominent Semantic Web ontology language following the description logic paradigm, and in Sect. 3.4.3 the OWL family is described in more detail.

A current trend in research on knowledge representation formalisms in the context of the Semantic Web is to integrate DL-style ontologies with LP-style rules to be interoperable on a semantic level. One attempt to do so is the Semantic Web Rule Language *SWRL*)[14] that extends the set of OWL axioms to include Horn-like rules interpreted under first-order semantics. Interoperability with OWL ontologies is realised by referring to OWL classes and properties within SWRL rules; however, the combination of OWL-DL and SWRL rules results in an undecidable formalism. Another approach to amalgamate OWL ontologies and rules are the so-called *DL-safe rules* [38], which extend DL knowledge bases in a way similar to SWRL. However, DL-safe rules preserve decidability of the resulting language by imposing an additional safety restriction on SWRL rules which ensures that they are only applied to individuals explicitly known to the knowledge base.

Languages that follow the logic programming paradigm mainly stem from deductive database systems, which apply rules on the facts stored in a database to derive new facts by means of logical inferencing. A common declarative language used in deductive databases is *Datalog* [47], which is syntactically similar to *Prolog* [31]. In the Semantic Web context, *F-Logic* is a more prominent rule language that combines logical formulas with object-oriented and frame-based description features. In its logic programming variant *F-Logic (LP)*, it adopts the semantics of Datalog rules. In Sect. 3.4.4 we investigate F-Logic in more detail.

Finally, the Web Service Modeling Language (*WSML*) family is the most recent attempt to standardise ontology languages for the web, with a special focus on annotating Semantic Web Services. Since WSML tries to cover all the major aspects of different knowledge representation formalisms, its various language variants are spread over the scheme of Fig. 3.4. They fit semantically in between existing languages by being based on similar formalisms in both the DL and the LP strands. We will have a closer look at the WSML family of languages in Sect. 3.4.5.

3.4.2 RDF(S)

The Resource Description Framework (RDF) [30] is a language recommended by the W3C standardisation body for representing information about resources in the World Wide Web. It is particularly intended for the representation of meta data about identifiable web resources, such as title and author of a web page, topic and copyright information of an electronic document retrievable from the web or functionality and access conditions of a Web Service.

Abstracting from retrievable or electronically processable web resources to anything that has identity, RDF can be used to represent information about just anything.

[14] `http://www.w3.org/Submission/SWRL/`

In this sense, RDF can serve as a language to represent knowledge as meta data about entities in, e.g., the business trips domain.

The RDF Vocabulary Description language RDF Schema (RDFS) [7] is an extension to RDF which facilitates the formulation of vocabularies for RDF meta data. While RDF is used to relate resources by means of properties, RDFS introduces the notions of resource classes and their hierarchies. The combined use of both RDF and RDFS is often referred to as RDF(S) and provides a simple ontology language for conceptual modelling with some basic inferencing capabilities.

Basic Elements of RDF

The approach for representing meta data about resources in RDF is based on a few main ideas.

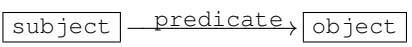
Identity through URIs

Uniform Resource Identifiers (URIs) are used for naming entities. They exhibit some naming conventions that allow for partitioning of names into namespaces. For modelling ontologies in RDF, URIs may be used to identify the following kinds of entities: individuals, such as the person MisterX or the company UbiqBiz; kinds of things, such as Employee or Company; properties of those things, such as mailbox; and values of those properties, such as the string "`mailto:mrX@ubiqbiz.com`".

By URIs, resources are uniquely identified throughout the web, which allows for a decentralised organisation of knowledge about commonly referenced resources.

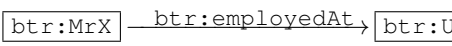
Sentences with Subject, Predicate and Object

Statements in RDF have the form of subject–predicate–object sentences, which are also referred to as RDF *triples*. A triple



relates a *subject* to an *object* via a *predicate*, while the roles of subject, predicate and object are played by resources identified by URIs. The subject is the resource to be described, the predicate is a specific property of this resource, and the object serves as a value of this property for this resource.

Examples for triples in RDF are[15]



,

stating that MisterX is employed at UbiqBiz, or

,

stating that MisterX is the author of his web page at the UbiqBiz website.

[15] In the examples, `btr:` refers to a namespace abbreviation for the business trips domain

Graph Representation

Several triples taken together form an RDF *graph*, whose nodes are resource URIs and whose arcs are properties. A node in an object position can be either a resources or an RDF *literal*, which represents a data value like the string "`mailto:mrX@ubiqbiz.com`" or some number. Furthermore, RDF graphs support *blank nodes*, which represent anonymous resources. From a knowledge representation view, an RDF graph can be seen as a semantic network, similar to the one depicted in Fig. 3.1.

Since RDF is a web language, the various triples in an RDF graph can originate from different sites, with the idea that anybody can state anything about any resource. In this sense, RDF is designed to capture knowledge and meta data that is spread over the web.

XML Serialisation

Another web-related aspect of RDF is its XML serialisation format in which RDF graphs are encoded for machine processing and for transport over the wire. An example of the above triples encoded in RDF/XML syntax is the following.

```
<rdf:Description rdf:about="http://ubiqbiz.com/web/MrX.html">
     <btr:hasAuthor rdf:resource="btr:MisterX"/>
</rdf:Description>
<rdf:Description rdf:about="btr:MisterX">
     <btr:employedAt rdf:resource="btr:UbiqBiz"/>
</rdf:Description>
```

Descriptions of resources are encoded using special XML tags from the RDF-predefined vocabulary.

Reification

RDF allows one to make statements about statements, which is referred to as reification. A reified statement is a resource that represents an occurrence of an RDF triple. In this way, meta statements can be formulated, which can be illustrated as follows.

Here, the subject role is played by a resource that represents a whole statement.

Reification is particularly interesting in the context of the Semantic Web, where it can be used to make statements about things that have been stated elsewhere by referring them as resources.

Data Structuring Facilities

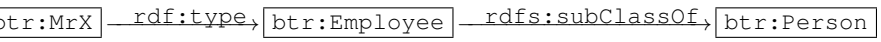
Furthermore, RDF specifies elements to represent basic data structures as known from programming languages, namely *containers* and *collections*. Containers can be used to realise open data structures, such as ordered and unordered sequences, whereas collections allow for list structures that can be closed by stating that there are no more members.

Typing Resources with RDFS

RDFS facilitates the specification of application-specific ontological vocabularies in the form of class and property hierarchies on top of RDF resources. For this purpose, it defines a set of reserved keywords that can be used in RDF triples to relate resources to classes.

Classes

RDFS defines a type system for RDF resources by introducing the concept of a *class*. The reserved predicate `rdf:type` is used to indicate class membership, i.e. that a resource is of a certain type. RDFS classes are organised in a hierarchy of types for RDF resources. The reserved predicate `rdfs:subClassOf` is used to state a subclass relationship between two types. The following RDF(S) graph illustrates the typing of resources.

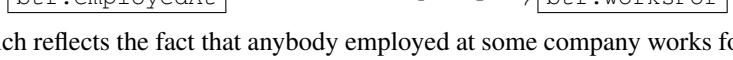


Here, the resource that represents MisterX is stated to be of type `btr:Employee`, i.e. MisterX is a member of the class of employees, which is itself a subclass of persons.

These RDF(S) constructs for typing allow for the formulation of subsumption hierarchies and for the distinction between instances and concepts in the ontological sense. However, in RDF(S) there is no clear separation between classes and their members. Instead, RDF(S) allows self-reference and classes being members of (meta) classes. Any resource can be tagged as a class by relating it to the predefined meta type `rdfs:Class`.

Properties

By the semantics of RDF(S), any resource used in the predicate position of an RDF triple is a member of the class `rdfs:Property`. Besides classes, properties can also be organised in a hierarchy by means of the keyword `rdfs:subPropertyOf`. An example is the following triple,



which reflects the fact that anybody employed at some company works for this company.

With the predefined predicates `rdfs:domain` and `rdfs:range`, one can define the domain and range for a property. By setting the range of the property `btr:employedAt` in the above example to `btr:Company`, any resource that fills the object position of an RDF triple with this property as predicate is a member of the company class.

Semantics of RDF(S)

RDF(S) comes with a formal semantics that is specified in a model-theoretic way in [24]. Here, we only sketch the basic ideas of the semantics defined there, giving an intuition on the inferencing characteristics of RDF(S).

In logical terms, RDF is an assertional language in which each triple expresses a positive ground proposition. An RDF graph, as a set of triples, makes up a logical theory that consists of positive ground assertions. Since there is no concept of negation, one cannot express contradictory information in the language. Although it is possible to express or infer that, e.g., a person is both male and female, there is no way of stating that the classes of males and females cannot have common resources as their members.

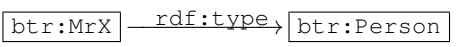
In [24], the semantics of RDF(S) is characterised in the form of axiomatic triples and entailment rules that derive new, inferred triples. To yield the set of all entailed statements for an RDF graph G_{RDF}, the rules are exhaustively applied to the triples of G_{RDF} together with all axiomatic triples. In this sense, the RDF(S) semantics determines which implicit knowledge is derived from explicitly stated assertions in a graph. To illustrate the most essential parts of the RDF(S) semantics, we give examples of some of these entailment rules and their application to triples.

For example, the semantics for class membership and inheritance is determined by the following two entailment rules applied to the triples of an RDF graph G_{RDF}.

(1) IF G_{RDF} contains $(\text{C}, \texttt{rdfs:subClassOf}, \text{D})$ and $(\text{R}, \texttt{rdf:type}, \text{C})$
THEN derive $(\text{R}, \texttt{rdf:type}, \text{D})$

(2) IF G_{RDF} contains $(\text{C}, \texttt{rdfs:subClassOf}, \text{D})$ and $(\text{D}, \texttt{rdfs:subClassOf}, \text{E})$
THEN derive $(\text{C}, \texttt{rdfs:subClassOf}, \text{E})$

Their reading is to derive the triple in the THEN part for any instantiation of triples in the IF-part. The variables occurring inside the triples range over RDF resource URIs. Rule (1) entails the membership of resources in superclasses, while rule (2) ensures the transitivity of the subclass relationship. From the previous triple about MisterX being an employee as a special kind of person, rule (1) would entail the following triple.



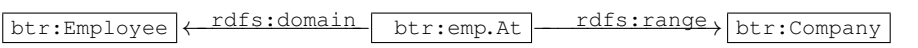
Thus, an implementation of an RDF system would include MisterX in the result for the query asking for all persons.

As another example, the semantics for domains and ranges of properties is determined by the following two entailment rules.

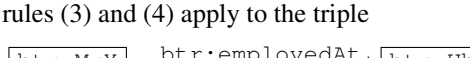
(3) IF G_{RDF} contains $(\text{P}, \texttt{rdfs:domain}, \text{C})$ and $(\text{R}, \text{P}, \text{S})$
THEN derive $(\text{R}, \texttt{rdf:type}, \text{C})$

(4) IF G_{RDF} contains $(\text{P}, \texttt{rdfs:range}, \text{C})$ and $(\text{R}, \text{P}, \text{S})$
THEN derive $(\text{S}, \texttt{rdf:type}, \text{C})$

By setting the domain and range of the property `btr:employedAt` to `btr:Employee` and `btr:Company`, as follows,



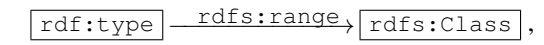
the rules (3) and (4) apply to the triple



,

deriving that MisterX is an employee and that UbiqBiz is a company.

The entailment rules also apply to the RDF(S) meta vocabulary, determining the relationship between predefined vocabulary resources like `rdfs:Class` or `rdfs:Property`. For example, the axiomatic triple



already triggers rule (4) for any class membership assertion, deriving that the referred type resource is a class.

Software Support for RDF(S)

The RDF(S) language is used by various web-based applications for describing meta data, and a number of tools are available that support visual editing and programmatic handling of RDF(S) descriptions.

One of the most common visual editors for RDF(S) is Protégé,[16] although recently its focus has been shifted towards OWL. Protégé allows to navigate and edit an RDF(S) class hierarchy and has special support for populating an RDF Schema with instances using customisable input forms. Other ontology editors that support RDF(S) are WebODE[17] [10], OntoEdit[18] [46] and KAON[19] OI-Modeller.

For in-memory processing and database storage of RDF(S) descriptions, common tool suites are Sesame[20] [8] and Jena[21] [32], which provide software libraries that enable software developers to process RDF(S) descriptions within their applications. They comprise parsing and serialisation for the RDF XML format, an in-memory object representation for RDF(S) descriptions as well as database persistency and querying functionality including reasoning capabilities. Recently, also oracle include RDF(S) support in their database solutions. [22]

3.4.3 OWL

The Web Ontology Language (OWL) [40] has been standardised by the W3C consortium as a language for semantic annotation of web content and is widely accepted within the Semantic Web community.

An important issue for the design of OWL was the trade-off between expressivity of the language on the one hand and scalability of reasoning on the other. To this end, OWL comes in three different flavours, namely OWL-Lite, OWL-DL and OWL-Full, reflecting different degrees of expressiveness. The design of OWL-Lite and OWL-DL has been significantly influenced by descriptions logics, and hence these two

[16] `http://protege.stanford.edu/`
[17] `http://webode.dia.fi.upm.es/WebODEWeb/index.html`
[18] Meanwhile OntoStudio – `http://ontoedit.com/`
[19] `http://sourceforge.net/projects/kaon`
[20] `http://sourceforge.net/projects/sesame/`
[21] `http://jena.sourceforge.net/`
[22] See the technical whitepaper at `http://www.oracle.com/technology/tech/semantic_technologies/pdf/semantic_tech_rdf_wp.pdf`

variants correspond to the description logic dialects $\mathcal{SHIF}(\mathbf{D})$[23] and $\mathcal{SHOIN}(\mathbf{D})$, respectively. OWL-Full, on the contrary, departs from description logic semantics in order to provide compatibility with RDF(S). The DL-based OWL variants benefit from well understood computational properties and decidability of description logic, while OWL-Full has shown to be undecidable [36]. In our presentation of OWL, we focus on OWL-DL as the most prominent language variant with the most support by the Semantic Web community.

Syntax and Intuitive Semantics

The OWL standard defines different syntaxes based on RDF(S), XML and proprietary text format. The *OWL RDF/XML syntax* allows for an encoding of an OWL ontology within the RDF(S) framework in RDF/XML serialisation. The *OWL XML presentation syntax* provides a more compact XML format for OWL ontologies, independent from RDF(S). In contrast to these machine-oriented serialisations, the *OWL abstract syntax* serves as a human readable text format to present OWL ontologies to knowledge engineers. Yet another popular way to present OWL content to a reader in a more scientific context is to make use of DL formulas. We choose to present examples in OWL abstract syntax as well as in the more compact description logic formal notation.

Similar to RDF(S), OWL provides syntactic modelling constructs for the basic elements of an ontology, i.e. concepts, relations and instances. In OWL these are called *classes*, *properties* and *individuals*, respectively, and they correspond to concepts, roles and individuals in description logics. In contrast to RDF(S), OWL-DL strictly separates classes from individuals and allows for building complex classes out of simpler ones by means of class *constructors*. In the following we go over a selection of the syntactic elements of OWL including various such constructors. For each example statement, taken from the geographic ontology depicted in Fig. 3.2, we give its intuitive meaning in natural language as well as notations in OWL abstract syntax and DL formulas.

OWL by Examples

Named classes are usually introduced by means of class declarations that correspond to DL inclusion axioms with an atomic concept on the left-hand side, as in the following example.

① "*A continent is a geographic location different from a country.*"	

`Class(`*Continent* `partial`
`intersectionOf(`*GeographicLocation*
`complementOf(`*Country*`))`
	Continent $\sqsubseteq$ *GeographicLocation* $\sqcap \neg$*Country*

Here the class *Continent* is introduced through a *partial* declaration, which specifies (some of) its necessary conditions. By means of the constructors `intersectionOf`

[23] The $\mathcal{F}$ stands for functional roles, i.e. it can be stated that role relationships must be functional

and `complementOf`, a continent is declared to be a geographic region but not a country. Hence, this syntactic construct states both subclass relationship and disjointness, according to the respective DL inclusion axiom. "Necessary" here means that any continent is also a geographic location and not a country. However, not any geographic location that is not also a country is necessarily a continent; the partial class declaration only works in one direction and does not impose a "sufficient" condition, which can be achieved by using the keyword `complete` instead of `partial`. The keyword `complete` specifies class equivalence.

Individuals are introduced based on class descriptions, as in the following example.

② "*Europe is a particular continent.*"	
`Individual(`*Europe* `type(`*Continent*`))`	*Continent*(*Europe*)

Here the individual *Europe* is introduced as an instance of the class *Continent*. Although this example shows the instantiation of a previously declared named class, the class description for the `type`-clause can be arbitrarily complex using class constructors.

An alternative way to define a class is to enumerate all its individuals, as shown in the following example.

③ "*The continents are America, Europe, Africa, Asia and Australia.*"	
`EnumeratedClass(`*Continent* *America Europe Africa Asia Australia*`)`	*Continent* $\equiv \{$*America*, *Europe*, *Africa*, *Asia*, *Australia*$\}$

Here the class *Continent* is defined by listing all its known members, i.e. all the different continents.

Similar to classes, properties are introduced through explicit declarations with optional domain and range classes and other modifiers, as shown in the following example.

④ "*Geographic regions in general contain geographic regions.*"	

`ObjectProperty(`*contains*
`domain(`*GeographicRegion*`)`
`range(`*GeographicRegion*`)`
`inverseOf(`*isContainedBy*`)`
`Transitive)`
	$\exists$ *contains*.$\top \sqsubseteq$ *GeographicRegion*,
	$\top \sqsubseteq \forall$ *contains*.*GeographicRegion*,
	locatedIn $\equiv$ *isContainedBy*$^-$
	Trans(*contains*)

Here the object property *contains* is declared as a transitive containment relation between geographic regions. It is linked to its inverse property *isContainedBy*. The `domain` and `range` clauses are mapped to appropriate DL inclusion axioms: anything that contains something is a geographic region, as well as anything that is being contained. In addition to the domain of individuals OWL also offers the so-called *concrete domains* [3], i.e. properties can alternatively range over datatypes such as integer, float or string.

Once properties have been introduced, complex class descriptions can be formed by imposing restrictions on them. The following example shows a general subclass statement including a restriction on the previously introduced property.

⑤ "*A planar region only contains planar or linear regions.*"	
`SubClassOf(`*PlanarRegion* `  restriction(`*contains* `allValuesFrom(` `  unionOf(`*PlanarRegion* *LinearRegion*`)))`	$\mathit{PlanarRegion} \sqsubseteq \forall\, \mathit{contains}.(\mathit{PlanarRegion} \sqcup \mathit{LinearRegion})$

Here planar regions are restricted to only contain planar or linear regions by means of the `restriction` constructor. The `allValuesFrom` clause requires that all values for the restricted property are of a certain type, which is specified as a disjunction by means of the `unionOf` constructor. Although this example states subclass relationship for a named class, both parameters of the `subClassOf`-clause can be arbitrarily complex class descriptions made up of constructors.

Statements of class equivalence can also be quite sophisticated as in the following example.

⑥ "*A European city is a city whose geographic region is contained in that of Europe.*"	

`EquivalentClasses(`*EuropeanCity*
`intersectionOf(`
`  `*City*
`  restriction(`*locatedIn*
`allValuesFrom(`
`  restriction(`*isContainedBy*
`someValuesFrom(`
`  restriction `*isRegionFor*
`someValuesFrom(`
`  oneOf(`*Europe*`)))))))`
	$\mathit{EuropeanCity} \equiv \mathit{City} \sqcap$
	$\forall\, \mathit{locatedIn}.\exists\, \mathit{isContainedBy}.\exists\, \mathit{isRegionFor}.\{\mathit{Europe}\}$

Here the class *EuropeanCity* is set equivalent to a complex class description with nested restrictions on properties and their inverses. By this, a city can be concluded to be European if its geographic region is contained by that of the European continent. The `someValuesFrom` clause restricts a property such that there must exist a value of a certain type, while the `oneOf` constructor creates a class from an explicitly named individual, similar to the enumerated class in ③.

Another way to restrict properties is to constrain their cardinality, as shown in the following example.

⑦ "*A city is a geographic location governed by a single country.*"	
`SubClassOf(`*City* `  restriction(`*governedBy* `maxCardinality 1))`	$\mathit{City} \sqsubseteq \leq 1\ \mathit{governedBy}$

Here cities are restricted to be governed by at most one country by means of the `maxCardinality` clause. Similarly, minimal cardinality can be realised with the `minCardinality` clause, while both can be combined to require a fixed cardinality.

Another usage of introduced properties is to connect individuals to other individuals or data values, as shown in the following example.

⑧ "*Munich is a German city with 1288307 inhabitants.*"	

`Individual(`*Munich* `type(`*City*`)`
`  value(`*governedBy* *Germany*`)`
`  value(`*numberOfInhabitants* `1288307))`
	$\mathit{City}(\mathit{Munich})$,
	$\mathit{governedBy}(\mathit{Munich}, \mathit{Germany})$
	$\mathit{numberOfInhabitants}(\mathit{Munich}, 1288307)$

Here the individual Munich is stated to be a city that lies in Germany by an appropriate connection to the individual *Germany*. It is asserted an integer value for the property *numberOfInhabitants*.

Model-Theoretic Semantics

The exact semantics of the DL-based OWL variants is determined by the model-theoretic semantics of the underlying description logic formalism. An OWL ontology consists of a collection of statements as the ones shown in the examples ① – ⑧. These statements are interpreted as axioms of a DL knowledge base, as described in Sect. 3.2, and thus OWL employs the open-world assumption. Table 3.1 shows the mapping of OWL abstract syntax constructs to their corresponding description logic axioms.

Table 3.1. Translation of OWL abstract syntax to description logic formal notation

OWL abstract syntax	**DL syntax**
Axioms	
`Class(`A `partial` $C_1 \ldots C_n$`)`	$A \sqsubseteq C_1 \sqcap \ldots C_n$
`Class(`A `complete` $C_1 \ldots C_n$`)`	$A \equiv C_1 \sqcap \ldots C_n$
`EnumeratedClass(`$A\ a_1 \ldots a_n$`)`	$A \equiv \{a_1\} \sqcup \cdots \sqcup \{a_n\}$
`SubClassOf(`$C\ D$`)`	$C \sqsubseteq D$
`EquivalentClasses(`$C_1 \ldots C_n$`)`	$C_1 \equiv \cdots \equiv C_n$
`DisjointClasses(`$C_1 \ldots C_n$`)`	$C_i \sqsubseteq \neg C_j$, $(1 \leq i < j \leq n)$
`ObjectProperty(`r `super(`r_1`)...super(`r_n`)`	$r \sqsubseteq r_1 \sqcap \cdots \sqcap r_n$
`domain(`C_1`) ... domain(`C_n`)`	$\exists\, r.\top \sqsubseteq C_1 \sqcap \cdots \sqcap C_n$
`range(`C_1`) ... range(`C_n`)`	$\top \sqsubseteq \forall\, r.C_1 \sqcap \cdots \sqcap \forall\, r.C_n$
`[inverseOf(`s`)]`	$r \equiv s^-$
`[Symmetric]`	$r \equiv r^-$
`[Functional]`	$\top \sqsubseteq \leq 1\, r$
`[InverseFunctional]`	$\top \sqsubseteq \leq 1\, r^-$
`[Transitive])`	$\mathsf{Trans}(r)$
`SubPropertyOf(`$r\ s$`)`	$r \sqsubseteq s$
`EquivalentProperties(`$r_1 \ldots r_n$`)`	$r_1 \equiv \cdots \equiv r_n$
`Individual(`a `type(`C_1`) ... type(`C_n`)`	$C_1 \sqcap \cdots \sqcap C_n(a)$
`value(`$r_1\ a_1$`) ... value(`$r_n\ a_n$`))`	$r_1(a, a_1), \ldots, r_n(a, a_n)$
`SameIndividual(`$a_1 \ldots a_n$`)`	$a_1 = \cdots = a_n$
`DifferentIndividuals(`$a_1 \ldots a_n$`)`	$a_i \neq a_j$, $(1 \leq i < j \leq n)$
Descriptions	
`Class(`A`)`	A
`Class(owl:Thing)`	$\top$
`Class(owl:Nothing)`	$\bot$
`intersectionOf(`$C_1\ C_2\ \ldots$`)`	$C_1 \sqcap C_2$
`unionOf(`$C_1\ C_2\ \ldots$`)`	$C_1 \sqcup C_2$
`complementOf(`C`)`	$\neg C$
`oneOf(`$a_1\ a_2\ \ldots$`)`	$\{a_1\} \sqcup \{a_2\}$
`restriction(`r `someValuesFrom(`C`))`	$\exists\, r.C$
`restriction(`r `allValuesFrom(`C`))`	$\forall\, r.C$
`restriction(`r `hasValue(`a`))`	$\exists\, r.\{a\}$
`restriction(`r `minCardinality(`n`))`	$\geq n\, r$
`restriction(`r `maxCardinality(`n`))`	$\leq n\, r$

Working with OWL Ontologies

Due to the connection of OWL to description logics, the basic reasoning services available for DL knowledge bases also apply to OWL ontologies. Thus, an OWL ontology can be checked for consistency or it can be queried for implicit knowledge.

Ontology Inconsistency

Consider the following OWL ontology consisting of three statements.

```
{ subClassOf(City restriction(governedBy maxCardinality(1))),
  Individual(Nicosia type(City) value(governedBy Greece) value(governedBy Turkey)),
  DifferentIndividuals(Greece Turkey) }
```

The first statement is taken from ⑦ and says that cities are uniquely governed by a single country. The second statement says that the city of Nicosia[24] is governed by both Greece and Turkey, while the third statement assures that these are two different countries. This is clearly a contradiction and this OWL ontology is therefore *inconsistent*. This can be verified by using the reasoning service of *knowledge base satisfiability*, offered by common description logic reasoners. Notice that for practical reasons an inconsistent ontology is quite useless, since it allows to conclude any arbitrary statement.

Ontology Coherency

Another kind of "problematic modelling" in ontologies is to introduce classes that cannot have instances, which is the case in the following OWL ontology.

```
{ subClassOf(City restriction(governedBy maxCardinality(1))),
  class(SplitCity complete
       intersectionOf(City restriction(governedBy minCardinality(2)))) }
```

Again, the first statement, taken from ⑦, restricts cities to be governed by at most one country. The second statement introduces a class *SplitCity*, requiring that split cities are cities governed by at least two countries. However, by the first statement, this is not possible and thus the class *SplitCity* cannot have an instance in any valid model of the corresponding description logic knowledge base. In DL-terms this means that the concept *SplitCity* is unsatisfiable. Common description logic reasoners offer the service of checking concepts for their satisfiability. An ontology that contains an unsatisfiable concept/class is said to be *incoherent*. In contrast to inconsistent ontologies, an incoherent ontology is not useless and many reasoning tasks might not be affected by the unsatisfiability of a particular class. However, incoherence of an ontology indicates erroneous modelling, and once an unsatisfiable class is assigned an individual as an instance the ontology becomes inconsistent.

[24] Nicosia is the capital of Cyprus and is split into a Greek and a Turkish part

Querying for Subsumption

Besides checking an ontology for consistency or coherency, its main usage is to be queried for implicit knowledge. Based on the notion of entailment, for any OWL statement we can ask whether it follows from an OWL ontology, i.e. whether its corresponding DL axiom is entailed by the respective DL knowledge base. Querying for subsumption between two classes underlies the most important usage of reasoning in the OWL language, namely *classification*. The following OWL ontology allows for the automatic classification of two classes that are not explicitly put in subsumption relation.

```
{ class(SplitCity complete
      intersectionOf(City restriction(governedBy minCardinality(2)))),
  class(GreekTurkishCity partial
      intersectionOf(City
                     restriction(governedBy someValuesFrom(oneOf(Greece)))
                     restriction(governedBy someValuesFrom(oneOf(Turkey))))),
  DifferentIndividuals(Greece Turkey)                                    }
```

The first statement introduces split cities as before, while the second statement introduces a class *GreekTurkishCity* for cities which are governed by both Greece and Turkey. The third statement assures the two involved countries to be distinct, as before. Notice that this time the ontology does not restrict cities to be governed by a single country. From the knowledge specified in the ontology, *GreekTurkishCity* is a subclass of *SplitCity* and a DL reasoner would derive the statement `subClassOf(GreekTurkishCity SplitCity)` as a logical consequence.

By checking subsumption between all the named classes in an OWL ontology, an inferred class hierarchy can be established.

Querying for Assertion

The other kind of statements an OWL ontology can be queried for are assertion axioms. For both role assertions and concept assertions, we can ask whether they hold with respect to an OWL ontology, as illustrated by the following example.

```
{ subClassOf(EUCountry restriction(officialCurrency hasValue(Euro))),
  Individual(Germany type(EUCountry),
  class(GermanCity partial
      intersectionOf(City restriction(governedBy hasValue(Germany)))),
  Individual(Munich type(GermanCity)                                  }
```

This ontology states that in countries in the EU, as e.g. Germany, the official currency is Euro, and that German cities, as e.g. Munich, are cities governed by Germany. From the knowledge specified in the ontology, it follows that *Munich* is governed by *Germany*, and a DL reasoner would derive the statement `Individual(Munich value(governedBy Germany))` as a logical consequence, since *Munich* is assigned to be a *GermanCity*. Furthermore, the ontology allows to conclude that in Munich one can pay with Euro, i.e. Munich is governed by a country that has Euro as official currency. A reasoner would derive the statement `Individual(Munich type(restriction(governedBy`

someValuesFrom(restriction(*officialCurrency* hasValue(*Euro*)))))), since, as a *GermanyCity*, *Munich* is governed by *Germany* whose official currency is *Euro*.

By iterating over all the individuals in an OWL ontology, querying for subsets of named individuals with certain properties can be achieved. For example, in the above query *Munich* can be subsequently replaced by other named individuals to retrieve all cities in which one can pay with Euro.

Software Support for OWL

Since OWL is technically built on top of RDF(S), some RDF(S) specific tools can be readily applied, e.g. for parsing and serialisation in the OWL RDF/XML format, while others have also been upgraded to OWL versions.

The ontology editor Protégé [22] also supports OWL and comes with a variety of plugins that allow for visualisation and management of OWL ontologies. In addition to different graphical views of the explicit class and property hierarchies, it facilitates the visual editing of OWL axioms and enables the embedding of reasoning tools for computing inferred subsumption hierarchies. Other visual editors for OWL ontologies that offer similar functionality are SWOOP[25] [28] or the commercial tools Altova Semantic Works[26] and TopBraid.[27]

For the programmatic handling of OWL ontologies, the OWL API[28] [5] as well as Jena [32] can be used by software developers to process OWL descriptions within their applications. They provide means for parsing and serialisation of the different OWL syntax formats and for in-memory manipulation of ontologies.

As OWL is an expressive knowledge representation language, reasoning plays an important role, and there are a number of description logic reasoners available that can be used for querying OWL ontologies with respect to inferred knowledge or for verifying their consistency. The most common description logic reasoners in the Semantic Web context are based on the tableau calculus, and available systems that support the OWL language are Racer[29] [23], FaCT[30] [25] and Pellet[31] [44]. Recently, new DL reasoning algorithms – based on deductive database technology – were devised for the development of the KAON2[32] [37] system, which is particularly optimised for querying ontologies with large A-Boxes.

3.4.4 F-Logic

Frame Logic (F-Logic) [29] is a deductive, object-oriented database language which aims at combining the declarative semantics and expressiveness of logic program-

[25] http://www.mindswap.org/2004/SWOOP/
[26] http://origin.altova.com/products_semanticworks.html
[27] http://www.topbraidcomposer.com/
[28] http://owl.man.ac.uk/api.shtml
[29] Meanwhile RacerPro – http://www.racer-systems.com/
[30] Meanwhile FaCT++ – http://owl.man.ac.uk/factplusplus/
[31] http://www.mindswap.org/2003/pellet/
[32] http://kaon2.semanticweb.org/

ming with rich and intuitive conceptual modelling capabilities, as provided by frame-based systems. The most significant language features of F-Logic comprise object identity, complex objects, classes, inheritance, polymorphic types, rules and queries. Besides the aspects of a frame-based language for conceptual modelling, it can also be perceived as a logic with model-theoretic semantics and a sound and complete resolution-based proof theory.

We give a short overview on syntax and informal semantics of the most important features of F-Logic. In the original specification [29], F-Logic is given several semantics and in its full version it is an extension of first-order logic. However, systems that support the language do not implement full F-Logic but a logic programming variant based on the well-founded semantics. Thus, we present F-Logic as a rule-based LP-style language, as it is widely perceived.

F-Logic by Examples

Frame-Based Modelling

F-Logic allows to describe *objects* – identified by an object ID – by grouping related information about the object in the so-called *F-molecules*. The following example illustrates the use of F-molecules to describe some objects from our business trips scenario.

```
UbiqBiz[hasLegalName   ->  'Ubiquitous Business Ltd.',
        hasOfficesIn   ->> {NewYork, London, Singapore},
        hasPhones      ->> {0017324747123, 00654564458},
        hasEmployees   ->> {MrX, MrY, MsZ}].

MrX[hasName       -> 'Mister X',
    hasAddress    -> AddressMrX[hasStreet -> 'Fifth Avenue',
                                hasNumber -> 521,
                                hasCity   -> NewYork].

BookingUbiqMrX[bookedBy    ->   UbiqBiz,
               bookedFor   ->   MrX,
               issuedFor   ->   FL4711].
```

In the example, objects, such as `UbiqBiz`, are described in terms of F-molecules that assign them values for certain attributes, such as legal name, locations of offices, phone numbers and associated employees. As values for attributes, F-Logic allows objects as well as data values, such as strings or numbers. The symbol `->` denotes an assignment of a single value, while the symbol `->>` indicates the assignment of multiple values for set-valued attributes. As illustrated by the attribute `hasAddress`, attribute assignments in F-molecules can be nested.

From an ontology point of view, the objects in the example can be seen as instances. Besides these, F-Logic also provides language features for describing *classes* of objects with attached *attributes* and relating them in class hierarchies, as shown next.

```
Company :: LegalEntity.
Company[hasLegalName   =>  STRING,
        hasOfficesIn   =>> City,
        hasPhones      =>> NUMBER,
```

```
          hasEmployees   =>> Person].

Person :: LegalEntity.
Person[hasName       => STRING,
       hasAddress    => Addresss].

Employee :: Person.
Employee[isEmployedAt => Company].

Booking[bookedBy   =>   LegalEntity,
        bookedFor  =>   Person,
        issuedFor  =>   Flight].

UbiqBiz : Company.
MrX : Person.
FL4711 : Flight.
BookingUbiqMrX : Booking.
```

In the example, the object `Company` is described as a class for company objects with appropriate attribute ranges. The symbol `=>` indicates a single-valued range, while the symbol `=>>` assigns a set-valued range for attributes with multiple values.

Both `Company` and `Person` are declared as subclasses of `LegalEntity` by means of the symbol `::`, which denotes class inheritance and is used to build class hierarchies. The class `Employee` is, in turn, a subclass of `Person` with an additional attribute for employment; it inherits the attributes from its parent class `Person`.

Objects can be assigned to classes using the symbol `:`. In the ontological view, this means to relate an instance to a concept. Here, the symbol `:` is used to state that UbiqBiz is a company, that MisterX is a person, etc. Since any object can serve as a class, classes can be declared as instances of other classes, and thus F-Logic supports metamodelling facilities.

Rules

In the Semantic Web context, F-Logic is primarily perceived as a language following the rule-based paradigm. Indeed, LP-style rules form the essential language feature for the deductive aspects of F-Logic.

The keyword `FORALL` – to indicate universal quantification of involved variables – is used together with the symbol `<-` to construct rules in F-Logic. A rule

```
FORALL  <variables> <head>   <-   <body>.
```

has the typical reading: for any possible instantiation of variables in the rule body, derive the corresponding instantiation of the rule head. By deriving new information, rules extend an F-Logic object base by intensional knowledge, forming its deductive closure.

The following is an example of a rule that operates on the descriptions of the classes and objects given before.

```
FORALL  C,E  C[hasEmployees ->> E]   <-   E : Employee[isEmployedAt -> C].
```

It captures a part of the inverse relationship between the attributes `hasEmployees` and `isEmployedAt`. Whenever an employee can be derived to be employed at a certain company, the rule derives that this employee is among the list of employees of that particular company.

Another, more complex example of a rule is the following, taken from Sect. 3.1.

```
FORALL  B,C,P  P : Employee[isEmployedAt -> C] <- P : Person AND
                                                  C : Company AND
                                                  B : Booking[bookedBy -> C,
                                                              bookedFor -> P].
```

It concludes a person to be an employee of a certain company whenever there is a booking for this person by that particular company. From the concrete booking `BookingUbiqMrX` for flight FL4711, specified before, this rule would derive the F-molecule

```
MrX : Employee[isEmployedAt -> UbiqBiz].
```

stating that MisterX is an employee of UbiqBiz.

Queries

F-Logic provides queries as a language element for the retrieval of (tuples of) objects. Objects are bound to possible instantiations of variables that occur in the query. Syntactically, queries in F-Logic are a special kind of rules with an empty head and have the following form.

```
FORALL  <variables>   <-   <body>.
```

As with rules, the variables that occur in the body of a query are universally quantified. Whenever a tuple of objects is a possible instantiation of variables that conform with the deductive closure of the object base, this tuple is part of the result for the query.

An example for an F-Logic query is the following,

```
FORALL E,A   <-   E : Employee[isEmployedAt -> UbiqBiz,
                               hasAddress -> A[hasCity -> NewYork]].
```

asking for all UbiqBiz employees who live in New York. Applied to the formerly described objects and rules, the answer to this query would be the object `MrX` because MisterX is assigned an address in New York and he can also be derived to be an employee.

Queries can also ask for schema elements and bind variables to classes. The following query asks for all classes which MisterX belongs to.

```
FORALL C   <-   MrX : C.
```

The answer to the query is the set $\{$`Person`, `Employee`, `LegalEntity`$\}$ of classes.

Negation as Failure

Under the semantics of the logic programming variant, F-Logic makes the closed-world assumption for the evaluation of queries and for the deductive closure on an object base. For example, the query

```
FORALL E   <-   E : Employee.
```

that asks for all employees only yields `MrX` as a result. For `MrY` and `MsZ`, it has not been stated that they are employees, nor can this information be derived from the specified knowledge. Therefore, MisterY and MissZ are assumed to be no employees.

Furthermore, the negation operator NOT, used in the bodies of rules and queries, is interpreted as negation-as-failure. The following is an example of a query that contains a negation operator, combined with a rule.

```
FORALL P  P : FlightParticipant   <-   F : Flight AND
                                       B : Booking[bookedFor -> P,
                                                  issuedFor -> F].
FORALL E   <-   UbiqBiz[hasEmployee ->> E] AND
                NOT E : FlightParticipant.
```

It asks for all the employees of UbiqBiz who do not participate in any known flight, which yields the set $\{\texttt{MrY}, \texttt{MsZ}\}$.

Software Support for F-Logic

Since F-Logic sets a focus on rule-based inferencing rather than on web aspects, it does not come in a web-style XML serialisation format like other ontology languages in the Semantic Web. Its syntax rather resembles the style of typical programming languages and is human-readable for people with a software development background. To this end, there is not much support in graphical editing tools and F-Logic ontologies are typically developed using text editors. An exception is OntoStudio,[33] which provides graphical editing capabilities for F-Logic rules, while some other ontology editors also support F-Logic export features.

There are two major inference engines available that perform reasoning on F-Logic rules: the freely available $\mathcal{F}$LORA-2[34] [49] and the commercial OntoBroker[35] [12]. Recently, also the KAON2[36] system has included some support for F-Logic.

3.4.5 WSML

The WSMO[37] initiative aims at providing an overarching framework for handling Semantic Web Services (SWS). It comprises the WSMO conceptual model, as an upper-level ontology for Semantic Web Services, the WSML language and the WSMX execution environment. WSMO (Web Service Modelling Ontology) is described in Part III Chap. 7 in more detail, while here we are concerned with ontology language aspects. WSML (Web Service Modeling Language) is a language to formally describe the elements defined in the WSMO conceptual model, providing syntax and formal semantics for them.

WSML is particularly designed for describing Semantic Web Services and is therefore not a mere ontology language. Besides typical ontological notions, it also provides SWS-specific language constructs, such as “goal”, “web service”, “interface”, “choreography” or “capability”, to capture different aspects of Web Service

[33] http://www.ontoprise.de/content/e1171/e1249/index_eng.html
[34] http://flora.sourceforge.net/florahome.php
[35] http://ontobroker.semanticweb.org/
[36] http://kaon2.semanticweb.org/
[37] http://www.wsmo.org

semantics. One of the corner stones in WSMO are the domain ontologies used to semantically annotate Web Services. Hence, WSML also provides means to describe such ontologies, as any ordinary ontology language does. Since here we are interested in the description of ontologies in general, we present the ontology-related part of WSML only.

Syntax

The Syntax of WSML is split into a *conceptual* part and a *logical expression* part. The conceptual syntax allows typical conceptual modelling with concepts, relations and instances, known from frame-based systems where information about a certain entity is specified locally in a single syntactic construct. The logical expression syntax allows the formulation of complex axiomatic information using logical formulas. It is very similar to F-Logic syntax and provides the typical logical symbols as well as different forms of negation and implication, LP-style rules and constraints. WSML also supports datatypes like integer, float or string, up to user-defined datatypes.

The following listing shows a fragment of our example geographic ontology in WSML syntax in its *human readable* serialisation.

```
concept GeographicRegion
    isRegionFor inverseOf(locatedIn) ofType GeographicLocation
    contains inverseOf(isContainedBy) transitive impliesType GeographicRegion
    boundedBy ofType (2 *) SurfacePoint

concept SurfacePoint
    hasLongtitude ofType _float
    hasLatitude ofType _float

concept City subConceptOf Infrastructure
    locatedIn ofType PlanarRegion
    officialName ofType _string
    numberOfInhabitants ofType _integer

concept EuropeanCity subConceptOf City

instance Europe memberOf Continent

instance Munich memberOf City
    officialName hasValue "München"
    numberOfInhabitants hasValue 1288307

axiom EuropeanCity_sufficient_condition definedBy
    ?c memberOf EuropeanCity :- ?c memberOf City and
                ?c[locatedIn hasValue ?rc] and
                ?rc[containedBy hasValue ?re] and
                ?re[isRegionFor hasValue Europe].
```

The upper part shows the conceptual syntax with bold-faced keywords for defining concepts, instances and their membership relations. *Attributes*, i.e. relations defined in the scope of a concept, are further restricted or filled with concrete values. They can be declared as being transitive or as the inverse of another attribute, and they can be constrained by their range type or cardinality. With the distinction between the **ofType** and **impliesType** constructs, WSML offers both range constraints that ensure

attribute values to be of a certain type, and range restrictions in the style of OWL that allow to conclude information about attribute values. Attribute ranges can be concepts or datatypes, such as **_integer**, **_float** or **_string**. The lower part of the listing shows an axiom defined by a logical expression in form of an LP-style rule with variables preceded by a ? symbol. The rule concludes a city to be European if its geographic region lies within that of Europe, referring to the elements declared in the conceptual part.

Besides the human readable form, there are other forms of serialisation for the WSML syntax, similar to the different serialisation formats for OWL. These cover serialisation in XML as well as in RDF.

Semantics

Similar to OWL, WSML comes in various language variants that have different expressiveness and that reflect different knowledge representation paradigms. The most basic and least expressive variant is WSML-Core, which is based on DLP [18] as a least common denominator for description logic formalisms on the one hand and logic programming and rule-based systems on the other hand. WSML-Core is separately extended in the directions of these two paradigms by the variants WSML-DL and WSML-Flight/Rule, respectively. Ultimately, the vision of WSML-Full is to semantically amalgamate the two paradigms in a language with first-order model-theoretic semantics augmented by non-monotonic extensions and typical LP-style features like default negation or constraints. At the current stage, however, the WSMO initiative is an ongoing effort and the semantics of WSML-Full is yet to be defined.

In Fig. 3.5 the WSML language variants are positioned with respect to different knowledge representation formalisms.

WSML-Core

This variant is based on the DLP fragment described in [18]. It offers basic conceptual modelling with concepts, attributes and instances, as well as taxonomic hierarchies and the use of datatypes. Its semantics is defined by a mapping to function-free horn logic interpreted in the classical model-theoretic way. Similar to RDF, it does not allow to express any form of negative information, and thus no contradictory statements can be formulated.

WSML-DL

This variant extends WSML-Core to a description logic formalism, namely to the logic $\mathcal{SHIQ}(\mathrm{D})$. In this sense, WSML-DL is very similar to the OWL language. The WSML syntax does not provide the variable-free constructs that are typical for DLs. Thus, in WSML-DL logical expressions with variables and logical connectives are interpreted as in first-order logic, with the restriction to only allow unary and binary predicates for DL concepts and roles.

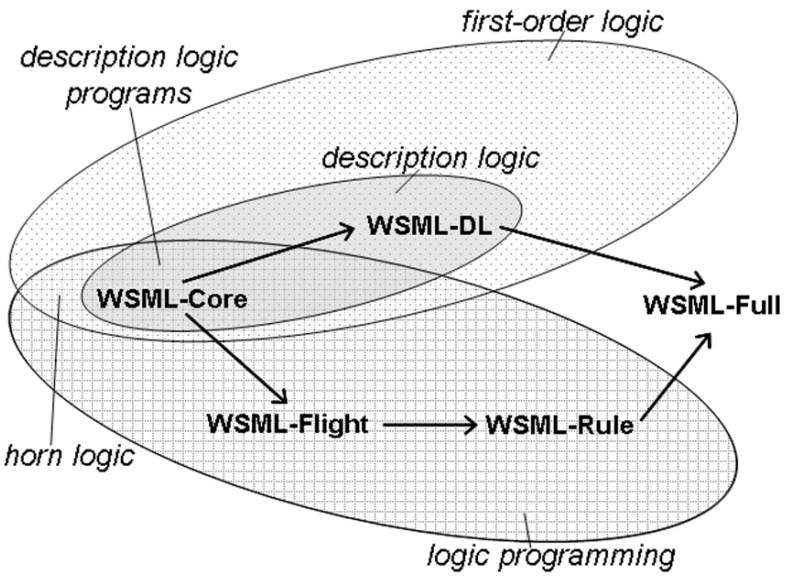


Fig. 3.5. WSML language variants in knowledge representation

WSML-Flight

This variant extends WSML-Core to an LP-style rule language with a closed-world semantics. It is similar to F-Logic (LP) and offers features like negation-as-failure, constraints and meta-modelling. The semantics of WSML-Flight is defined by a mapping to F-Logic formulas interpreted under perfect model semantics.

WSML-Rule

This variant further extends WSML-Flight with more expressive logic programming features, such as function symbols or unsafe rules. Its semantics is based on the well-founded semantics.

WSML-Full

The still-to-be-defined semantics of WSML-Full is envisioned to combine WSML-DL and WSML-Rule. A candidate formalism to achieve integration of the two paradigms is autoepistemic logic.

Software Support for WSML

Since WSML is a relatively new language, tool development is in an early stage. However, there are some tools available for handling and editing WSML ontologies, all driven by the WSMO initiative.

The WSMO4J[38] framework enables parsing, serialisation and in-memory processing of WSML ontologies an other WSML elements.

The Web Services Modelling Toolkit (WSMT)[39] is a graphical editor that allows for visualisation and manipulation of WSML ontologies. Other tools for editing WSML elements are WSMO Studio[40] [13] and DOME.[41]

3.5 Outlook

In this chapter we have presented an overview on the topics of knowledge representation, ontologies and Semantic Web languages. Here we want to briefly sketch future research and usability issues around these knowledge-based technologies.

Having reviewed various ontology languages and knowledge representation paradigms, we have seen that there are multiple different ways of approaching the representation and computational handling of knowledge. There is still room for research on which approach is most suitable for which kind of application context. A current trend in ontology languages is to perceive LP-based approaches as particularly suitable for data-intensive retrieval tasks with rule-based inferencing on the one hand, and DL-based approaches for automated classification and for satisfiability problems on the other hand.

To achieve wide-spread use of ontologies, they have to be established as usable software artefacts that are interchanged and traded between parties, similar to computer programs or other forms of electronic content. As such, they can principally be plugged in systems that make use of knowledge-based technology. However, the logic-based notions in which ontologies are described are typically too technical and too onerous to handle to be widely accepted. To overcome this deficiency, design methodologies and higher-level descriptive languages should be introduced that abstract from the surfeit of logical details, presenting the user a more intuitive view on domain knowledge. An analogous level of abstraction has been achieved in the field of software engineering, where more and more abstract higher-level languages have been build on machine codes and assembler languages.

In the Semantic Web context, also other techniques from the field of Artificial Intelligence are used, such as lexical methods for natural language processing or statistics-based methods for machine learning. There, the symbolic knowledge representation in ontologies should used complementarily to exploit synergies with such techniques in Semantic Web applications. Moreover, there is a trend to forbear from the heavy-weight semantics of logical formalisms, moving to the light-weight semantics of languages with decreased expressive power in applications where precision and exactness is not the main focus. In this sense, some applications prefer, e.g., RDF(S) over the semantically richer OWL due to simplicity or scalability issues.

[38] http://wsmo4j.sourceforge.net/
[39] http://wsmt.sourceforge.net
[40] http://www.wsmostudio.org/
[41] http://dome.sourceforge.net/

Finally, there is much space for research on finding the right degree of formality in semantics for a particular application scenario.

References

1. G. Antoniou. *Nonmonotonic Reasoning*. MIT Press, 1996.
2. F. Baader, S. Brandt, , and C. Lutz. Pushing the EL Envelope. In *Proceedings of the 19th Int. Joint Conference on Artificial Intelligence (IJCAI-05), Edinburgh, UK*. Morgan Kaufmann, 2005.
3. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, January 2003.
4. F. Baader and B. Hollunder. Embedding Defaults into Terminological Knowledge Representation Systems. *Journal of Automated Reasoning*, 14:149–180, 1995.
5. S. Bechhofer, R. Volz, and P. Lord. Cooking the Semantic Web with the OWL API. In *Proc. of the First International Semantic Web Conference 2003 (ISWC 2003), October 21-23, 2003, Sanibel Island, Florida*, 2003.
6. P. Bonatti, C. Lutz, and F. Wolter. Description Logics with Circumscription. In *Proceedings of the 10th Int. Conference on Principles of Knowledge Representation and Reasoning, KR-06*, 2006.
7. D. Brickley and R.V. Guha. RDF Vocabulary Description Language – RDF Schema. `http://www.w3.org/TR/rdf-schema/`, 2004.
8. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 54–68, London, UK, 2002. Springer.
9. D. Calvanese, G. de Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. DL-Lite: Tractable Description Logics for Ontologies. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005)*, 2005.
10. Ó. Corcho, M. Fernández-López, A. Gómez-Pérez, and Ó. Vicente. WebODE: An Integrated Workbench for Ontology Representation, Reasoning, and Exchange. In *EKAW*, p. 138–153, 2002.
11. E. Craig. Ontology. In E. Craig, editor, *Routledge Encyclopedia of Philosophy*, pages 117–118. Routledge, New York, 1998.
12. S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: Ontology Based Access to Distributed and Semi-Structured Information. In *Semantic Issues in Multimedia Systems. Proceedings of DS-8*, pages 351–369, 1999.
13. M. Dimitrov, A. Simov, V. Momtchev, and D. Ognyanov. WSMO Studio - An Integrated Service Environment for WSMO. In *Proc. of the 2nd WSMO Impl. Workshop, Innsbruck, Austria*, 2005.
14. F.M. Donini, D. Nardi, and R. Rosati. Description Logics of Minimal Knowledge and Negation as Failure. *ACM Transactions on Computational Logic*, 3(2):177–225, 2002.
15. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005.
16. O. Etzioni, K. Golden, and D. Weld. Tractable Closed World Reasoning with Updates. In *Proceedings of the 4th International Conference on Knowledege Representation and Reasoning (KR-1994)*, pages 178–189. Morgan Kaufmann, 1994.

17. A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider. Sweetening Ontologies with DOLCE. In *EKAW-02: Proceedings of the 13th Int. Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 166–181. Springer, 2002.
18. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logics. In *Proceedings of WWW-2003, Budapest, Hungary*, pages 48–57. ACM, 2003.
19. T.R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 6(2):199–221, 1993.
20. N. Guarino. Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction, and Integration. In M.T. Pazienza, editor, *Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology*, number 1299 in LNCS, pages 139–170. Springer-Verlag, 1997.
21. N. Guarino. Formal Ontology and Information Systems, Preface. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems, FOIS-98, Trento, Italy*, pages 3–15. IOS Press, 1998.
22. N. Noy M. Musen H. Knublauch, R. Fergerson. The Protege OWL Plugin: An Open Development Environment for Semantic Web Applications. *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, 2004.
23. V. Haarslev and R. Möller. Description of the RACER System and its Applications. In *International Workshop on Description Logics*, 2001.
24. P. Hayes. RDF Semantics. `http://www.w3.org/TR/rdf-mt/`, 2004.
25. I. Horrocks. Using an Expressive Description Logic: FaCT or Fiction? In *Proceedings of the 6th International Conference on Knowledege Representation and Reasoning (KR1998)*, pages 636–645. Morgan Kaufmann, 1998.
26. I. Horrocks and P. F. Patel-Schneider. A Proposal for an OWL Rules Language. In *Proceedings of the 13th International World Wide Web Conference (WWW-2004)*. ACM, 2004.
27. U. Hustadt, B. Motik, and U. Sattler. Data Complexity of Reasoning in Very Expressive Description Logics. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), Edinburgh, UK*, pages 466–471. Morgan Kaufmann, 2005.
28. A. Kalyanpur, B. Parsia, E. Sirin, B. Cuenca Grau, and J. Hendler. Swoop: A Web Ontology Editing Browser. *Journal of Web Semantics*, 4(2):144–153, 2006. `http://dx.doi.org/10.1016/j.websem.2005.10.001`.
29. M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.
30. G. Klyne and J. Carroll. RDF Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-primer/`, 2004.
31. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1988.
32. B. McBride. Jena: Implementing the RDF Model and Syntax Specification. In *SemWeb*, 2001. `http://CEUR-WS.org/Vol-40/mcbride.pdf`.
33. J. McCarthy. Circumscription – A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13(1):27–39, 1980.
34. J. Minker. Logic and Databases: Past, Present, and Future. *AI Magazine*, 18(3):21–47, 1997.
35. R. Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 25(1), 1985.

36. B. Motik. On the Properties of Metamodeling in OWL. In Y. Gil, E. Motta, V.R. Benjamins, and M. Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC-2005)*, volume 3729 of *LNCS*, pages 548–562. Springer-Verlag, 2005.
37. B. Motik and U. Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In Miki Hermann and Andrei Voronkov, editors, *Proc. of the 13th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006)*, LNCS, Phnom Penh, Cambodia, November 13–17 2006. Springer.
38. B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proc. of the 3rd Int. Semantic Web Conf. (ISWC 2004)*, pages 549–563, Hiroshima, Japan, November 7–11 2004. Springer.
39. I. Niles and A. Pease. Towards a Standard Upper Ontology. In C. Welty and B. Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, 2001.
40. P.F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language; Semantics and Abstract Syntax. `http://www.w3.org/TR/owl-semantics/`, November 2002.
41. S. Pepper and G. Moore. XML Topic Maps (XTM) 1.0. `http://www.topicmaps.org/xtm/1.0/`.
42. R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13:81–132, 1980.
43. S. Russel and P. Norvig. *Artificial Intelligence – A Modern Approach*. Prentice-Hall, 1995.
44. E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A Practical OWL-DL Reasoner. Technical report, University of Maryland Institute for Advanced Computer Studies (UMIACS), 2005. `http://mindswap.org/papers/PelletDemo.pdf`.
45. J.F. Sowa. *Knowledge Representation*. Brooks Cole Publishing, Pacific Grove, CA, USA, 2000.
46. Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. OntoEdit: Collaborative Ontology Development for the Semantic Web. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 221–235. Springer-Verlag, 2002.
47. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volumes I and II*. Computer Science Press, 1989.
48. F. van Harmelen, Z. Huang, H. Stuckenschmidt, and Y. Sure. A Framework for Handling Inconsistency in Changing Ontologies. In Y. Gil, E. Motta, V.R. Benjamins, and M. Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC-2005)*, volume 3729 of *LNCS*, pages 353–367. Springer-Verlag, 2005.
49. G. Yang, M. Kifer, and C. Zhao. Flora-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web. In *CoopIS/DOA/ODBASE*, pages 671–688, 2003.
50. M. Yue and L. Zuoquan. Infering with Inconsistent OWL DL Ontology: a Multi-valued Approach. In *Proceedings of the International Conference on Semantics in a Networked World, ICSNW-2006, Munich, Germany*. Springer-Verlag, 2006.