الجمهوريـــة الجزائريـــة الديمقراطيـــة الشعبيـــة
People's Democratic Republic of Algeria
وزارة التعليـــم العالـــي والبحـــث العلمـــي
Ministry of Higher Education and Scientific Research
جامعـة محمـد البشـير الابراهيمـي – برج بوعريريج
University of Mohamed El Bachir El Ibrahimi - Bordj Bou Arréridj

Lecture notes on
Ontology and Semantic Web

Compiled by
SABRI Lyazid

Department of Mathematics and Information Technology
Mohamed El Bachir El Ibrahimi university

Course Objectives

The objective of this course is not to serve as an exhaustive reference work. Indeed, comprehensive studies already exist on specific topics such as XML, RDF, RDFS, SPARQL or OWL. Moreover, the field of the Semantic Web does not necessarily require a complete textbook, since definitions, specifications, and practical guides are widely available online. We therefore adopt an approach focused on presenting the fundamental concepts and key techniques, providing sufficient detail to enable students to master the essential notions and to develop their own applications in a constructive and independent manner. To reinforce this learning process, the chapter also includes practical exercises and hands-on activities designed to help students apply the concepts and consolidate their understanding through practice. The course presents the principles of automatic reasoning, enabling systems to infer new knowledge from existing information and supporting consistent decision-making. The course also covers the logical foundations necessary for formal knowledge representation, including classical logic and description logic, which are essential for defining relationships between concepts and specifying constraints on data.

Concrete applications of ontologies are illustrated across multiple domains, including the Semantic Web, the Internet of Things (IoT), healthcare (e.g., SNOMED CT for disease classification), and industry, demonstrating how ontologies facilitate interoperability, consistency, and logical reasoning over complex knowledge bases.

Level of study

This course targets Master 2 students specializing in Information and Communication Technologies (ICT).

Assessment procedures

Students will be assessed based on practical work and workshop activities. A final exam will be held at the end of the module.

Prerequisites for the Course

Students are expected to have a basic understanding of graphs and networks, which are essential for knowledge modeling. They should also be familiar with how the Web functions, including HTTP, HTML, and URLs. A foundational knowledge of formal reasoning, inference, and proof concepts is required, along with familiarity with propositional logic and first-order logic. Finally, students should have experience with a programming language such as Java or Python.

Content

1. Chapter 1 : Introduction to Ontologies

Humans access the world by naming it. In their earliest encounters with the world, they spontaneously assign words to things based on their lived experience: a flame can be joy or pain, depending on the emotion that accompanies it. Gradually, they abandon this intimate naming to adopt the common language imposed by society, thus integrating their unique experience into the collective symbolic order. Through language, humans weave a symbolic web, the fabric of their world. Each symbol refers to a part of their experience; Each experience shades a thread and gives it depth. Gradually, this web becomes a representation of reality: a memory of experiences, an organisation of knowledge, and a support for reasoning. Emphasizing certain aspects of our perception, with the primary goal of understanding its behavior. When words fail, humans resort to metaphors or images to give form to what they perceive. The image then acts as a temporary substitute for the word, allowing them to inscribe the experience in the symbolic web and share it with others. Thus, each individual builds a true personal ontology throughout their life, rooted in their emotional and sensory memory. To communicate and share, they must align this intimate ontology with those of the community. This shift from the individual to the collective is the foundation of knowledge interoperability—between humans, but also between humans and machines in AI and the Semantic Web. The human brain is distinguished by its ability to generate different ontologies based on its domains of experience, and to align them almost instantly with those of others. This conceptual **adaptability** is at the heart of communication. But transposing it to the digital world remains a challenge: we are still struggling to equip machines with algorithms capable of creating and aligning contextual ontologies with the flexibility, speed, and finesse of the human mind. Can we ever teach this plasticity, or will it remain a unique privilege of the human mind? **SABRI LYAZID**

1.1. Ontology: What exactly are we talking about?

Ontologies are widely used in knowledge engineering, artificial intelligence, and computer science, in applications related to knowledge management, natural language processing, information retrieval, database design and integration, bioinformatics, education, e-commerce, intelligent integration of information such as sensor data, electronic medical records, and in new emerging fields such as the Semantic Web. Thus, ontologies help solve several significant problems that hinder communication between or among people/robots, organisations, and/or software systems. Indeed, due to different needs and contexts, there may be different perspectives and assumptions regarding what is essentially the same topic. Thus, each uses a different vocabulary and each may have different concepts, structures, and methods that overlap and/or do not match. The resulting lack of shared understanding leads to poor communication within and between these people/robots and their organisations.

1.1.1. What exactly are we talking about

The term “ontology” is used with different meanings across various communities. Among philosophers and computer scientists, particularly in knowledge engineering, the interpretations diverge significantly. In knowledge engineering, ontology is understood as a discipline that enables the development of digital artifacts capable of encoding and conveying meaning. Leveraging mathematical foundations, knowledge engineering employs logical formulations for the formal manipulation of symbols and, consequently, for processing meaning. We will explore ontologies in greater detail throughout this document.

To better understand, let us first consider the concept of knowledge and its characteristics. Although definitions vary, we adopt the perspective that knowledge allows the mind to construct action plans or strategies beyond the immediate demands of a situation. Hadot (1969) emphasizes the principle of acquiring self-knowledge to design one’s own happiness. Similarly, the Roman philosopher Marcus Aurelius (121–180 CE) asserts that our power resides within the mind, not in external circumstances. Furthermore, careful observation of the environment enables us to discern what is beneficial, feasible, or harmful, rather than reacting solely based on emotions or external constraints.

Historically, philosophers of the 18th century viewed knowledge as the natural ability to perceive and distinguish phenomena. Two centuries later, knowledge was further defined as the contribution of thought to external reality, linking the concept to truth as the adequacy between mind and object.

In the context of modern big data, confusion often arises between data, information, and knowledge. For instance, consider the example provided by Xavier Aimé and Frank Arnould: a heat source is thermal energy for physicists. This energy triggers chemical, mechanical, and other reactions, making it a factual phenomenon. However, it is neither data (not directly observable), nor information (requiring interpretation), nor knowledge (not yet assimilated). The sensation of heat perceived by our sensory receptors constitutes data. Interpretation of this data generates information, while reflection and experience transform it into knowledge—for example, associating a burning heat source with avoidance behavior. The organization and formal representation of such knowledge require structured mechanisms, the pinnacle of which are computational ontologies. These ontologies enable the systematic encoding, sharing, and reasoning over knowledge within digital systems.

1.2. Stoicism, Knowledge, and Understanding of the World

You must be the designer of your own future; detach yourself from your past. While the past can serve as a refuge to justify failures, it only holds power over you depending on the meaning you have assigned to it. Humans are not affected by events themselves, but by their interpretation and the judgments formed about these actions and external occurrences. Every experience and failure can reveal a hidden capacity within you—an innate vocation or power.

A key to shaping the future is therefore to overcome challenges by understanding the present and learning from the past. As Thomas Edison famously said, he did not fail to light a bulb after hundreds of attempts; rather, he discovered hundreds of ways that did not work.

From the Stoic perspective, logic studies the conditions for accessing knowledge[1]. Awareness and understanding allow us to distinguish between self-knowledge and comprehension of the world, thus enabling the free design of our actions.

Throughout history, humans have used simple tools to facilitate cognition: stones for counting or memory, for example, allowed knowledge to be constructed without mastering arithmetic operations. Similarly, a flame can be interpreted in multiple cognitive ways: as a source of heat, light, burns, or fire. These examples illustrate that knowledge emerges through interpretation and understanding of the phenomena around us, not merely from their existence.

1.3. Concepts in Cognitive Science and Knowledge Representation

Understanding what a concept truly is constitutes a fundamental step in any scientific inquiry. Before addressing complex theoretical or computational models, it is necessary to clarify how human beings define, structure, and relate concepts to one another. Concepts form the foundation upon which all reasoning, theories, and representations of knowledge are built. They serve as mental tools that allow us to categorize experience, interpret phenomena, and communicate meaningfully about the world. The following sections explore this notion from complementary perspectives, psychological, philosophical, linguistic, and neuroscientific, to show how the definition, reference, and validation of concepts shape both human cognition and artificial knowledge systems.

1.3.1. Defining Concepts

Concepts are the basic building blocks of all theories, models, and scientific discourse. According to the American Psychological Association (2015), attention is defined as a state in which cognitive resources focus on certain aspects of the environment, while the central nervous system remains ready to react to stimuli. From this definition, two key ideas emerge:

1. The mind selectively focuses on some stimuli rather than others.
2. The mind is prepared to react to these stimuli.

Thus, each concept should be specific and context-dependent. For example, listening or observing allows the mind to pay attention selectively, either visually or auditorily, to certain stimuli.

Memory, another essential concept, allows us to associate past experiences (autonoetic self-knowledge) to relive them in the present and even anticipate future events. Questions arise

[1] https://la-philosophie.com/le-stoicisme

regarding animals as well: can they possess episodic memory similar to humans, i.e., experiences situated in space, time, and emotion?

1.3.2. Meaning and Reference

Philosophical insights help clarify the distinction between meaning and reference. Klein suggests that measurement can establish a concept's reference but not its meaning. Frege (1892) defined meaning (Sinn) as a mode of access to denotation (Bedeutung). For instance, the same individual can be identified as a doctor, a husband, or a teacher, depending on context, while an object like a table can refer to a work desk or a café table.

1.3.3. Scientific vs. Vernacular[2] Terms

A semantic shift often occurs when everyday language is adapted for scientific use. Scientists aim to make complex concepts and theories accessible to non-specialists, which can simplify or alter their original meaning. Francken & Slors (2014) suggest that scientific concepts can be subdivided into sub-concepts, emphasizing that concepts rarely exist in isolation. Rather, they form hierarchies or networks of semantic relationships. Consequently, a concept's definition depends on its position within this network.

1.3.4. Cognitive Neuroscience and Concept Validation

Modern cognitive neuroscience provides tools to observe how the brain represents concepts. By examining brain activation during object recognition or memory tasks, researchers can infer which cognitive processes correspond to specific concepts. Techniques such as fMRI (functional magnetic resonance imaging) or PET (positron emission tomography) reveal the relationship between cognition and brain structures. Integrating data from genomics[3], proteomics[4], and psychiatric studies further enhances our understanding of complex cognitive functions (Xavier Aimé & Frank Arnould). It is obvious that Franz Gall's theory (1809), which introduces the principle of phrenology[5], should be avoided.

1.3.5. Implications for Knowledge Representation

The interplay between philosophical definitions, cognitive science, and neuroscience has direct implications for knowledge representation. Understanding concepts, their meaning, and their hierarchical organization is crucial for building computational ontologies, designing artificial intelligence systems, and modeling human cognition.

[2] What is related to the language or culture of a country, a group, or to local/traditional practices
[3] Study the DNA sequenc
[4] Study the proteins in a cell, tissue, organ, etc.
[5] A science that studies and evaluates human intellectual and moral faculties by inspecting the bumps on the head (https://www.cnrtl.fr/definition/phrénologie). It assumed that instincts, character, aptitudes, and mental and emotional faculties were, by virtue of cerebral localizations, conditioned by the external shape of the skull.

1.4. Bibliographic Databases

Databases continue to grow (think Big Data). However, querying these reservoirs relies on coherent relationships between concepts (we speak of equivalence relationships, hierarchy, etc.). Thus, the possible semantic relationships between concepts are not sufficiently explicit. One way to share these concepts and enable better knowledge extraction is to find standard terminologies by building consensus within scientific communities. Thus, this type of controlled vocabulary will describe the data in a very formal and semantically rich manner. This data (we are actually talking about knowledge) will be exploited more or less easily by machines. This is precisely the role offered by ontologies in knowledge engineering, a branch of artificial intelligence. The latter will enable the design of knowledge representation systems. Many existing ontologies that are gaining popularity now, such as medical ontologies, Drug Ontology (DRON), Symptom Ontology (SYMP), and Human Disease Ontology (DOID), Genetic Ontology (GO). The development of the semantic web aims to link multiple pages, particularly data on the network. Thus, ontologies can be a solution in different fields to formalise concepts and their relationships. As very well pointed out by Eisenberg et al. 2019, ontologies constitute a research tool and are selected as a real additional source of scientific discoveries. They will facilitate the clarification of concepts, in particular cognitive concepts and their relationships.

So knowledge is generated following a perception that will gradually become based on descriptive properties (either through words) or functional properties linked to already acquired knowledge, allowing the construction of a semantic network.

1.5. Philosophical Origins of Ontology

The term *ontology*, a pivotal concept in the history of ideas, first appeared in 1606 in the writings of Jacob Lorhard. His work reflects contemporary debates on the relationship between scientific inquiry and religious belief, while also providing a bridge to the intellectual landscape of the past. Lorhard's distinction between what depends on the rational mind and what belongs to the real world—though not yet between the abstract and the concrete—marks a key milestone in the evolution of ontology. A few years later, Rudolf Goclenius (1613) revisited the term in a study on being, its categories, and its fundamental relations.
Among scholastic philosophers, ontology held a distinctive role as both a theory and a methodology of thought. It sought to structure the categories of being and to define a universal framework for reasoning. From its very beginning, ontology thus combined two perspectives: a philosophical reflection on being and a methodological tool for organizing knowledge.

1.5.1. From the Philosophical Framework to Artificial Intelligence

The emergence of artificial intelligence (AI) has renewed interest in the concept of ontology, notably through the work of John McCarthy (1980). McCarthy proposed adopting ontology as

the foundation for representing *common-sense knowledge*—the everyday knowledge required for machines to reason.

For instance, modelling the concept of a *boat* involves specifying not only that it can be used to cross a river, but also that certain conditions may prevent its use. This requires introducing into the ontology not only the category *boat*, but also categories related to its possible limitations or malfunctions. As McCarthy observed: *"Using circumscription requires that common-sense knowledge be expressed in a form that says a boat can be used to cross rivers unless there is something that prevents its use… We must therefore introduce into our ontology categories that include something wrong with a boat or conditions that may prevent its use."*

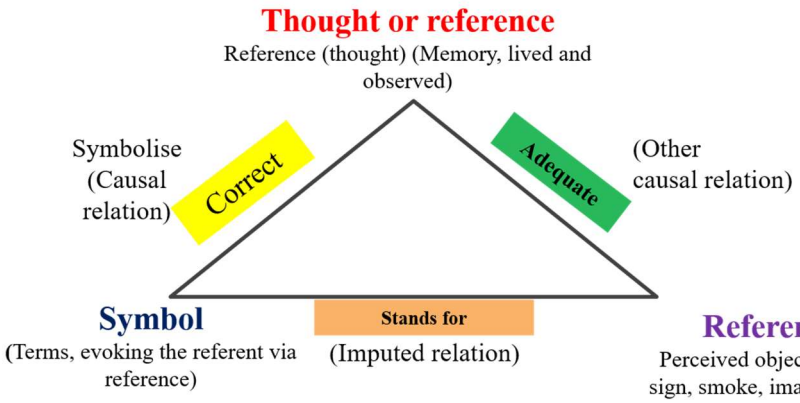
From this point onward, ontology ceased to be merely a conceptual framework and became a **practical instrument for knowledge engineering**, providing structured foundations for reasoning systems in AI:

- Facilitates knowledge sharing,
- Promotes interoperability between systems,
- Provides a basis for formalising logical theories in AI.

1.5.2. Meaning of Meaning[6]

The semiotic model of the semantic triangle proposed by Ogden and Richards (1923) illustrates the relationship between sign, referent, and concept. It highlights the linguistic foundation of ontology, where terms of natural language must be systematically linked to the conceptual entities they denote. Ogden and Richards, with The Meaning of Meaning, show that meaning is a triangular relationship: word – concept – thing. The word has no direct link with reality: it is via thought (reference) that we establish the relationship. Words do not 'mean' (i.e. relate directly to) things in the world. They are symbols, which 'symbolise' a thought or 'reference'. This thought in turn 'refers to' (or 'is of') something (a thing or event) in the world, which they term the 'referent'.

[6] https://pure.mpg.de/rest/items/item_2366948/component/file_2366947/content



• Ogden and Richards aim to clarify the notion of meaning in language.

• They distinguish language as a system of signs and its relationship to thought and the world.

• Their goal: to avoid philosophical and linguistic confusion about what "meaning" means.

Ogden and Richards emphasize that meaning is not an inherent property of words, but the outcome of a triangular relationship between:

1. The symbol — the linguistic sign (e.g., the word *tree*),
2. The thought or reference — the mental concept evoked by the word,
3. The referent — the real-world entity (e.g., the tree in the garden).

1.5.3. Key Contributions of Ogden and Richards

- The triangle corrects the simplistic view of the sign as a direct word–thing relation.
- It highlights that meaning emerges from the interaction between language, thought, and the world.
- It helps explain key linguistic phenomena:
    - *Polysemy*: a single word can activate multiple concepts.
    - *Misunderstandings*: occur when mental references are not shared.
    - *Abstract symbols*: words that lack a direct referent (e.g., *justice*).

This perspective is crucial for ontology building, as it shows why formal models must mediate between symbols, concepts, and real-world entities.

1.5.4. IoT / AI Example of an Abstract Symbol

• Symbol (word, sign): "Smartness" (intelligence of a system, or "intelligent" applied to a city, a vehicle, a house, etc.).

• Reference (thought, concept): the general idea of a system capable of capturing data, reasoning, learning, and adapting.

• Referent (perceived object / real world): there is no single concrete object. This could refer to a sensor network, an AI platform, a contextual service—but the word does not directly point to a single physical object (unlike "sensor" or "camera").

1.6. Controlled vocabulary, terminology, taxonomy and thesaurus

In knowledge organisation, several structures help standardise and relate concepts. Controlled vocabularies, terminologies, taxonomies, and thesauri represent successive levels of complexity, from simple lists of terms to rich semantic networks.

1.6.1. Vocabulary

Within the framework of RTOs (termino-ontological resources), must respect certain minimum constraints. On the one hand, when the same term is used to designate two distinct concepts depending on the context, it is necessary to adjust the naming of the concepts to eliminate any ambiguity. On the other hand, when several terms refer to the same entity or notion, it is appropriate to retain a preferred term and to consider the others as synonyms.

Examples

o Temperature sensor: can be called thermometer, thermal sensor, temp sensor.

We choose "temperature sensor" as the preferred term; the others become synonyms.

o Concept: Car

Terms used: car, automobile, auto, private vehicle

In the RTO, we choose a preferred term (e.g., automobile) and declare the others as synonyms.

1.6.2. Controlled vocabulary:

A set of terms standardized and validated by a community, without a hierarchical structure (e.g., glossary). A controlled vocabulary is an information organization tool based on a restricted and standardized list of authorized terms for indexing, categorization, or tagging. Its

function is to ensure consistency, particularly when used by multiple indexers, by limiting the introduction of new terms according to established rules and by integrating cross-references between preferred and non-preferred terms. It thus allows the selection of a single term when there are several synonyms for the same concept, in order to avoid dispersion and ensure consistency in research. Unlike a glossary, a controlled vocabulary does not provide term definitions or a hierarchical organization; its role is solely normative and functional. It can take the form of authority files (especially for named entities) or synonym rings (when no term is preferred but several variants are related). Particularly useful for indexing databases, periodicals, or web pages, controlled vocabulary improves the relevance of searches by reducing ambiguity and harmonizing documentary language. Combined with search engines and automated indexing, it provides an effective and economical intermediate solution between exhaustive manual indexing and free-text searching.

1.6.3. Terminology

Terminology refers to the study and use of terms that express notions or concepts in a given field. It helps structure and understand the world, while facilitating communication and information sharing through a clear and precise vocabulary. Terms are organized into systems of conceptual relationships, of which hyperonymy (the relationship between a generic term and specific terms) is central, but which also include hyponymy, synonymy, antonymy, and partonymy. Thus, terminology contributes to building a controlled language that organizes knowledge and ensures its common interpretation.

1.6.4. Taxonomy

The term taxonomy in ancient Greek means (taxiq: oredre, classification) and nomos to designate rule and law (administrate). Thus, taxonomy originally referred to the science of classification, particularly of plants and animals, but it is now used to designate any hierarchical system of organisation or categorisation. In this sense, a taxonomy can be considered a controlled vocabulary structured into levels of generality (general terms / specific terms), without necessarily meeting all the requirements of a thesaurus. There is increasing talk of enterprise taxonomies, which bring together controlled vocabularies adapted to organisational needs. Conceptually, a taxonomy of concepts corresponds to a set of notions organised by subsumption relationships (or Is-A relationships), where each instance of a class is also an instance of its superclasses. It therefore allows the world to be represented hierarchically and coherently, facilitating the classification, communication, and reuse of knowledge. For example, in IoT: “Sensor” → “Temperature Sensor” → “Infrared Sensor”.

1.6.5. Thesaurus

A thesaurus (from the Latin thesaurus, "treasure") is a structured, controlled vocabulary designed for indexing and document retrieval. Unlike a simple dictionary or a general-language thesaurus, it does more than provide synonyms: it organises terms according to hierarchical (generic term/specific term), equivalence (synonyms or near-synonyms, with a mention of preferred terms), and associative (related terms such as chair/table,

doctor/hospital) relationships. It also provides application notes to clarify the use of terms, ensuring greater consistency in information representation and retrieval.

A thesaurus is thus similar to an ontology, but it differs in its linguistic nature: it links terms, while an ontology links concepts and formalises their logical relationships. At the same time, thesauruses describe relationships such as generalisation/specialisation, synonymy, or association, whereas ontology models stricter and more deductive relationships, such as conceptual subsumption (Is-A) or logical properties (e.g., is-next-to, part-of).

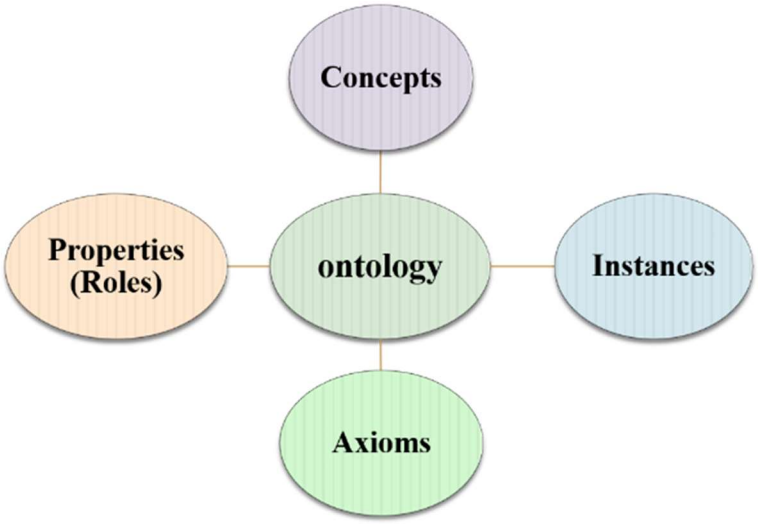
Finally, thesaurus construction is governed by international standards (ISO 2788, ISO 5964, ANSI/NISO Z39.19, BS 8723), which specify relationship categories and best indexing practices. These structures are beneficial for large terminology sets requiring human indexing or intuitive user navigation.

1.7. Fundamental Components of an Ontology

The primary objective of an ontology**, which** is to represent the reality of the world we wish to describe and model.

For instance, in a medical oncology, the concept of 'disease' and its properties like 'symptoms' and 'treatments' must faithfully reflect real-world diseases and their symptoms and treatments.

As a developer or researcher, your role in effectively testing the ontology is crucial. It is your responsibility to verify its consistency, relevance, and usefulness for the intended application.



An ontology is formally defined by a structured set of concepts, relationships, instances and axioms that together capture the semantics of a given domain.

An **ontology** is mainly a structured tool for representing and organising knowledge. It is built on several core components:

1. **Concepts**: broad categories or ideas, such as *"sensor," "connected device,"* or *"building."*
2. **Roles (Relations): D**escribes a **relation between two concepts (or individuals)**.
    - **Examples :**
        - *"sensor **isInstalledIn** building"*
        - *"sensor **measures** temperature"*
        - *"device **communicatesWith** gateway"*

We can speek also :

. Functions / Properties : A property (often called *attribute*, *datatype property* in OWL) describes an intrinsic characteristic of a concept or individual. It connects an entity to a **value** (number, string, boolean, etc.).

- **Examples**:
    - *"sensor **hasPrecision** = ±0.1 °C"*
    - *"sensor **hasSamplingFrequency** = 10s"*
    - *"device **hasBatteryLevel** = 90%"*

3. **Axioms and rules**: the constraints or necessary truths within the system, such as *"a sensor cannot be installed in two different buildings at the same time."*

Each concept can be described from two perspectives:

4. **Intension (or comprehension)**: the set of defining properties.
   IoT example: the concept *"temperature sensor"* may be defined as *"an IoT device designed to measure ambient temperature, equipped with a unit of measurement (°C, °F) and a configurable sampling frequency."*
5. **Extension (Instances)** : the concrete instances that fall under the concept.
   IoT example: {the *DHT22 sensor* in a connected living room, the *LM35 sensor* in an agricultural greenhouse, the *Xiaomi sensor* placed in a bedroom}.

In this way, an IoT ontology bridges the gap between the abstract level (the definition of the concept *"sensor"* and its properties) and the concrete level (the actual sensors deployed). This dual perspective—intension and extension—is what enables effective communication and interoperability, both among different IoT systems and between humans and machines.

1.8. Ontology Typologies

Different types of ontologies can be classified depending on their level of abstraction, purpose, and degree of formality. Understanding these typologies helps clarify whether an ontology is meant to provide general categories, describe a specific domain, or support a concrete application.

1. **Upper-level ontology (or meta-ontology)**:
   – Describes very general and abstract concepts (e.g., *time, space, object, event*).
   – Serves as a framework for organizing other ontologies.

2. **Generic (or core) ontology**:
   – Less abstract, it describes general concepts reusable across multiple domains (e.g., *person, organization, place*).
3. **Domain ontology**:
   – Specializes concepts for a particular field (e.g., *medicine, transportation, agriculture*).
   – IoT example: an ontology of **intelligent transportation** with concepts such as *bus, station, traffic sensor*.
4. **Task ontology**:
   – Targets a specific activity (e.g., *planning a surgery, detecting a failure*).
   – IoT example: *diagnosing a fault in a sensor network*.
5. **Application ontology**:
   – The most concrete, designed for a specific system.
   – IoT example: the ontology used in a **smart home application** linking *temperature sensors*, *cameras*, and *heating systems*.

Examples of Typologies

• Frame Ontology (Gruber, 1993): Representation of concepts via "frame"-type structures.

• DOLCE (Gangemi et al., 2002): A descriptive ontology focused on linguistic and cognitive modelling. • CYC (Lenat & Guha, 1989): a broad base aimed at capturing common sense for AI.

1.9. Ontology Construction

Building an ontology involves addressing several fundamental questions:

- **What knowledge should be represented?**
- **Which sources of information can be used?** (texts, databases, multimedia, or existing ontologies)
- **What level of granularity and precision is needed?**
- **How should we prioritize concepts and define their key properties?**

There are three main approaches to ontology construction:

1. **Manual construction** – developed entirely by domain experts.
2. **Semi-automatic construction** – supported by tools (e.g., terminology extraction, corpus analysis) with validation by experts.
3. **Automatic construction** – generated using machine learning or natural language processing techniques.

Finally, a crucial step is the **evaluation of the ontology**: testing its coherence, completeness, and usability in real scenarios.

1.9.1. Classical Approaches to Ontology Construction

- **Two main sources of knowledge:**
  - Human expertise.
  - Texts.
- **Expert-centered methods (KOD, CommonKADS):**
  - Rely on interviews and transcriptions to extract individual knowledge.
  - Costly approaches, with limited use of linguistic resources.
  - Often produce results that are limited or not fully relevant to the application.
- **Text-oriented approaches:**
  - **Ontological approach:**
    - Minimal reliance on texts.
    - Prioritizes reusability, but often at the expense of quality.
  - **TKB approach (Terminological Knowledge Bases):**
    - Focuses on extracting terms and relationships from a corpus.
    - Requires adaptations to fit specific application needs.
  - **Differential approach:**
    - Uses texts as the primary source of knowledge.
    - Seeks continuity between lexical expressions, conceptual models, and computational systems.
    - Typically results in the creation of *regional ontologies*.
- **General development:**

- A shift from relying solely on experts toward complementarity between experts and texts, in order to enhance the adaptability of ontologies to practical applications.

1.9.2. Ontology Acquisition from Texts

Texts are a valuable source for constructing domain ontologies. The main techniques applied include:

- **Linguistic methods:** lemmatization, grammatical analysis, and term identification.
- **Statistical methods:** frequency analysis, TF-IDF weighting, and clustering.

Constructing ontologies from texts is a specific branch of ontology engineering. It is particularly crucial for the Semantic Web, as it supports the annotation of resources and the organization of knowledge bases. Texts are privileged sources because they contain shared knowledge and are often more accessible than human experts.

An ontology is a formal and shared representation of a domain, organized into hierarchical and interrelated concepts. Most approaches follow four main steps:

1. Corpus construction.
2. Linguistic analysis.
3. Conceptualization.
4. Operationalization.

The goal of this process is to move from the textual level (discourse) to the conceptual level (ontology). This transition distinguishes three planes:

- Corpus (discourse).
- Terminology (linguistic entities).
- Ontology (concepts and relations).

This distinction enables more effective use of NLP and text mining techniques to support conceptualization.

1.10. Development Methodologies

Several methodologies have been proposed to guide the construction of ontologies

1.10.1. Uschold & King (1995)[7] :

Uschold and King's article proposes a methodology for constructing ontologies focused on goals and intended use. The process begins by clarifying the ontology's purpose and identifying key concepts and their relationships. Term definitions are first

[7] https://www.aiai.ed.ac.uk/publications/documents/1995/95-ont-ijcai95-ont-method.pdf

formulated in natural language to ensure shared understanding and address ambiguities. The methodology includes several steps: ontology capture, formal coding, integration with existing ontologies, evaluation, and documentation. The authors recommend prioritizing the definition of "cognitively basic" terms to reduce subsequent rework. This approach provides a solid foundation for developing, sharing, and reuse of ontologies tailored to specific application needs.

1.10.2. TOVE «TOronto Virtual Enterprise "(Gruninger & Fox, 1996)[8] :

oriented towards business-applied reasoning. The authors propose a rigorous methodological framework for building business ontologies, using first-order logic (FOL) to formalize concepts, objects, relationships, and properties. The process begins with defining application scenarios and competency questions, enabling the identification of essential information for each context. Then, an initial terminology of objects, relationships, and attributes is developed and refined iteratively to eliminate ambiguities and imprecisions. This terminology is formalized with FOL axioms, ensuring semantic consistency and enabling automatic reasoning. The approach thus ensures the accuracy, reusability, and integration of ontologies into information systems and business process modeling.

1.10.3. Methontology *(Lopez et al., 1997)9* :

complete cycle, inspired by software engineering.

- ✓ **Ontology Lifecycle**: METHONTOLOGY defines a set of activities organized into successive phases, from requirements specification to ontology maintenance, including conceptualization, formalization, and integration.
- ✓ **Evolving Prototypes**: The approach emphasizes incremental construction, where each version of the ontology is improved based on feedback, thus allowing continuous adaptation to users' needs.
- ✓ **Structured Methodology**: It provides clear guidelines on the activities to be carried out at each stage, the techniques to be used, and the roles of the various actors involved, thereby facilitating a systematic and reproducible approach to ontology construction.

This methodology has been successfully applied in various fields, including chemistry, and has helped establish standardized practices in the field of ontological engineering.

[8] https://www.researchgate.net/publication/2520448_The_Logic_of_Enterprise_Modelling#fullTextFileContent

[9] https://www.researchgate.net/publication/50236211_METHONTOLOGY_from_ontological_art_towards_ontological_engineering#fullTextFileContent

1.10.4. Noy & McGuinness (2001)[10] :

A pragmatic eight-step approach, from domain definition to evaluation. This methodological approach is designed to be flexible and adaptable to the specific needs of each project, while providing a solid foundation for creating effective and coherent ontologies by following seven main steps:

Phase 1: Define the boundaries of the domain.
Phase 2: Check the possibility of reusing all or part of another ontology.
Phase 3: List the terms (i.e., vocabulary) that will determine the concepts and the relationships (properties) between concepts.
Phase 4: Decide on the process to follow for defining the class hierarchy:

1- **Bottom-up process**

2- **Top-down process**

3- **Combination:** First define the most relevant concepts.

Phase 5: Define the relationships between concepts (properties).
Phase 6: Define the values (facets / role restrictions) for each property:

4- **Cardinality**

5- **Domain** (the class whose role describes the properties) and **Range** (the extent of the class)

Phase 7: Finally, create instances and define the property values.

1.11. AI for Ontology Construction

Machine learning approaches facilitate the automation of concept and relation identification, hierarchy generation, and ontology enrichment, particularly suited for large and dynamic domains such as IoT, biology, or the Semantic Web.

Automatic Concept and Relation Extraction

- ✓ Techniques: supervised learning, unsupervised learning, reinforcement learning.
- ✓ Sources: texts, databases, sensor logs, social networks.
- ✓ Objective: automatically detect entities, concepts, and relationships.
- ✓ Examples: Named Entity Recognition (NER), relation extraction via graph embeddings.

Clustering and Semantic Grouping

- ✓ Techniques: K-Means, Fuzzy C-Means, HDBSCAN, Word2Vec, BERT embeddings.
- ✓ Objective: group similar terms or concepts to create classes or subclasses.

[10] ontology101.pdf

- ✓ Advantage: automatic identification of hypernymic relations (Is-A) and co-occurrences.

1.11.1. Graph Learning (Graph Neural Networks)

- ✓ Techniques: GNNs, Graph Convolutional Networks, Knowledge Graph Embeddings (TransE, ComplEx).
- ✓ Objective: learn vector representations of concepts and relations in a graph.
- ✓ Usefulness: complete, infer, or correct ontological links.

1.11.2. Ontology Enrichment via NLP

- ✓ Techniques: Transformers (BERT, GPT), word embeddings, sentence embeddings.
- ✓ Applications:
  - Extracting conceptual hierarchies from texts.
  - Detecting synonyms and near-synonyms.
  - Automatic alignment between existing ontologies.

1.11.3. Hybrid Methods

- ✓ Combination: human experts + ML + NLP.
- ✓ Process: ML proposes concepts and relationships → validation and adjustment by the expert → enriched ontology.
- ✓ Advantages: fast, scalable, less dependent on manual effort.

1.12. Tools for Ontology Construction

1- **TextToOnto**[11] : A semi-automatic platform for extracting concepts and relations from texts. The aim of TextToOnto is to assist developers in the ontology construction process by applying text mining techniques

2- **Text2Onto**[12] : This extension of Text-To-Onto employs probabilistic and linguistic techniques to automatically generate ontologies

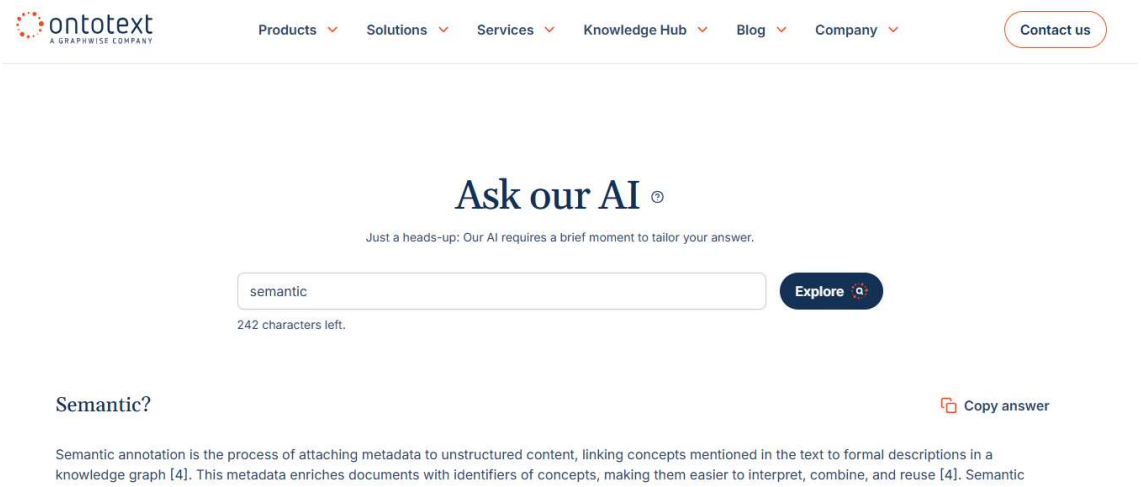
3- **OntoLT**[13] : A Protégé plugin designed for extracting terms and structuring ontologies from textual data.

4- OntoGen[14] : An interactive tool for exploring and generating concepts and relationships from corpora. It is a semi-automatic ontology editor that

[11] https://sourceforge.net/projects/texttoonto/
[12] https://phenotero.github.io/data_text2onto/
[13] https://www.dfki.de/fileadmin/user_upload/import/1216_iswc03-demo.pdf
[14] https://ailab.ijs.si/tools/ontogen-2/

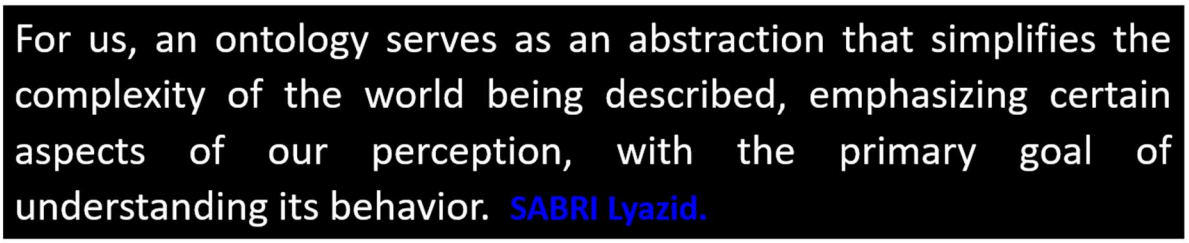
combines text mining with a user-friendly interface, enabling domain experts to create subject ontologies without advanced technical skills.

Ontotext [15]: **Enhance your LLM with a Knowledge Graph!**



1.12.1. Ontology as a Model

Finally, ontology can be considered as a model in the sense defined by Birou (1966): a mathematical, logical, or physical system that represents the essential structures of a reality and is capable of reproducing its functioning."



In the IoT, for example, an ontology can model entities (sensors, actuators), events (measurements, alarms), and relationships (location, causality), thereby facilitating the integration of heterogeneous systems.

1.13. Definition of Ontologies

Several researchers have proposed complementary definitions, ranging from a simple specification of terms and relationships to a formalization understandable by machines. At the

[15] https://www.ontotext.com/

same time, ontology finds its roots in philosophy, where it refers to the study of being and its understanding. In the context of computer science, an ontology is an essential tool for structuring and sharing knowledge in a formal and shared way, thereby facilitating communication and analysis within complex systems. According to Guarino, Oberle, and Staab (2009), as published in the *Handbook on Ontologies*, we can distinguish definitions based on:

- **Philosophy:** Ontology is the study of the nature and structure of reality, independently of its actual existence. It focuses on the essential attributes of entities, whether they exist or not.
- **Computer Science:** Ontology is an explicit and formal specification of a shared conceptualization. It serves to model the structure of a domain by identifying relevant entities and their relationships.

Conceptualization
A conceptualization is an abstract and simplified representation of the world, adapted to a specific purpose. It includes:

- Entities**:** objects, concepts, events, etc.
- Relations**:** links between entities.
- Constraints**:** rules that define the relationships and entities.

Explicit and Formal Specification
An ontology must be:

- Explicit**:** the meanings of terms must be clearly defined.
- Formal**:** represented in a machine-understandable language, excluding natural language.

1.13.1. Shared Nature

An ontology is shared when it reflects a common vision among multiple stakeholders, thereby facilitating communication and interoperability.

Ontology Structure
An ontology generally includes:

- Taxonomy**:** hierarchy of concepts.
- Relations**:** associations between concepts.
- Instances**:** concrete examples of concepts.

Ontology and Logic
Ontologies are often based on logical languages, enabling:

- Reasoning**:** to infer new knowledge.
- Validation**:** to verify data consistency.

Applications of Ontologies
Ontologies are used for:

- Knowledge Sharing**:** to facilitate common understanding.
- Interoperability**:** to enable data exchange between systems.
- Data Analysis**:** to extract relevant information.

1.14. Definitions Evolving Across the Years

1.14.1. Origins and Philosophical Foundations

Classical Philosophy

- Aristotle: the science of being as being.
- General philosophy: ontology as a branch of metaphysics, concerned with being and the discourse about being.

Contemporary Philosophy

- **Heidegger:** the understanding of being, emphasizing how being manifests itself and is understood in the world.

1.14.2. Ontology as Vocabulary and Relations

Neches et al. (1991):
An ontology defines the basic terms and the relationships that make up the vocabulary of a domain, as well as the rules for combining these terms and relationships to extend this vocabulary.

Fikes (1999):
Ontologies are considered as domain theories that specify a particular vocabulary including entities, classes, properties, predicates, and functions, along with a set of necessary relations among these terms. They provide a vocabulary to represent knowledge of a domain and describe specific situations.

1.14.3. Ontology as Conceptualization

Gruber (1993):
An ontology is an explicit specification of a conceptualization, corresponding to an abstract and simplified view of the world that one wishes to represent.

Borst (1997):
An ontology is a "formal specification of a shared conceptualization."

Studer (1998):
An ontology is a formal, explicit, and machine-understandable specification of a conceptualization shared by a group of individuals.

Uschold et al. (1996):
An ontology serves as a support for a structured and formal semantic description of the concepts in a domain and their interrelations. It facilitates knowledge exchange between users and systems, as well as among systems themselves.

1.14.4. Ontology as a Set of Primitives

Gruber (2007):
An ontology defines a set of representational primitives for modeling a knowledge or discourse domain. These primitives generally include:

- Classes (or sets)
- Attributes (or properties)
- Relations (between class members)

The definitions of these primitives include their meaning and the constraints on their logical application.

1.15. Knowledge Representation Formalisms in AI

- ✓ These are the various languages, models, and structures used to represent information in a way that can be processed by a machine.
  Examples: predicate logic, semantic networks, production rules, frames, ontologies.
- ✓ Objective: to enable AI to reason, infer, and make decisions based on the represented knowledge.

To achieve knowledge representation close to human reasoning

Many "non-logical" formalisms (often based on graphical interfaces) have been developed.
It was concluded that classical logic formalisms are inadequate for knowledge representation in AI applications.

1. Semantic networks
2. Frame-based systems
3. Conceptual graphs

1.15.1. Semantic Networks

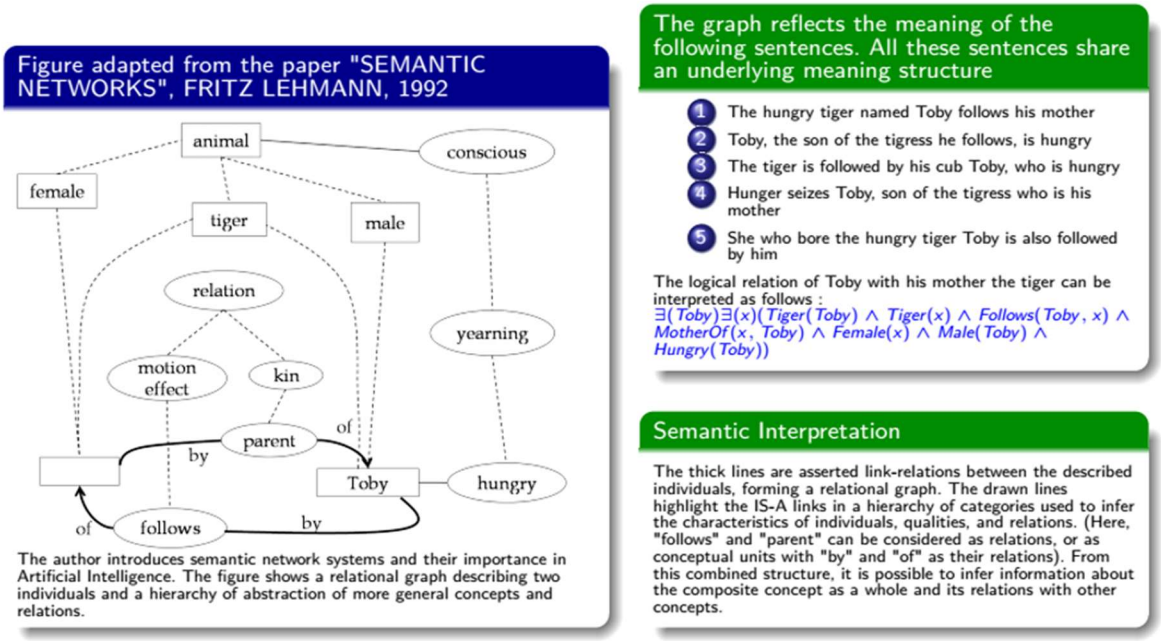
introduced by Quillian in 1967, use graphical structures to represent knowledge. Nodes symbolize objects such as concepts, situations, or events, while arcs express the relationships between these objects. These relationships can be of the "is-a-kind-of" type between concepts (e.g., *Seat is-a-kind-of Furniture*) or "instance-of" between a concept and an individual (e.g., *a chair is-an instance of Seat*). The graphical representation makes these networks easily interpretable by humans while remaining machine-processable.

- Originally introduced by *Quillian* in 1967
- Use graphical structures to represent knowledge
  - Nodes represent objects (concepts, situations, events, etc.)
  - Arcs express relations (links) between objects
- These relations are of two types :
  - The "is-a-kind-of" relation between concepts
    Example : the concept Chair is-a-kind-of Furniture
  - The "is-a" relation between a concept and an individual
    Example : a chair is-a Seat
- The graphical representation of semantic networks makes them easily interpretable by humans and usable by machines.

1.15.2. Semantic Networks: Example

An illustrative example presents a relational graph describing two individuals and a hierarchy of concepts, reflecting the meaning of several similar sentences involving a tiger and its offspring. IS-A relationships and other logical links allow the deduction of properties and relationships between concepts and individuals. The thick lines in the graph indicate the

primary relationships, and the combined structure enables the inference of information about composite concepts and their interactions, thereby facilitating automatic semantic analysis and conceptual navigation.



1.15.3. Frames

Introduced by Marvin Minsky in 1974, are hierarchical data structures designed to represent stereotypical situations. They serve as an alternative to semantic networks and have inspired many ontology formalisms and languages. Each frame contains attributes called slots, with facets defining possible values, and relations of specialization/generalization.

- Introduced by Marvin Minsky in 1974, Frames correspond to hierarchical data structures used to represent a stereotypical situation.
- Serve as an alternative to knowledge representation based on semantic networks.
- Inspired most of the work on ontology formalisms and definition languages.
- Frames include :
  1. A set of attributes (properties) called *slots*, defining a data structure.
  2. Facets that semantically describe possible values for each *slot*.
  3. Relations of specialization/generalization type.
- Gather relevant knowledge of a given situation into a single object called a Frame, instead of distributing it across multiple axioms.

1.15.4. Frames/Cycl ontology

Frame consolidate relevant knowledge in a single structure and support both explicit representation and inference through attribute inheritance. The Cycl language was the first ontology language based on frames and was later extended to first-order logic.

- A Frame can be viewed as a network of nodes and relations between nodes (i.e., a semantic network).
- Knowledge supported by Frames can be either explicitly described or inferred. Inferences are mainly performed by exploiting attribute inheritance from parent frames.
- The *Cycl* language was the first ontology language initially based on Frames, later extended to first-order logic in its final version.

1.15.5. The Conceptual Graph (CG) model

Proposed by John F. Sowa in 1976 and later formalized by Chein, Michel, and Mugnier in 1992. It originates from semantic networks and is based on graph theory.

- Derived from semantic networks and based on graph theory.
- Knowledge representation relies on describing concepts and their relationships.
- Semantic knowledge of the application domain is defined through canonical graphs associated with specific types of information to specify the roles that concepts can play, such as :
  - AGT [1]
  - ORIG [2]
  - OBJ [3]
  - LOC [4]
  - DEST [5]

1. Active agent or initiator of an action
2. Origin or source (spatial or abstract) of an event
3. Object, entity affected by an action
4. Location where an event takes place
5. Destination or outcome of an event

Conceptual graph relations label nodes and organize them hierarchically. Constraints—positive (obligations) or negative (prohibitions)—enhance expressiveness and are represented as bicolored graphs

- Relations are used to label graph nodes and order them according to a specialization/generalization hierarchy.
- To introduce more expressiveness, the model can be enriched with :
  1. Rules
  2. Constraints
- A positive constraint consists of a condition and an obligation
  Example : *a space must be monitored by at least one camera*
- A negative constraint consists of a condition and a prohibition
  Example : *A person cannot be both sitting and standing at the same time*
- Constraints are represented as bicolored conceptual graphs :
  1. The condition of a constraint consists of the set of 0-colored nodes
  2. Obligation and prohibition constraints consist of the set of 1-colored nodes

1.15.6. GC & Concepts

A concept consists of a type and a referent, defining what the entity is and what it represents. A conceptual graph knowledge base is organized into terminological levels: the support level for concepts and relations, and the assertional level for facts or instances.

Concept Notion

A concept is composed of two elements : its type and its referent.

Example : Defining a Table as an Object

Object : Table, where *Object* represents the type and *Table* is the referent.

Terminological Levels

A knowledge base represented as a conceptual graph includes several terminological levels :

1. The "support" level consists of a set of concepts and relations. These two sets form a hierarchy defining an ordering relation [a].
2. The "assertional" level consists of facts (i.e., individuals/instances)

a. The ordering relation is defined according to the "is-a-kind-of" principle, noted $\leq_C$

1.15.7. GC: The Support Level and the Terminological Level

A support is represented by the quintuple $S = (T_C, T_R, \sigma, \psi, \delta)$, where :

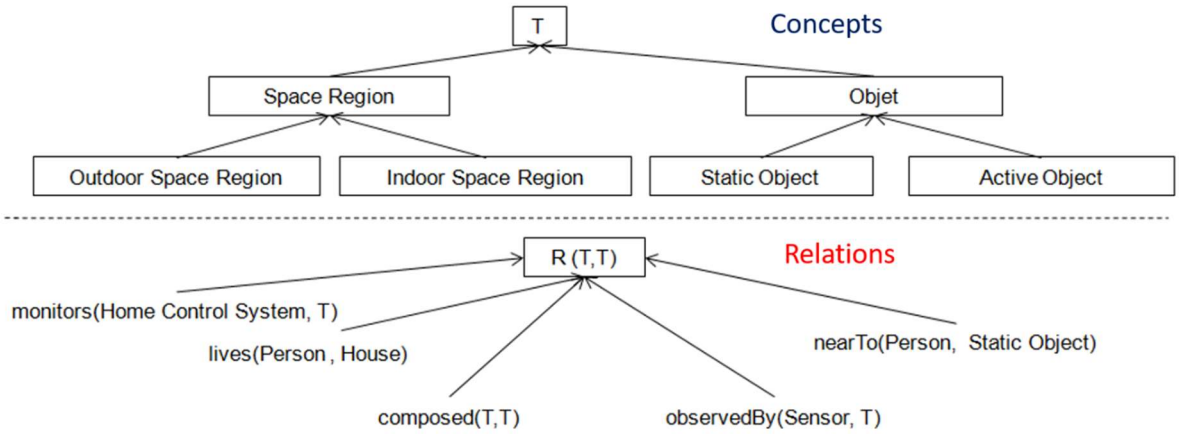
- T_C represents an ordered set of concepts, where the symbol $\top$ denotes the greatest element (i.e., universal type) and the symbol $\bot$ the smallest element.
- T_R represents the set of relation types, partitioned into subsets of relations with the same arity (i.e., having the same number of arguments). These relations, also ordered, are of the "is-a" type.
- σ associates to each relation type t of arity n a signature $\in T_C^n$.
  For example, the signature of the relation *isControlled* (Space, SensorTypes) means that the first argument of the relation *isControlled* is Space and the second argument is of type *SensorTypes*.
- ψ represents the set of individual markers. It is a countable set that also includes a generic marker denoted * to represent an unidentified individual.
- δ is a mapping from ψ to $T_C \setminus \{\bot\}$, called the compliance mapping, which associates each individual with a concept.

GC Example 1

The figure shows an example of a conceptual graph support that includes a set of binary relations. The concept *Indoor_Space_Region* is a type of *Space_Region*; in other words, *Indoor_Space_Region* is subsumed by *Space_Region* (this term comes from philosophy and replaces the notion of generalization/specialization in ontologies. It is the main element for

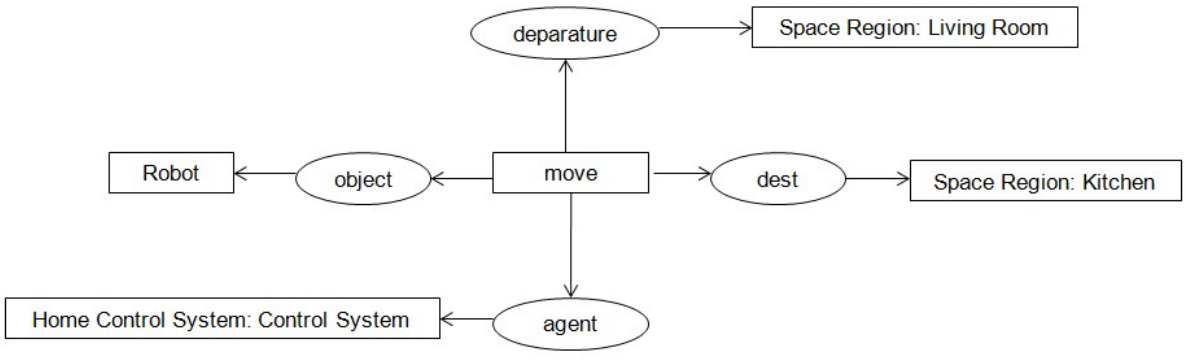
structuring an ontology, often referred to as a "subsumption link". For example, the concept *Human_Being* is a specialization of the concept *Man*).

The relation *nearTo(Person, Static_Object)* has two arguments: the concept *Person* and the concept *Static_Object* within the context of *Space_Region*.



GC: Example 2

In this second example, the goal is to model in a conceptual graph the movement of a Robot from an initial location, *Living_Room*, to a final location, *Kitchen* (is an instance of Space_Region). The individuals *Living_Room* and *Kitchen* are associated with the concept *Space*, while the individual *Control_System* is associated with the concept *Home_Control_System*. The *agent* relation indicates the agent performing the *move* action. The *departure* relation indicates the current position of the robot, the *dest* relation indicates the robot's new destination, and the *object* relation indicates the object to be moved.



Applications of Ontologies

Ontologies, as expressive formalisms, find applications in numerous domains and enable semantic interoperability, providing different systems with the ability to share and use knowledge in a consistent manner:

1.15.8. Semantic Web: intelligent annotation and advanced search.

The Semantic Web represents an extension of the current Web, where information is given a well-defined meaning, allowing computers and humans to collaborate more effectively (Berners-Lee et al., 2001). The vision was that if interactions between users and hypertext could be intuitive enough, the machine-readable information space would accurately reflect people's thoughts, interactions, and work patterns. This would enable powerful automated analysis, revealing patterns in work and supporting collaboration across the typical challenges faced by large organizations.
(***Tim Berners-Lee.*** *1996 and* ***Tim Berners-Lee.*** *2001)*

1.15.9. Internet des objets (IoT) : Enabling seamless interoperability among sensors and systems.

Ontologies provide a formal and structured representation of knowledge in IoT environments. They allow devices, sensors, and systems to share and interpret data consistently, enabling semantic interoperability. By defining concepts, relationships, and constraints, ontologies support reasoning, context-awareness, and intelligent decision-making in applications such as smart homes, healthcare, transportation, and industrial IoT systems.

1.15.10. Healthcare: disease classification and standardization.

Ontologies in Healthcare, such as **SNOMED CT**[16] (Systematized Nomenclature of Medicine – Clinical Terms), enable the standardization of the description of diseases, symptoms, procedures, and treatments. They facilitate the exchange of information between healthcare systems, ensure the consistency of electronic medical records, and support clinical reasoning and decision support applications.

1.15.11. Industry: interoperability in engineering and logistics processes.

Industrial ontologies enable the standardization and harmonization of concepts used in engineering, production, and logistics. They facilitate data exchange between heterogeneous

[16] https://www.snomed.org/

systems, ensure information consistency throughout the value chain, and support process automation and optimization.

1.15.12. Tim Berners-Lee Reinvents the Web

W3C's Mission: "to lead the Web to its full potential" "The social value of the Web is that it enables human communication, commerce, and opportunities to share knowledge [and] to make these benefits available to all people, whatever their hardware, software, network infrastructure, native language, culture, geographical location, or physical or mental ability." Sir Tim Berners-Lee, Inventor of the Web and Founder of W3C

Objectives of SOLID (SOcial LInked Data):

1. Data Protection: Give users full control over their personal data and privacy.
2. Creation of New Standards and Formats: Develop open standards for storing and exchanging decentralized data.
3. Leverage W3C and the Semantic Web**:** Use W3C technologies and recommendations (RDF, SPARQL, ontologies) to ensure interoperability and integration within the existing Web ecosystem.

2. Chapiter 2 XML Schema: The Building Block for Structured Data with URIs

2.1. Introduction

The World Wide Web has profoundly transformed communication and information dissemination, becoming a cornerstone of the emerging knowledge society. Once primarily used for computation, computers are now central to information processing and knowledge exchange. However, today's Web is still largely designed for humans: content, even when automatically generated, often lacks machine-readable semantics. As a result, search engines like Google remain limited—returning imprecise results, overwhelming users with irrelevant content, and requiring manual navigation. The core challenge lies in the fact that the meaning of Web content is not directly accessible to machines. The Semantic Web seeks to overcome this limitation by representing data in a machine-understandable way and enabling intelligent techniques to exploit it. Closely tied to knowledge management—which focuses on acquiring, sharing, and maintaining knowledge as a strategic asset—the Semantic Web faces challenges of integration, standardisation, tool development, and adoption. The following chapters introduce the foundational technologies, including XML, RDF, and OWL, that are essential to achieving this vision.

2.2. Tim Berners-Lee's well-known Semantic Web "layer cake"

illustrates its layered design:

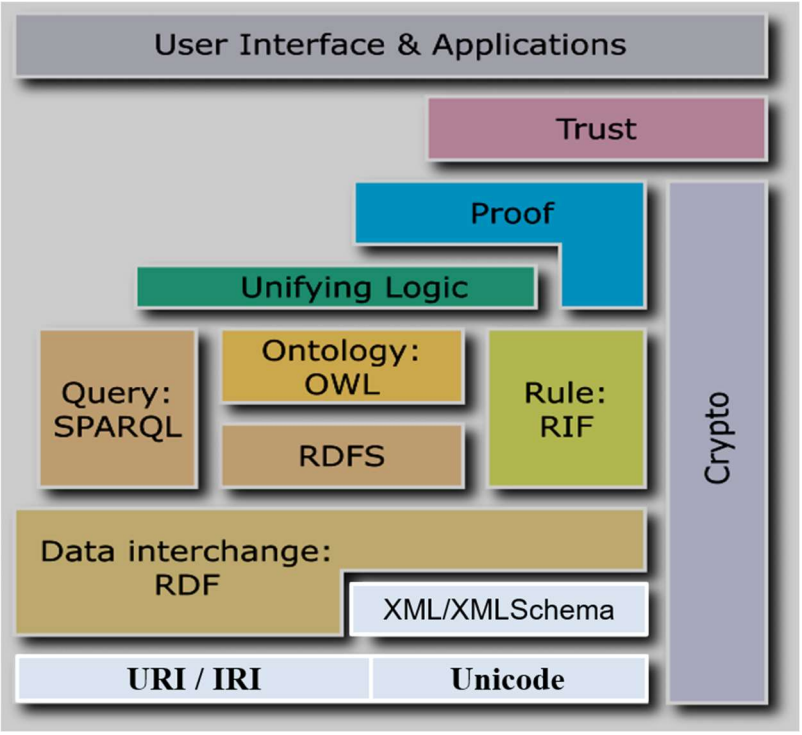
- **XML**: for structuring documents on the Web.
- **RDF**: a basic data model, independent of XML, but often using an XML-based syntax.
- **RDF Schema**: primitive ontology language with classes, properties, subclassing, and domain/range restrictions.
- **Ontologies (OWL, rule-based languages)**: to express richer and more complex relationships.
- **Logic layer**: enables declarative knowledge and reasoning.
- **Proof layer**: manages deduction processes and validation.
- **Trust layer**: built on digital signatures, certification authorities, and reputation systems to ensure reliability and security.
- **URI (Uniform Resource Identifier)**
    - A general character string that uniquely identifies a resource, whether abstract or physical.

- Serves as the foundation for defining other identifiers.

- **URL (Uniform Resource Locator)**
  - A specific type of URI designed to locate a resource on the Web.
  - Combines both the access path and query parameters.
  - Does not guarantee the actual existence of the resource on the server; it only provides an address.

- **IRI (Internationalized Resource Identifier)**
  - An extension of the URI concept that supports international characters (e.g., accented letters, non-Latin scripts).
  - Facilitates the identification and localization of resources in a multilingual and global context.

Key remark: Every URL is a URI, but not every URI is a URL. A URN designates a resource by its name in a persistent and location-independent way. Unlike a URL, it does not indicate how or where to access the resource. For example, urn:isbn: 978-2-07-061275-8 identifies a specific book through its ISBN, but does not provide any information about its online location or availability.



2.3. Bridging the Gap between the Semantic Web and AI

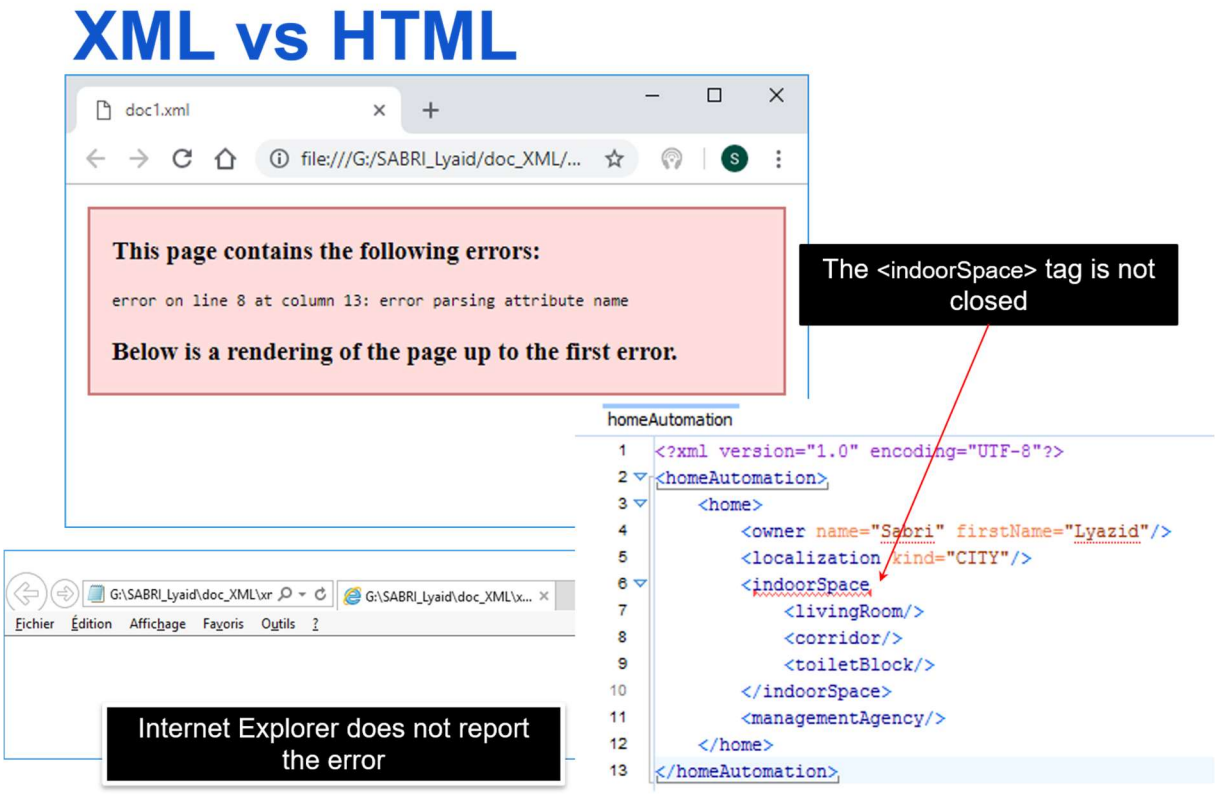
Although an intelligent agent does not have all the reasoning capabilities of a human being, it can already contribute to making the web significantly more efficient than it is today. The goal

differs from that of AI: while artificial intelligence aims to design agents with faculties comparable to, or even superior to, those of humans, the Semantic Web pursues a more pragmatic objective—to support users in their daily online activities. Current AI technologies are already playing a key role in this evolution, and their future advancements will further enhance the potential of the Semantic Web.

2.4. eXtensible Markup Language vs HTML (Background note)

HTML, derived from SGML, is the standard language for creating web pages, but it remains limited because it focuses primarily on the display and not the structure of information. XML, also derived from SGML, was designed to overcome these limitations by separating content and presentation. Unlike HTML, XML (W3C 1998) allows for the definition of its own tags adapted to each domain, making it a metalanguage. This flexibility makes XML documents more machine-readable: the explicit structure and relationships between elements (e.g., an author linked to a book) facilitate automatic data mining, unlike HTML, which requires uncertain inferences.

Thus, XML is widely used as a standard data exchange format between applications and sectors (e.g., MathML, NewsML, BSML (Bioinformatic Sequence Markup Language)), reducing technical costs and promoting interoperability. Where HTML is primarily used for formatting and display, XML provides a structured, extensible, and machine-readable representation.



Important reminder (XML is not an ontology language)
It is only a data structuring language that specifies the syntax and organization of information. Unlike RDF, RDFS, or OWL, it does not provide formal semantics and cannot define concepts, relationships, or reasoning rules."

2.5. XML Schema. W3C Recommendation. May 2001

XML Schema is a language standardized by the W3C that allows to describe the structure and the expected content of an XML document. Unlike DTDs, it is written in XML, which gives it better extensibility and a homogeneous syntax. This allows to specify the elements, the attributes, their hierarchical links and especially the categories of data, whether they are simple (like a number or a date) or complex.

Role and importance

By using XML Schema, we ensure that the XML documents are not only well formed, but also valid, that is to say, they respect precise rules of structure and content. In this way, it promotes cooperation between systems and applications, particularly in areas such as e-commerce, web services and data exchange between organisations. Although it does not specify the semantics of information, XML Schema is crucial for semantic web technologies, in particular RDF and OWL, as it allows to ensure a first level of structuring and validation.

2.5.1. Foundation

An XML Schema document has the .xsd extension, for example sensors.xsd. It is composed primarily of elements and attributes that define the structure and constraints of XML documents.

All XML Schema documents must be declared valid with respect to an official namespace provided by the W3C:http://www.w3.org/2001/XMLSchema.

So, a schema definition usually starts with:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    ..... insert content (in the form of XML tags)...
</xs:schema>
```

Note: XML 1.1 was introduced to align with the Unicode Consortium, which published characters that XML 1.0 could not support. In other words, XML 1.1 allows the use of

additional Unicode characters that were previously invalid in XML 1.0, ensuring better global text compatibility.

Namespace: **ns/xmlns** (pseudo-attribute)

Role:

- Resolve ambiguities
- Ensuring clarity and consistency in defining the XML structure
- Enable merging: prevent name collisions
- Qualify the elements and attributes defined in the schema
- Allow multiple vocabularies within the same document

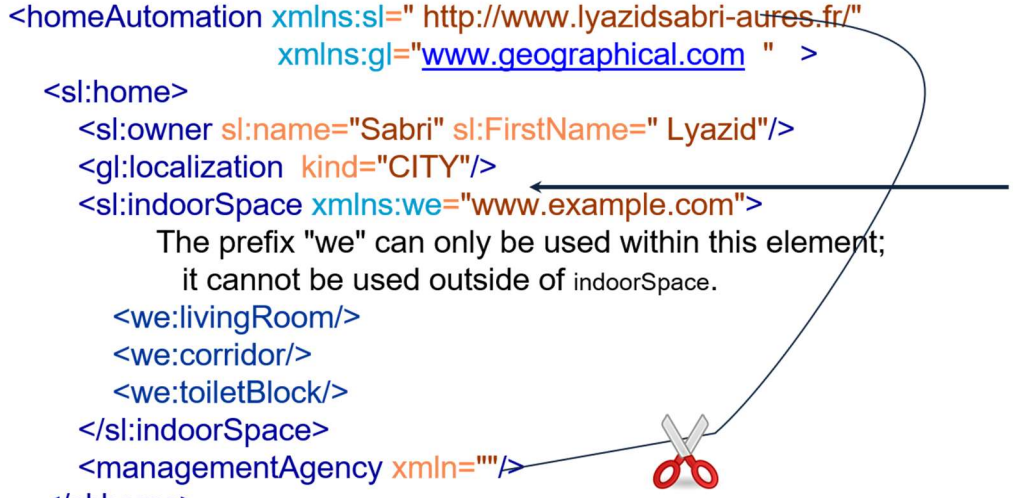
Standard name space :

- ❑ **XML : http://www.w3.org/XML/1998/namespace**
- ❑ **MathML : http://www.w3.org/1998/Math/MathML**
- ❑ **Schémas : http://www.w3.org/2001/XMLSchema**
- ❑ **http://www.w3.org/2001/XMLSchema-instance**
- ❑ **XSLT : http://www.w3.org/1999/XSL/Transform**
- ❑ **DublinCore : http://purl.org/dc/elements/1.1/**

Examples (name space)

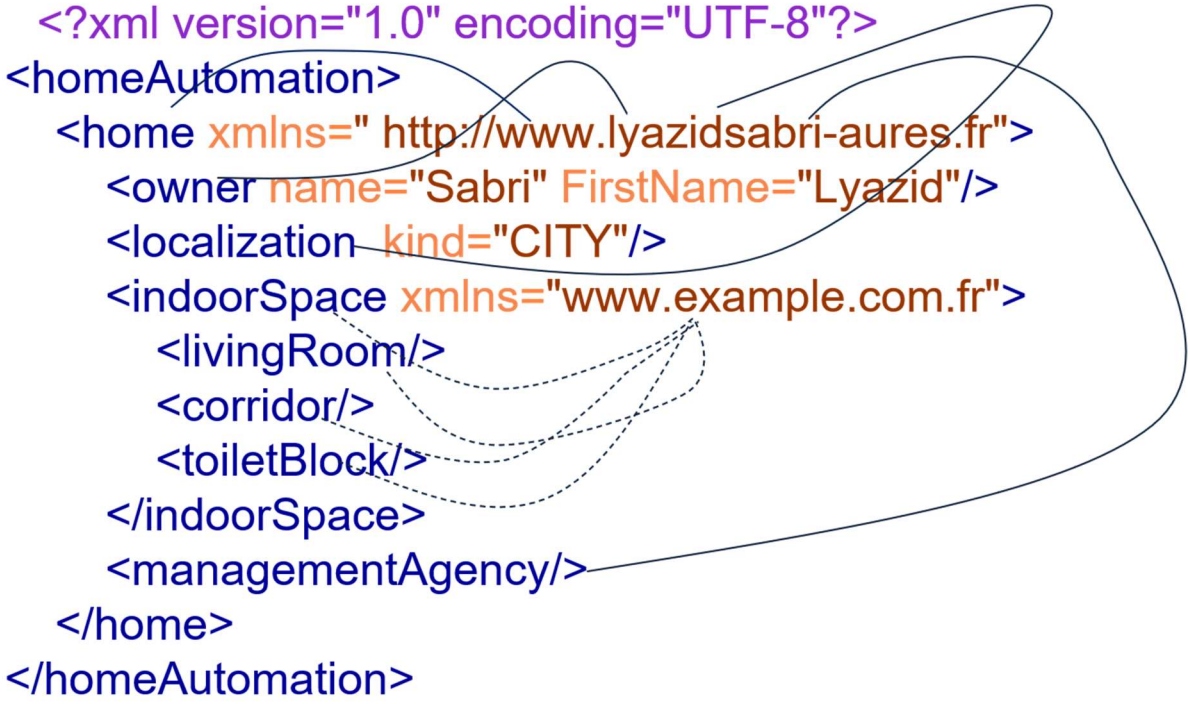
Explicit name space `xmlns:prefix="URI"`

```
<?xml version="1.0" encoding="UTF-8"?>
<homeAutomation xmlns:sl=" http://www.lyazidsabri-aures.fr/"
                xmlns:gl="www.geographical.com  "  >
  <sl:home>
    <sl:owner sl:name="Sabri" sl:FirstName=" Lyazid"/>
    <gl:localization  kind="CITY"/>
    <sl:indoorSpace xmlns:we="www.example.com">
        The prefix "we" can only be used within this element;
          it cannot be used outside of indoorSpace.
      <we:livingRoom/>
      <we:corridor/>
      <we:toiletBlock/>
    </sl:indoorSpace>
    <managementAgency xmln=""/>
  </sl:home>
</homeAutomation>
```



- ❑ The choice of prefix is arbitrary, but avoid xs (for XML) and xsl (for XSLT).
- ❑ Elements are considered identical even if their prefixes are different, as long as they reference the same URL.
- ❑ An element can use multiple prefixes from multiple namespaces.

Namespace. By default
(difficult to manage, applies to the entire document)



2.5.2. Key points to remember:

1. No : in the prefix.
2. The prefix value is a URI.
3. The prefix is used to qualify element names.
4. **Scope of a namespace declaration:**
   Specifies the namespace of an element/attribute.
5. **Default namespace:**
   - Provides flexibility when writing XML documents.
   - The most commonly used namespace should be set as the default namespace.
6. If no default namespace is specified, unqualified elements do not belong to any namespace (allowing compatibility with other documents without namespaces).
7. Attributes are either qualified or never qualified (they do not participate in a default namespace).

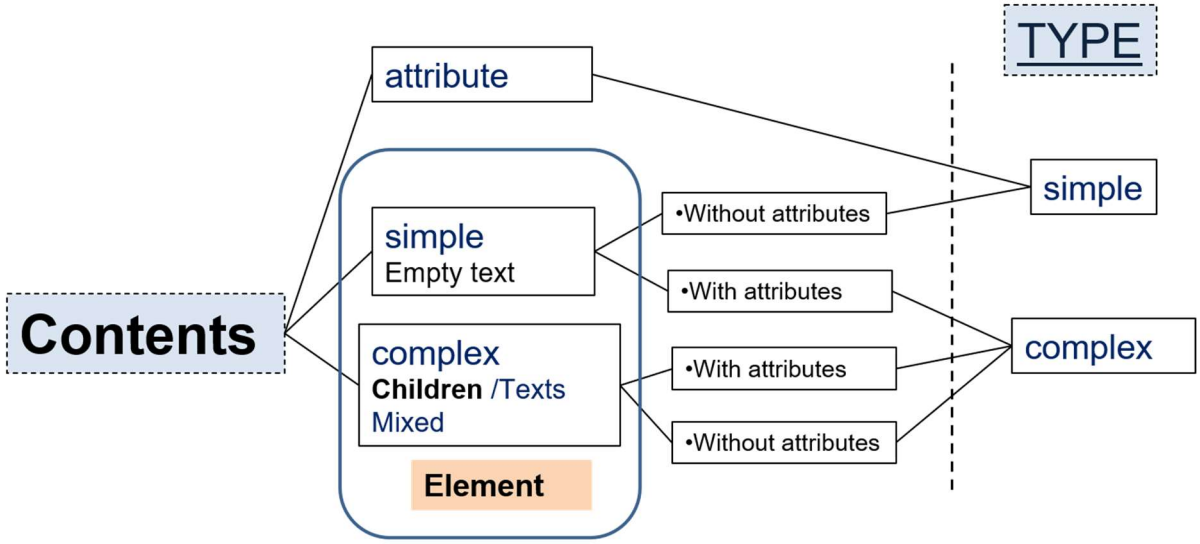
   Caution: The concept of a default namespace does not apply to attributes or text content. Unlike a default namespace, when an element is prefixed, all its child elements must also use the same prefix; otherwise, the namespace will not be applied to them.

2.6. General structure of an XML Schema document

The content of an element can be:

1- empty,
2- text only,
3- have an attribute,
4- contain child elements, or
5- a mixed element (composed of elements and text).

In the last three cases, the element is considered **complex**. It should be noted that the presence of an attribute in a simple element makes it a **complex type** element. Only the definitions of an attribute or an empty element without attributes are of **simple type**. Consequently, an element with no content and no attributes has no type.



[An element without content and without attributes has no type]

2.6.1. Type Definition. Basic Type

The definition of a type is based on predefined types (which are simple, also called basic types) or custom types, such as:

- **xs:string**: A string accepts any value (beware of special characters).
- **xs:normalizedString**: A string where the XML parser removes whitespace (tabs, carriage returns, line feeds).
- **xs:token**: A string with leading and trailing spaces removed.
- **xs:language**: Language codes (e.g., en, fr, ar, de (GER)).

- **xs:NMTOKEN**: No spaces; mostly used for attributes to maintain compatibility with DTD.
- **xs:NMTOKENS**: A list of NMTOKENs separated by spaces.
- **xs:dateTime**: Format CCYY-MM-DDThh:mm:ss (e.g., 2017-10-22T11:45:00).
- **xs:date**: Format CCYY-MM-DD.
- **xs:time**: Format hh:mm:ss.
- **xs:duration**: Expressed as PnYnMnDTnHnMnS (e.g., P1Y5M5DT1S).
- **xs:integer, xs:decimal, xs:float, xs:double**
- **xs:unsignedLong, xs:unsignedInt, xs:unsignedByte**, …
- **xs:long, xs:short, xs:byte, xs:int**
- **xs:nonPositiveInteger, xs:negativeInteger**
- **xs:boolean**: true/false or 1/0.

The definition of a type is based on **predefined types** (which are simple, also called basic types) or **custom types**. XML Schema provides the possibility to create new types from existing simple types. In fact, the user can impose:

1. a **restriction**, for example limiting integer values to a specific range,
2. a **list of values**, or
3. a **combination of types**.

2.6.2. XML Schema: Simple Content Element

As a reminder, an element with no content and no attributes has no type. Therefore, an element without a type should be represented with a self-closing tag. For example:

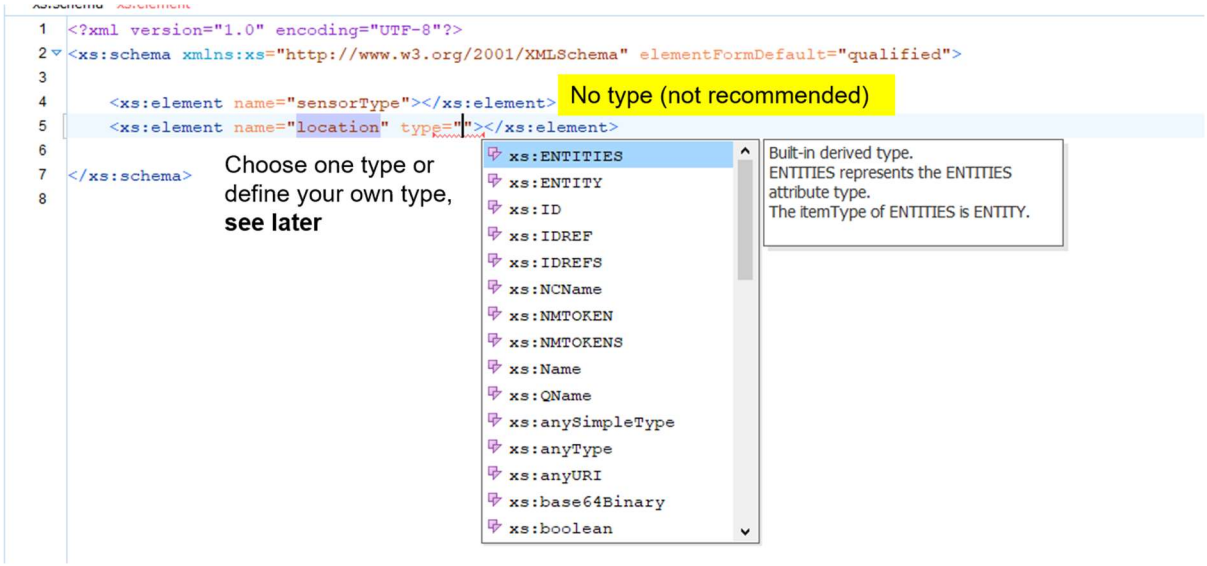
<my_Element/>

The syntax for defining a simple element with a type is as follows:

<**xs:element** **name** = "nom_élément" **type** ="xs:définir_type" />

Be careful: a space is considered a **string type**, so the element is no longer simple.

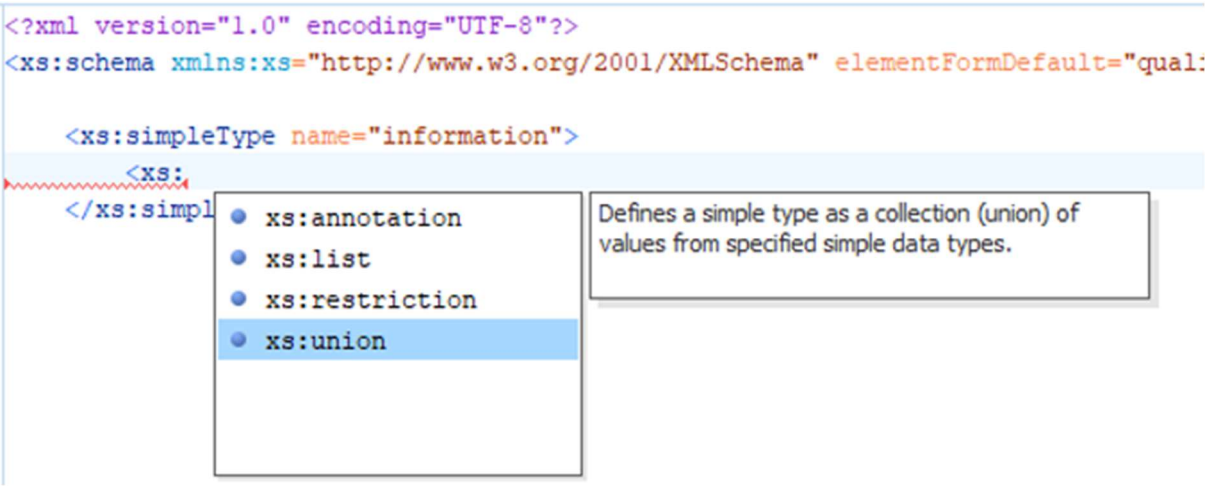
Examples of simple elements:



Lines (4) and (5) above each define an element whose content is simple and without attributes. **Simple type** here means it is either a string, a date, a number, etc. The type of the location element can be a sequence of words. This element has no child elements and no attributes (otherwise it would be a complex type). Therefore, the location element is a **simple element** since its value is of a simple type (basic type).

Regarding the definition of simple content, it is obtained by **extension** or **restriction** of another type. In the following sections, we will first provide a non-exhaustive overview of how to derive a simple type, and then show how to declare the content of a simple element through **extension** and **restriction** of the type.

2.6.3. SimpleType and Union



1. xsd:union allows you to define a **new simple type**.
2. The possible values must be listed in the memberTypes attribute.
3. **Only one of the values** can be used.

Example

In many applications, it may be useful for an element to have **multiple types**. Consider the case where a person can provide either the **postal code** of a city or the **name** of that same

city. For example, the postal code 04000 corresponds to the city **Oum Elbawaghi**. XML Schema provides the xs:union operator to combine multiple types.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<xs:simpleType name="zipCode">
            <xs:restriction base="xs:positiveInteger">
                  <xs:pattern value="[0-9]{5}"></xs:pattern>
            </xs:restriction>
</xs:simpleType>

<xs:simpleType name="zipChar">
       <xs:restriction base="xs:string">
                     <xs:minLength value="3"></xs:minLength>
                     <xs:maxLength value="10"></xs:maxLength>
        </xs:restriction>
</xs:simpleType>

<xs:simpleType name="cityCode">
       <xs:union memberTypes="zipChar    zipCode">
      </xs:union>
</xs:simpleType>

<xs:element name="identite" type="codeVille"/>

<xs:element name="identityBis">
    <xs:complexType>
                <xs:attribute name="zone" type="cityCode"></xs:attribute>
    </xs:complexType>
</xs:element>
</xs:schema>
```

zipCode takes one of the two types.

1. Usage as an Element Type

1. Usage as an Element Attribute

Some xml instances of the xml schema above

```
<xs:element name="identity" type="zipCode"/>
```

```
<identity>34000 BBA</Identity>
<Identity> 34000  </Identity>
<Identity> Bordj Bou </Identity>
<Identity> 45875 25p </Identity>
<Identity> BBA 34000  </Identity>
```

Instance: By mixing digits and letters, this value is of type zipChar (see the previous XSD slide). For example, if you insert two spaces right after BBA, it will be considered a string exceeding 10 characters, and therefore an **ERROR**.
Note: Choosing the correct types carefully will help you **avoid many errors**.

Exercise: Try defining zipChar in the XSD as a type other than string; you will notice the difference.

```
<xs:element name="identityBis">
    <xs:complexType>
        <xs:attribute name="zone" type="cityCode"></xs:attribute>
    </xs:complexType>
</xs:element>
```

Same issue

```
<?xml version="1.0" encoding="UTF-8"?>
<identityBis  zone="4000 OMB"/>
```

2.6.4. Default Value/Fixed Value and Global Elements

When the type is simple, it is possible to assign a **default value** or a **fixed value** to the element, as shown in the following examples. This is done by providing values for the default or fixed attributes of the xsd:element.

```
<xs:element name="location"   fixed="house_125"></xs:element>

<xs:element name="City"   default="BBA"></xs:element>
```

A **global element**, declared as a direct child of the xsd:schema element using xsd:element, can serve as a basis for defining other types. This approach enhances modularity in schema design and helps reduce redundancy.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

    <!-- Global declaration of the element location -->
    <xs:element name="location" type="xs:ID"></xs:element>


    <xsd:complexType ... >
        ...
        <!-- Use of the location element -->
        <xsd:element ref="location"/>
        ...
    </xsd:complexType>


</xs:schema>
```

Note 1: The two attributes name and ref cannot be present simultaneously in the xsd:element.

Note 2: One of them must always be present—either to provide the name of the element being defined or to reference an element that has already been defined.

```xml
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">


  <xs:element name="entity">
 <xs:complexType>
 <xs:attribute name="src" type="xs:anyURI"></xs:attribute>
 </xs:complexType>
 </xs:element>
</xs:schema>
```

Schema (Validator)

```xml
<?xml version="1.0" encoding="UTF-8"?>
 <entite  src="https://www.photo-paysage.com/index.php?cat=19" ></entite>
```

Instance XML

2.6.5. Anonymous Type

When declaring an element, it is possible to explicitly define its type. In this case, the type definition is included directly within the xsd:element. The type is then local (anonymous) and can take one of two forms: a complex type for elements containing multiple child elements or attributes, such as the sensor element, or a simple type for elements containing only a value with possible constraints, such as the temperature measurement.

- An element that has **no name**.
- How to construct it.

❖ **Content within an element, so it is local**

<xs:element name="information"> <xs:simpleType> **Note:** The type must be specified. </xs:simpleType> </xs:element>	<xs:element name="information"> <xs:complexeType> </xs:complexeType> </xs:element>

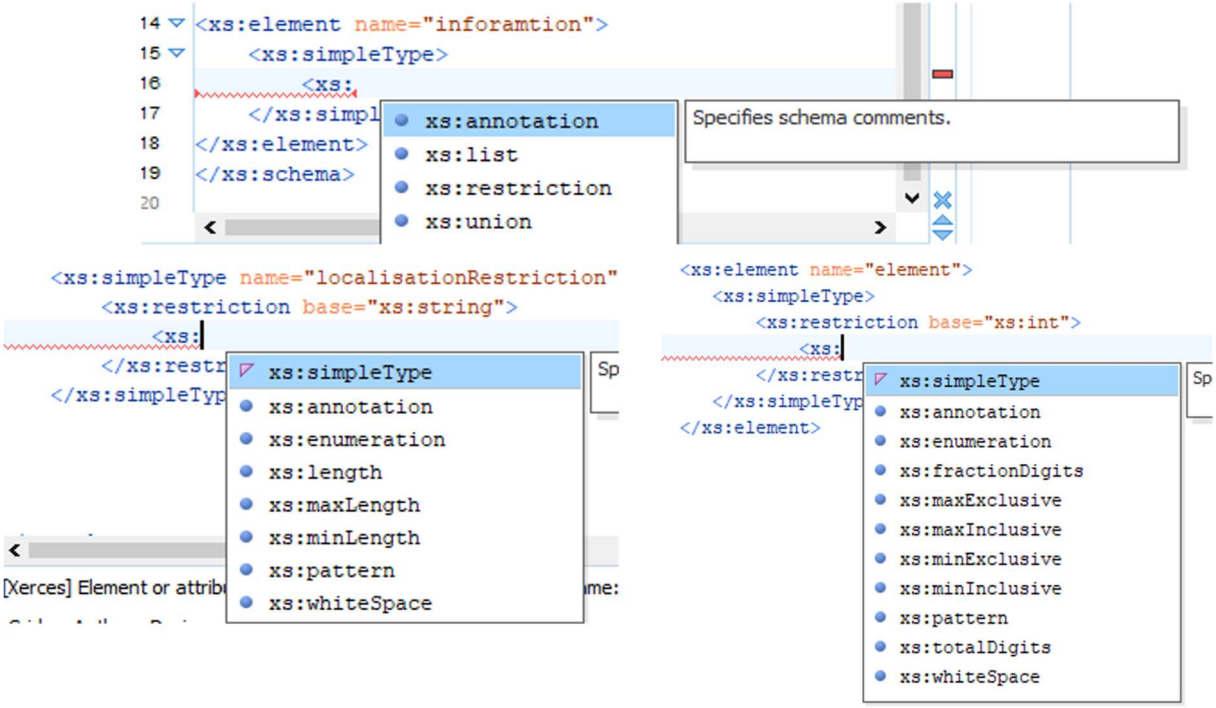
❖ Direct child of xsd:schema, so it is **global** and can be referenced

<xs:simpleType> **Note:** The type must be specified. </xs:simpleType>	<xs:complexeType> </xs:complexeType>

Example 1

•**Mandatory to associate a type**
•No elements and no attributes

Note: Each type has a **restriction field**.



Example 2

The xsd:simpleType element can have a name attribute if the declaration is **global**. The type definition is then provided within the content of the xsd:simpleType element, as shown in the following example.

❖ Restriction on xs:string

```
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:simpleType name="localisationRestriction">
      <xs:restriction base="xs:string">
            <xs:enumeration value="Classe17"></xs:enumeration>
            <xs:enumeration value="Classe16"></xs:enumeration>
            <xs:enumeration value="labo04"></xs:enumeration>
       </xs:restriction>
   </xs:simpleType>
 </xs:schema>
```

Question: Can an instance be created, that is, an XML document can be validated against this schema?

Example 3

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

   <xs:element name="sensor">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="sensorID" type="xs:string"/>
            <xs:element name="sensorType" type="xs:string"/>
            <xs:element name="value" type="xs:decimal"/>
         </xs:sequence>
         <xs:attribute name="unit" type="xs:string"/>
      </xs:complexType>
   </xs:element>

   <xs:element name="temperature">
      <xs:simpleType>
         <xs:restriction base="xs:decimal">
            <xs:minInclusive value="-40"/>
            <xs:maxInclusive value="125"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>

</xs:schema>
```

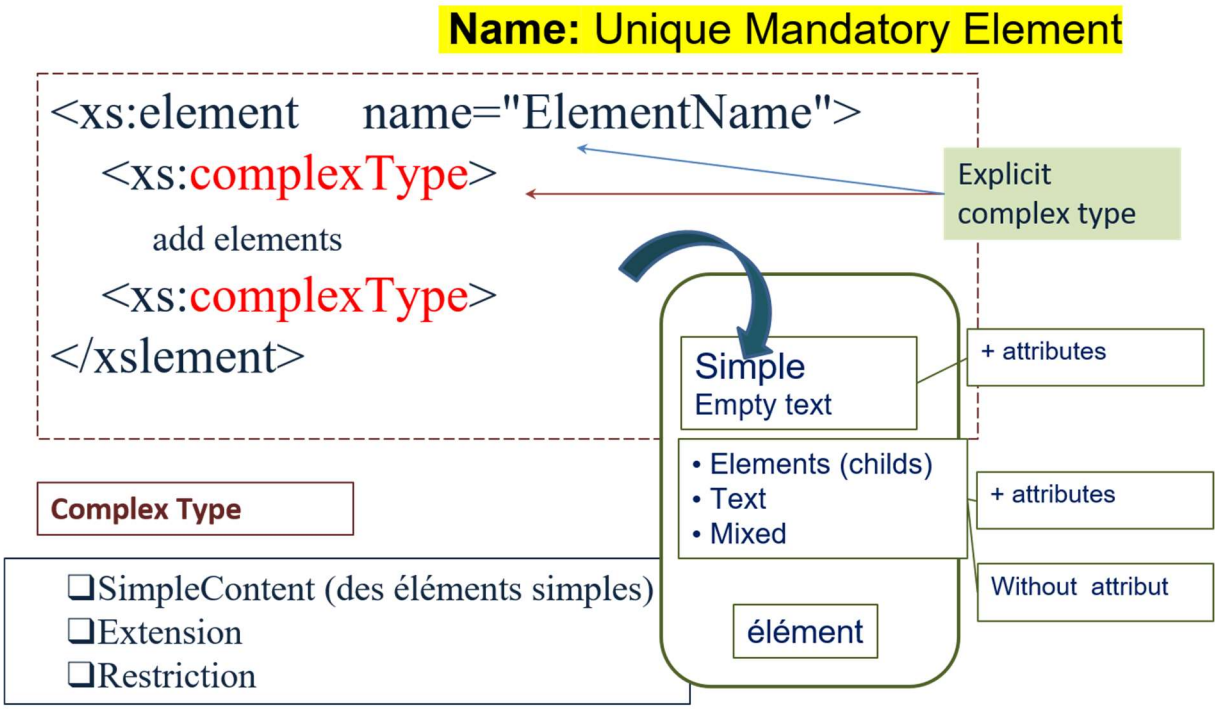
Here we can create an instance, because we defined an element of that type

2.6.6. Complex type

Complex types can define **pure content** (containing only elements), **textual content**, or **mixed content**. All these content types can also include **attributes**. Complex types can only be used for **elements** and are introduced using the xsd:complexType element.

A complex type can be **explicitly constructed** or **derived from another type** via **extension** or **restriction**. When a type is derived, the xsd:complexType element must include either an xsd:simpleContent or xsd:complexContent element to indicate whether the content is purely textual or not. The declaration of a complex type then takes one of the two forms.

Explicit construction of a type is done using operators such as xsd:sequence, xsd:choice, or xsd:all. The type is defined directly within the xsd:complexType element.



2.6.7. Example: XML Schema definition:

<table>
<tr>
<td rowspan="2">

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema

<xs:element name="entite">
<xs:complexType>
</xs:complexType>
</xs:element>

</xs:schema>
```

</td>
<td>

```
<?xml version="1.0" encoding="UTF-8"?>
<entite>

</entite>
```

Error (empty string)

</td>
</tr>
<tr>
<td>

```
<?xml version="1.0" encoding="UTF-8"?>
<entite></entite>
```

Correcte XML document

</td>
</tr>
</table>

Pattern

It is possible to **restrict values** by using the xsd:pattern element, which defines a **regular expression** describing the allowed values of a predefined or previously defined type. The content is considered valid if it matches the regular expression.

In the following example, a simple type ZipCode is explicitly restricted to **5-digit positive integers**, and an element cityOfBirth uses this type:

Explanation:

- ZipCode is a **simple type** derived from xs:positiveInteger.
- The <xs:pattern> restricts values to exactly **5 digits**.
- The element cityOfBirth uses this restricted type, so its content must conform to the 5-digit format.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
   <xs:simpleType name=" ZipCode ">
      <xs:restriction base="xs:positiveInteger">
         <xs:pattern value="[0-9]{5}"></xs:pattern>
      </xs:restriction>
   </xs:simpleType>

 <xs:element name= "cityOfBirth" type="ZipCode"></xs:element>
</xs:schema>
```

```
<cityOfBirth ">
123657
</cityOfBirth >
```

```
<cityOfBirth ">
12367
</cityOfBirth >
```

Regular Expressions: Common Patterns (PATTERN)

- . : A dot matches **any single character**
- [ ] / - : Define **character classes**
- ?, *, +, {} : **Occurrence operators**
- | : **Choice (OR) operator**
- \ : **Escape character**
- \s : A **whitespace character** [\t \n \x0B \f \r]
- \S : Any character **except whitespace**
- \d : A **digit**
- \D : Any character **except a digit**
- \w : An **alphanumeric character** [a-zA-Z_0-9]
- \W : Any character **except alphanumeric** [^\w]
- {n} : **Exactly n occurrences**
- {n,} : **At least n occurrences**
- {n,m} : **Between n and m occurrences** (inclusive)

- In XML Schema, element nesting is controlled using three main operators: `xsd:sequence`, `xsd:choice`, and `xsd:all`, which are used to explicitly construct a

complex type within an `xsd:complexType` element. The `xsd:sequence` operator defines a type as a specific sequence of elements**,** where each element appears **exactly** once by default and must follow the defined order. Elements can be explicit, referenced with `ref`, or recursively constructed using other operators.

- The `xsd:all` operator defines a type in which all listed elements must appear exactly once**,** but in any order. However, its use comes with significant constraints that limit its flexibility. An `xsd:all` element cannot be nested within other constructors like `xsd:sequence`, `xsd:choice`, or another `xsd:all`; its children can only be `xsd:element` elements. Additionally, `xsd:all` must always be a child of `xsd:complexType` or `xsd:complexContent`. The `minOccurs` and `maxOccurs` attributes of elements under `xsd:all` are restricted: `minOccurs` can only be 0 or 1, and `maxOccurs` must be 1 (the default). These attributes can also be applied directly to `xsd:all`, in which case they apply **to** all child elements with the same constraints.
- `xs:choice` is an XML Schema operator used to define a group of elements where only one element from the group can appear in the XML document. It allows mutually exclusive elements. We can specify minOccurs and maxOccurs to control how many times an element can appear. Useful when an element can take **one of several possible types or forms**.
- By combining these operators thoughtfully, XML Schema enables precise control over element order, optionality, and structure while enforcing the rules necessary for valid and well-formed XML documents.

2.6.8. Example sequence

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<xs:element name="Student">
    <xs:complexType>
            <xs:sequence>
                    <xs:element name="StudentNumber" type="xs:ID"/>
                    <xs:element name="level" type="niveau" />
                    <xs:element name="year" type="xs:positiveInteger"/>
            </xs:sequence>
    </xs:complexType>
</xs:element>
```

•Maintain the order of appearance
•No duplicates

```
<?xml version="1.0" encoding="UTF-8"?>
<etudiant>
      < StudentNumber >P</ StudentNumber >
      < level >Master 1</ level >
      < level >Master2 </ level >
      < year>2023</ year>
</etudiant>
```

Example 2 sequence and Name conflict

<xs:sequence>

```
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
 <xs:element name="Student">
     <xs:complexType>
          <xs:sequence>
               <xs:element name="StudentNumber" type="xs:ID"/>
               <xs:element name="level" type="level" />
               <xs:element name="year" type="xs:positiveInteger"/>
          </xs:sequence>
     </xs:complexType>
 </xs:element>

<xs:simpleType name="level">
        <xs:restriction base="xs:string">
               <xs:enumeration value="Master 1"></xs:enumeration>
               <xs:enumeration value="Master2 "></xs:enumeration>
               <xs:enumeration value="Licence"></xs:enumeration>
        </xs:restriction>
 </xs:simpleType>

</xs:schema>
```

<!-- No name conflict in `level` because it is local here -->

Sequence

explicit

```
<xs:element name="smartEnvironment">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="homeAutomation">
                <xs:complexType>
                    <xs:sequence maxOccurs="unbounded">
                        <xs:element name="location"/>
                        <xs:element name="indoorSpace"/>
                        <xs:element name="courtyard"/>
                        <xs:element name="managementAgency"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

ref

```
<xs:complexType name="components">
    <xs:sequence maxOccurs="unbounded">
        <xs:element name="location"/>
        <xs:element name="indoorSpace"/>
        <xs:element name="courtyard"/>
        <xs:element name="managementAgency"/>
    </xs:sequence>
</xs:complexType>

<xs:element name="homeAutomation" type="components"></xs:element>

<xs:element name="smartEnvironment">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="homeAutomation"></xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

- **Structure of an XSD document**
- **Provides better readability**
- **Reference: an attribute or element**
- **The ref attribute can only be used within a local element**

Example xs:all

xs:all

Elements must appear exactly once, in any order

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="CourseType">
    <xs:complexType>
      <xs:all>
        <xs:element name="m1" type="xs:string"/>
        <xs:element name="m2" type="xs:string"/>
        <xs:element name="m3" type="xs:string"/>
      </xs:all>
    </xs:complexType>
</xs:element>
</xs:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
< CourseType >
        <m2></m2>
        <m1></m1>
        <m3></m3>
</ CourseType >
```

Example xs:choice

`xs:choice` (by default, **only one element** can appear among `m1`, `m2`, `m3`)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name ="CourseType">
     <xs:complexType>
                <xs:choice>
                   <xs:element name="m1" type="xs:string"/>
                   <xs:element name="m2" type="xs:string"/>
                   <xs:element name="m3" type="xs:string"/>
                </xs:choice>
     </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<CourseType>
      <m2>fff</m2>
</CourseType>
```

2.6.9. Number of Occurrences in XML Schema

1. Controls how many times an element or a group of elements can appear.
2. minOccurs: specifies the minimum number of occurrences.
    - Default: 1 (the element must appear exactly once if not specified).
3. maxOccurs: specifies the maximum number of occurrences.
    - Default: 1; must always be greater than 0.
    - Can be set to unbounded to allow unlimited occurrences.
4. By default, minOccurs = maxOccurs = 1.
5. If minOccurs > 1, maxOccurs must be explicitly defined.
6. For elements under xs:all, maxOccurs is always 1, and minOccurs can only be 0 or 1.
    - Note: if minOccurs=0 for one element, all other elements must still be provided.
7. Cardinalities applied to group connectors (like xs:sequence, xs:choice, xs:all) generally apply to their child elements, but this is not always guaranteed.

This allows precise control over element repetition and optionality within XML documents.

Example

xs:schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
        <!-- Define a complex type for a sensor reading -->
        <xs:complexType name="SensorReadings">
            <xs:sequence>
                <!-- Temperature readings: at least 1, at most 5 readings -->
                <xs:element name="temperature" type="xs:decimal" minOccurs="1" maxOccurs="5"/>

                <!-- Humidity readings: optional, can appear at most 3 times -->
                <xs:element name="humidity" type="xs:decimal" minOccurs="0" maxOccurs="3"/>

                <!-- Pressure readings: exactly 1 reading -->
                <xs:element name="pressure" type="xs:decimal"/>
            </xs:sequence>
        </xs:complexType>

        <!-- Use the complex type in an element -->
        <xs:element name="sensorData" type="SensorReadings"/>

</xs:schema>
```

2.6.10. Mixed Content & Attribute Declaration in XML Schema

1. An element is considered **mixed** if it contains **text outside of child xs:element tags**.
2. The mixed attribute of xsd:complexType indicates whether an element is mixed:
    - mixed="true" → element can contain both text and child elements
    - mixed="false" → element contains only child elements
3. Attributes are **always of a simple type**.
4. Attributes are declared using <xs:attribute> within the complex type.

1. Attributes are always of a simple type
2. Use <xs:attribute> to declare them within a complex typeUtiliser

```
<xs:attribut  name= "attributeName"   type="xs:typechoice" />
```

Exemple XM schema.

```
<xs:attribut   name="unitofMesire"       type="xs:string"/>
```

2.6.11. Example Mixed xml document

```
<?xml version="1.0" encoding="UTF-8"?>
    <sensordata xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <reading id="1">
                    Temperature alert:
                    <value>28.5</value>
                     °C detected at
                    <timestamp>2025-09-22T14:30:00</timestamp>
         </reading>
</sensordata>
```

1. The <reading> element contains both text ("Temperature alert:" and "°C detected at") and child elements (<value> and <timestamp>), making it mixed content.
2. This models an IoT sensor reading with textual context plus structured data.

The above example XML instance shows how mixed content and attributes can be combined in an IoT sensor data XML document valid against this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified">

  <!-- Define the reading element with mixed content -->
  <xs:element name="reading">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="value" type="xs:decimal"/>
        <xs:element name="timestamp" type="xs:dateTime"/>
      </xs:sequence>
      <!-- Attribute for sensor ID -->
      <xs:attribute name="id" type="xs:positiveInteger" use="required"/>
    </xs:complexType>
  </xs:element>

  <!-- Root element for multiple readings -->
  <xs:element name="sensordata">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="reading" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

2.6.12. Type Extension in XML Schema

Extension consists of adding content and/or attributes to an existing type. Generally, the extended type is not valid for the base type alone, because new elements or attributes may be added that are not compatible with the base.

1. Applicability: Extensions can be applied to both simple and complex types.
2. Result: The resulting type is always a complex type, even if the base type is simple.
3. Syntax:

```
<xs:extension base="...">
    <!-- additional elements or attributes -->
</xs:extension>
```

6. The base type can be:

- Predefined (e.g., <xs:extension base="xs:string">)
- User-defined (custom type defined elsewhere in the schema)

7. Placement: <xs:extension> is a child of either xs:simpleContent or xs:complexContent, which in turn is a child of xs:complexType.

Example extension 1

Explanation:

- SensorReadingType is a base type (could be simple or complex).
- We extend it by adding a new element <location>.
- The resulting type ExtendedSensorReading is complex and suitable for IoT data with extra information.

Note that :

1. SensorReadingType contains:
    a. value → the sensor measurement
    b. timestamp → the time of measurement
2. Attribute unit specifies the measurement unit (e.g., °C, %, etc.).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

    <xs:complexType name="SensorReadingType">
        <xs:sequence>
            <xs:element name="value" type="xs:decimal"/>
            <xs:element name="timestamp" type="xs:dateTime"/>
        </xs:sequence>
        <xs:attribute name="unit" type="xs:string" use="required"/>
    </xs:complexType>

    <xs:complexType name="ExtendedSensorReading">
        <xs:complexContent>
            <xs:extension base="SensorReadingType">
                <xs:sequence>
                    <xs:element name="location" type="xs:string"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:schema>
```

Example extension 2

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

    <xs:complexType name="TemperatureReadingType">
        <xs:simpleContent>
            <xs:extension base="xs:decimal">
                <xs:attribute name="unit" type="xs:string" use="required"/>
                <xs:attribute name="sensorId" type="xs:positiveInteger"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

</xs:schema>

```

Explanation:

Base type is xs:decimal (the sensor value).
xs:extension adds attributes:
unit → measurement unit
sensorId → identifier of the sensor
The resulting type is complex because of the added attributes, even though the base is simple.

Example XML instance:

```xml
<TemperatureReadingType unit="°C" sensorId="101">28.7</TemperatureReadingType>
```

2.6.13. Optional, Required, or Prohibited Attributes in XML Schema

- An attribute is optional by default; it may be present or absent.
- The use attribute of <xs:attribute> allows specifying:
    - optional : attribute may appear (default)
    - required : attribute must appear
    - prohibited : attribute cannot appear
- Notes:
    - optional and required correspond to #IMPLIED and #REQUIRED in DTD attribute declarations.
    - prohibited is useful when restricting a type to disallow an inherited attribute.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
        <xs:complexType name="Example">
            <!-- Pas de séquence d'éléments, uniquement des attributs -->
            <xs:attribute name="lang"  type="xs:string" use="optional"/>
            <xs:attribute name="myId"  type="xs:string" use="required"/>
            <xs:attribute name="indtalledIn" type="xs:string" use="prohibited"/>
        </xs:complexType>
</xs:schema>

```

A complex type cannot be instantiated directly; you must create an element of type complexType as follows: <xs:element name="ele1" type="Example"></xs:element>

To create a valid instance of the XML Schema document, it is required to define a mandatory attribute myId. The attribute lang is optional, meaning it may be omitted. However, the attribute installedIn is prohibited: therefore, if one wants to create an instance of the Example type, this attribute cannot be used or extended.

<ele1 myId="MPL1235"/> correct

<ele1 myId="MPL1235" lang="En"/> correct

<ele1 myId="MPL1235" lang="En" ~~installedIn =""/>~~ this attribute must never appear.

If we extend the above XML Schema with this complex type and an element elt2,

```
12    <xs:complexType name="ExtendedEXample">
13        <xs:complexContent>
14            <xs:extension base="Example">
15                <xs:sequence>
16                    <xs:element name="data" type="xs:string"/>
17                </xs:sequence>
18                <xs:attribute name="value"  type="xs:positiveInteger" ></xs:attribute>
19            </xs:extension>
20        </xs:complexContent>
21    </xs:complexType>
22
23    <xs:element name="el2" type="ExtendedEXample"></xs:element>
24</xs:schema>
25
```

then we can create the following minimal tree:

```
<?xml version="1.0" encoding="UTF-8"?>
<el2  myId="" >
   <data></data>
</el2>
```

only the attribute myId is mandatory, while the other attributes are not required. Moreover, the attribute value is optional by default.

2.6.14. restriction

In XML Schema, a restriction allows you to specialize an existing type by reducing its possibilities (unlike an extension, which adds more).

Restricting a complex type by limiting its attributes and elements.

```
27    <xs:complexType name="RestrictedType">
28        <xs:complexContent>
29            <xs:restriction base="ExtendedEXample">
30                <xs:sequence>
31                    <!-- Keep only elt1, elt2 is removed -->
32                    <xs:element name="data" type="xs:string" minOccurs="0"/>
33                </xs:sequence>
34                <!-- Keep required attribute -->
35                <xs:attribute name="myId" type="xs:ID" use="required"/>
36                <!-- Prohibit the lang attribute -->
37                <xs:attribute name="lang" use="prohibited"/>
38                <xs:attribute name="indtalledIn" use="optional"/>
39            </xs:restriction>
40        </xs:complexContent>
41    </xs:complexType>
42
```

Explanation

- **ExtendedEXample :** contains one element data with minoccurs=0, means that it could not appear.
    - keeps myId as required,
    - prohibits lang (use="prohibited").
    - The modification of installedIn is not permitted.

2.6.15. Groupe and AttributeGroupe

To structure a complex XML schema and promote reuse, XML Schema allows you to define element groups and attribute groups. These mechanisms allow you to factor out common parts and improve the modularity and readability of schemas.

1. An element group is defined with <xsd:group> and a name attribute.
2. It must be a direct child of <xsd:schema>, which gives it global scope.
3. Its contents must be enclosed in an operator of type <xsd:sequence>, <xsd:choice>, or <xsd:all>.
4. It can be reused in the definition of complex types or other groups using the ref attribute.
5. IoT example: grouping the name and location information of a sensor:

Example group

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" xmlns
    <xsd:group name="SensorInfo">
        <xsd:sequence>
            <xsd:element name="sensorName" type="xsd:string"/>
            <xsd:element name="location" type="xsd:string"/>
        </xsd:sequence>
    </xsd:group>
        <xsd:complexType name="TemperatureSensor">
        <xsd:sequence>
            <xsd:group ref="SensorInfo"/>
            <xsd:element name="temperature" type="xsd:decimal"/>
        </xsd:sequence>
    </xsd:complexType>
        <xsd:element name="sensors">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="sensor" type="TemperatureSensor" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

2.7. Example XML document valid according to the IoT schema (TemperatureSensor + SensorInfo):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sensors xmlns:xsi="http://www.w3.org/2001/XMLSchema-instar
    xsi:noNamespaceSchemaLocation="file:/I:/cours%20WS%202(

        <sensor>
            <sensorName>Thermo1</sensorName>
            <location>LivingRoom</location>
            <temperature>22.5</temperature>
        </sensor>
        <sensor>
            <sensorName>Thermo2</sensorName>
            <location>Bedroom</location>
            <temperature>20.0</temperature>
        </sensor>
        <sensor>
            <sensorName>Thermo3</sensorName>
            <location>Kitchen</location>
            <temperature>24.3</temperature>
        </sensor>

</sensors>
```

Explanation:

1. <sensors>: the root element corresponding to the global sensors element.
2. <sensor>: a repeatable element (maxOccurs="unbounded") defined with the TemperatureSensor type.
3. <sensorName> and <location>: included in the SensorInfo group (mandatory elements in this order due to sequence).
4. <temperature>: element defined in TemperatureSensor, type decimal.

This document **respects the order and the type** of each element and would be validated by the XSD.

3. Chapter 3 : Towards Interoperability: Beyond XML with Semantic Web Technologies

3.1. Introduction

XML alone provides no means to add semantic meaning to data, and it imposes no constraints on the choice of terms or vocabulary. As a result, the same information can be represented in multiple ways, making standardization difficult. Furthermore, interpreting the semantic meaning of nested elements is challenging, since the terms carry meaning only for humans; to a machine, they are merely sequences of characters with no inherent significance, and XML offers no mechanism for understanding or reasoning about their content.

To overcome these limitations, a widely adopted expressive standard is needed. Such a standard would facilitate interoperability between software agents, enable the collection and aggregation of distributed information, and allow information to be represented in a graph structure, which supports adding or removing elements without altering the original data. It would also allow new knowledge to be inferred by combining collected data, ensure that queries can be answered without errors or ambiguities, and be usable by technologies that enable machines to actively participate in reasoning and decision-making processes.

3.2. XML and the Challenge of Machine-Understandable Data

XML: Formalism and Semantic Limits

XML as a Universal Metalanguage

XML is a universal markup metalanguage that allows data structures to be described in a standardized way. It is widely used as a formalism for data exchange on the Web, ensuring compatibility and interoperability across heterogeneous systems.

Semantic Limitations of XML

While XML defines the structure of data, it provides no information about the actual meaning of elements. Consider the following example:

Structure 1 :	Structure 2 :
<person> John <helpedby>Kompai</helpedby> </person>	<robot name="Kompai"> <assists> John</assists> </robot>

In this structure, there is no indication that “Kompai” is human. The <helpedby> relationship merely shows that “John” was helped by “Kompai,” without revealing the nature of “Kompai.” Likewise, nothing prevents “Kompai” from being a robot. The <assists> relationship simply indicates that “Kompai” assists “John,” while the semantics of the element depend on the implicit definition of <robot>. Although XML is powerful for structuring data, it does not by itself support automatic inference or reasoning. To address this limitation, several mechanisms can be employed:

Use of Schemas (XML Schema / XSD)

Schemas provide control over structure, data types, and value constraints (e.g., simple types, numeric ranges, regular expression patterns).

They enable static validation but do not allow the derivation of new relationships between elements.

Annotations and Metadata

Attributes or annotations can specify implicit relationships (e.g., role, category, type).

These annotations can guide simple programmatic inferences.

Ontologies and RDF/OWL

For more sophisticated reasoning, XML can be combined with ontologies (OWL, RDF). For example, if an ontology defines <person> as human and <robot> as non-human, a reasoning engine could infer whether Kompai is a person or a robot based on the semantic context.

Limitations of Pure XML-Based Inferences

XML provides syntax and structure, but not semantics.

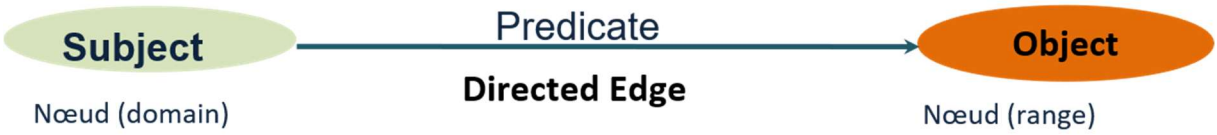
Any meaningful inference requires an external engine (ontology, rules, or AI) capable of interpreting the data.

3.3. RDF Model (Resource Description Framework)

RDF (Resource Description Framework) has been recommended by the W3C since February 22, 1999, with its latest version released on February 25, 2014. It serves as a foundational framework for processing metadata and ensures interoperability between applications. RDF enables information on the Web to be machine-readable, focusing on features that support automated processing of web resources. It can be applied across a wide variety of domains, allowing software agents to use so-called “intelligent” data. Moreover, RDF files with digital signatures are expected to play a crucial role in establishing a “Web of Trust.” Considered an extensible and flexible model, RDF aligns with the overarching objective of the Web by promoting semantic interoperability. https://www.w3.org/TR/rdf-syntax-grammar/

3.3.1. RDF Graph and foundations

An RDF graph is represented as a **directed** graph in which the subject and object are depicted as nodes, while the predicate is represented by a directed edge connecting a subject to an object. The nodes represent the resources being described, and the predicate specifies the type of property. The object, which corresponds to the value of the property (its range), can be either a data value or another resource, with the subject serving as the domain of the property. Additionally, the object can also be a literal or an identifier. Each resource in the graph is uniquely identified by its URI, ensuring consistent reference and interpretation across the Web.



Note: The direction of the arrow is important.

The arc always starts at the subject and points toward the object of the statement.

In RDF, a property has a domain, which defines the class of its subject, and a range, which defines the type of its object.

1. RDF is based on XML.
2. It allows the description of web resources.
3. It relies on the principle of URIs to uniquely identify resources.
4. Information is structured into triples.
5. Each triple consists of a subject, a predicate, and an object.
6. A triple can be named, functioning as a declaration, assertion, or instruction.
7. A **statement** is the basic unit of the RDF model.
8. The collection of triples forms an RDF **graph**.
9. Each node in the graph represents a resource.
10. Only the subject is required to be a resource.

Graph Theory and the Semantic Web

Graph theory, which models **nodes and relationships**, is a fundamental discipline behind the Semantic Web. Its roots go back to 1736, when Leonhard Euler solved the *Seven Bridges of Königsberg* problem, showing mathematically whether a route crossing each bridge once exists—avoiding tedious trial-and-error. This principle of mapping paths and connections directly applies to the Semantic Web, where data and hyperlinks form complex networks. By

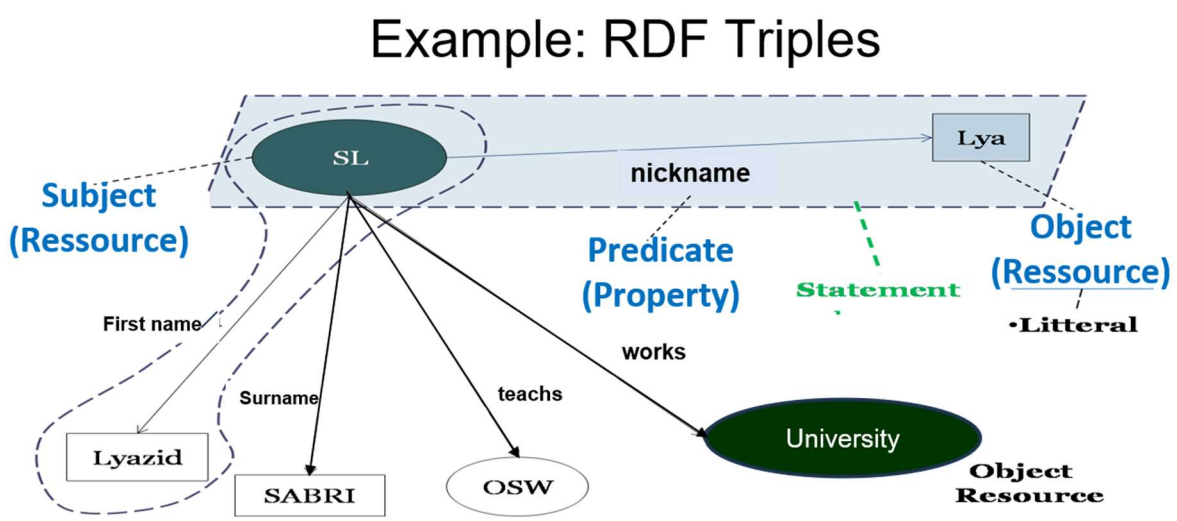
leveraging graph theory, Semantic Web solutions can efficiently organize, traverse, and process information, taking advantage of the mathematical structure of graphs.

RDF: Enabling Machine-Readable Linked Web Data

In RDF, each triple conveys semantic meaning. Unlike XML, there is no inherent hierarchy; information is represented as nodes within a graph, and any relationship can connect these nodes. The model is flexible and independent, allowing multiple RDF graphs to be combined seamlessly. Since we are merging directed graphs and the Web can also be viewed as a graph, it becomes possible to connect web pages and resources directly, enabling the integration of distributed information and supporting reasoning over interconnected data while maintaining semantic clarity.

3.4. Core Structure of the Data Model: Resources, Properties, Statements

In RDF diagrams, literals are usually represented by rectangles. They correspond to concrete values such as strings, numbers, or dates, and are always linked to a datatype, which ensures proper processing by machines. Unlike resources, literals can only occur as objects in triples ⟨Subject, Predicate, Object⟩, never as subjects or predicates. A typed literal associates a data item (e.g., a number or date) with a datatype identified by a URI, most often from XML Schema to ensure interoperability (see RDF 1.1 Concepts). In RDF, resources (subjects and objects) denote entities from either the real or virtual world. Each resource must be globally identified with a URI, allowing description even without direct Web access. Predicates are also resources identified by URIs. URIs guarantee uniqueness, whereas URLs are a subset of URIs that both identify and locate entities on the Web (using '/' or '#' separators).



In the RDF model, all entities described by RDF expressions are called resources. A resource may correspond to an entire web page (e.g., an HTML document), a specific part of a page (such as an HTML or XML element), a collection of pages (such as a complete website), or even an object not directly accessible on the Web, like a printed book.

Each resource is uniquely identified by a URI, optionally extended with anchors, which allows virtually any entity to be assigned an identifier.

Properties describe resources by expressing their aspects, attributes, or relationships. Each property has a well-defined meaning, specifies the types of resources it applies to, the range of possible values, and its relation to other properties.

An RDF statement then combines a resource, a property, and a value into a triple of the form subject–predicate–object. The object may be either a resource, identified by a URI, or a literal value such as a string, number, or date. Literals may contain XML markup, though RDF imposes certain syntactic restrictions on how this markup is used.

1. Objects and subjects denote entities from the real or virtual world.
2. The RDF standard requires that a resource must be global and represented by a URI.
3. A URI makes it possible to describe an entity even if it is not directly accessible on the Web.
4. A predicate can itself be a resource, described by a URI.
5. A URI guarantees the uniqueness of a resource and access to that resource.
6. A URL is used to identify and locate entities on the Web (using separators such as “/” or “#”).

3.4.1. Anonymous resource (blank node)

In RDF, a **blank node** is used to represent an unnamed resource. It can serve as a container to group together several related pieces of information or resources, without assigning a URI to that node. A blank node (it refers to a resource whose name is unknown or unspecified).

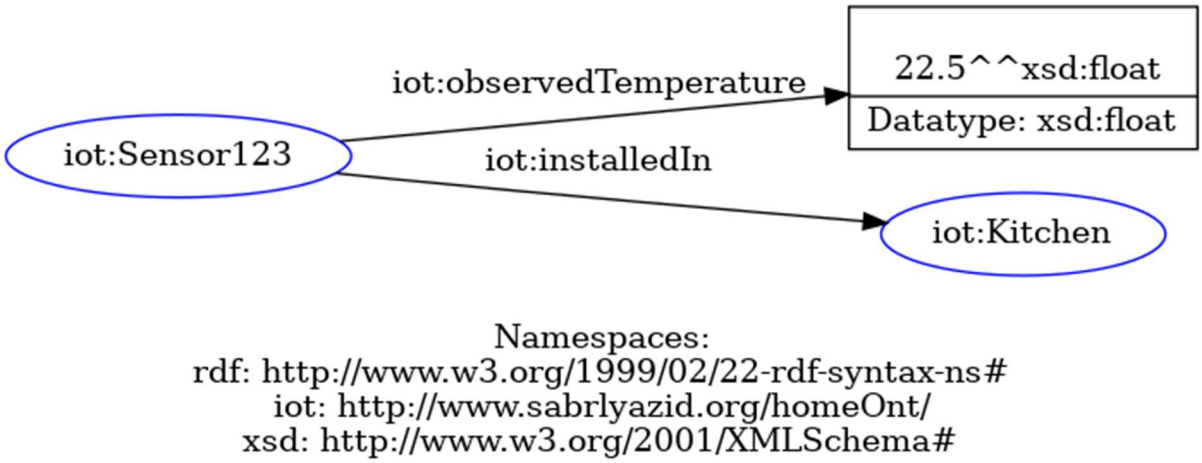
RDF/XML Structure and Triplet Descriptions

1. A **subject** can be either a blank node or a URI.
2. An **object** can be a literal, a blank node, or a URI.
3. RDF defines a **namespace**: http://www.w3.org/1999/02/22-rdf-syntax-ns#
4. An RDF/XML document must declare a **single root element**.

The **XML prologue** is the opening declaration of an RDF/XML document. It specifies the XML version and the character encoding used in the file

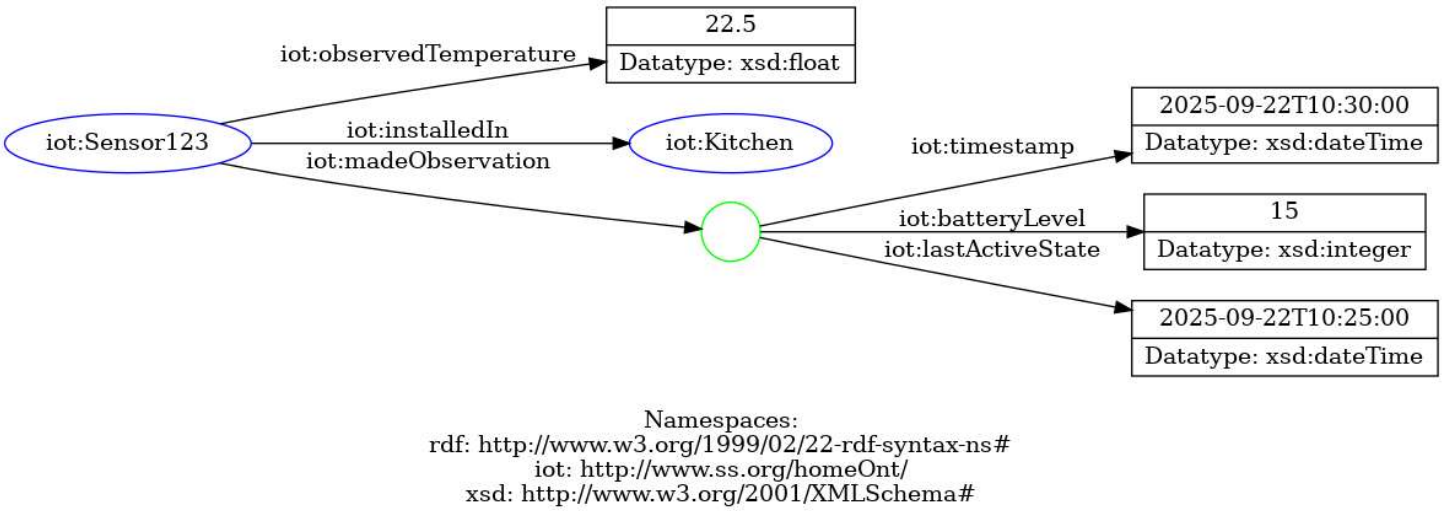
Example 1

The graph expresses that **the sensor *Sensor123*, installed in the kitchen, has measured a temperature of 22.5 °C**.



Example 2 (blank node)

Let us consider the case where we want to provide more details: the sensor *Sensor123*, installed in the kitchen, measured a temperature of 22.5 °C on September 22, 2025, at 10:30, with a battery level of 15% and a last activation recorded at 10:25. To represent this, we need to design another structure that explicitly captures the measurement time, the sensor's last activation time, and its battery level. These three pieces of information will be grouped together and referenced by means of a blank node.



When we say that a blank node is represented by a green circle in an RDF graph, it simply means:

1. It is a node without a URI (no global unique identifier),
2. Used to group together related information (in this case: measurement time, last activation time, and battery level),

3.5. Overview

The basic structure of any RDF expression is built on **triples** (subject, predicate, object).

- A set of triples forms an **RDF graph**.
- **Subject**: can be a URI or a blank node (i.e., a resource without an explicit name).
- **Predicate**: represents a property and must always be a URI.
- **Object**: can be a URI, a blank node, or a literal (value).
- **URIs** are used to clearly identify the resources described in an RDF graph:
    - They remove ambiguity in designations.
    - They allow multiple applications to share the same vocabulary.

    - They prevent naming conflicts.
- A **blank node** is a node that is neither a URI nor a literal.

3.6. RDF Graph Serialization

Serializing an RDF graph means representing its nodes and edges (triples) in a **concrete syntax** that can be stored, exchanged, or processed by machines. Common RDF serialization formats include:

- RDF/XML – the XML-based standard for expressing RDF graphs.
- Turtle (TTL) – a compact and human-readable syntax.
- N-Triples – a line-based plain text format for storing individual triples.
- JSON-LD – RDF expressed in JSON for web applications.

The W3C defines several syntaxes for serializing an RDF model:

1. RDF/XML is officially adopted as the standard representation of an RDF graph.
2. It is easier to process and manipulate by software agents.
3. Other formats, such as N-Triples, Notation3 (N3), and Turtle, are more readable for users who are not familiar with XML.

In RDF serialization:

- Each triple consists of a subject, a predicate, and an object.
- The subject can be a URI or a blank node.
- The object can be a URI, a blank node, or a literal.
- Serialization preserves the structure of the RDF graph, allowing the information to be reconstructed by RDF processors.

Example

The graph expressed in the example 1 above that **the sensor *Sensor123*, installed in the kitchen, has measured a temperature of 22.5 °C**. can be serialized in RDF/XML as follow:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:iot="http://www.sabrlyazid.org/homeOnt/">

    <!-- IoT Sensor -->
    <rdf:Description rdf:about="http://www.sabrlyazid.org/homeOnt/Sensor123">

        <!-- Measured value: typed literal -->
        <iot:observedTemperature rdf:datatype="http://www.w3.org/2001/XMLSchema#float">
            22.5
        </iot:observedTemperature >

        <!-- Location: resource -->
        <iot:installedIn rdf:resource="http://www.sabrlyazid.org/homeOnt/Kitchen"/>

    </rdf:Description>

</rdf:RDF>
```

A RDF/XML concerning the example 2, where the blank node is represented:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:iot="http://www.ss.org/homeOnt/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
    <!-- IoT Sensor -->
  <rdf:Description rdf:about="http://www.ss.org/homeOnt/Sensor123">
     <!-- Measured value: typed literal -->
    <iot:observedTemperature rdf:datatype="xsd:float">22.5</iot:observedTemperature>
        <!-- Location: resource -->
    <iot:installedIn rdf:resource="http://www.ss.org/homeOnt/Kitchen"/>
       <!-- Blank node for an observation -->
    <iot:madeObservation>
      <rdf:Description>
        <iot:timestamp rdf:datatype="xsd:dateTime">2025-09-22T10:30:00</iot:timestamp>
        <iot:batteryLevel rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">15</iot:batteryLevel>
       <iot:lastActiveState rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2025-09-
22T10:25:00</iot:lastActiveState>
      </rdf:Description>
    </iot:madeObservation>
  </rdf:Description>
</rdf:RDF>
```

Or you can use the following website: https://www.ldf.fi/service/rdf-grapher or https://www.w3.org/RDF/Validator/ simply copy and paste the text and execute it. To better use both sites, refer to **Lab 1** dedicated to this topic.

3.6.1. The same file serialized with N-Triples

<http://www.ss.org/homeOnt/Sensor123>
<http://www.ss.org/homeOnt/observedTemperature> "22.5"^^<xsd:float> .
<http://www.ss.org/homeOnt/Sensor123> <http://www.ss.org/homeOnt/installedIn>
<http://www.ss.org/homeOnt/Kitchen> .
<http://www.ss.org/homeOnt/Sensor123>
<http://www.ss.org/homeOnt/madeObservation> _:genid1 .
_:genid1 <http://www.ss.org/homeOnt/timestamp> "2025-09-
22T10:30:00"^^<xsd:dateTime> .
_:genid1 <http://www.ss.org/homeOnt/batteryLevel>
"15"^^<http://www.w3.org/2001/XMLSchema#integer> .
_:genid1 <http://www.ss.org/homeOnt/lastActiveState> "2025-09-
22T10:25:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .

For information see: https://www.w3.org/TR/n-triples/

The same file serialized with Turtle (TTL)

@prefix homeOnt: <http://www.ss.org/homeOnt/> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://www.ss.org/homeOnt/Sensor123>

homeOnt:observedTemperature "22.5"^^xsd:float ;

homeOnt:installedIn ns0:Kitchen ;

homeOnt:madeObservation [

homeOnt:timestamp "2025-09-22T10:30:00"^^xsd:dateTime ;

homeOnt:batteryLevel 15 ;

homeOnt:lastActiveState "2025-09-22T10:25:00"^^xsd:dateTime

] .

For information see: https://www.w3.org/TR/turtle/

3.7. From Data to RDF: Creating Your RDF Graph

In this section, we explore how to create RDF data files to represent knowledge as triples—consisting of a subject, predicate, and object—allowing machines to understand and process data. We will cover the basic structure of RDF files, and practical examples for constructing

RDF graphs. By the end of this section, readers will be able to author RDF files that describe resources, relationships, and data values in a machine-readable and interoperable way.

3.7.1. rdf:Description + rdf:about

The <rdf:Description> element is the central component of RDF/XML. It is used to describe a specific resource in the RDF graph and is typically associated with the rdf:about attribute, which identifies the URI of the resource being described. All properties and values related to that resource are nested inside this element, forming the subject-predicate-object structure of RDF triples.

Creating a triple to describe the resource **Ly** involves specifying a subject, a predicate, and an object. In RDF/XML, the subject is represented by the resource **Ly**, the predicate defines the property or relationship, and the object provides the value or another resource. This forms a complete RDF statement that can be processed by machines to represent semantic information.

1. The prologue indicates that the document is in XML format.
2. The root element, introduced by rdf:RDF, indicates that the document represents an RDF graph.
3. The namespace of this model is defined by the attribute xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#".
4. The resource to be described in this document is introduced by rdf:Description.
5. The rdf:about attribute specifies the URI of the resource (Ly) to be described.
6. Namespaces prefixed by `homeOnt` and `foaf` will be used to delimit vocabularies not belonging to the RDF vocabulary.
7. The value of the `foaf:firstname` property is a literal.
8. `rdf:about`, `foaf:name`, and `Lyazid` represent, respectively, the Subject, Predicate, and Object in the RDF graph described by the following RDF/XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:homeOnt="http://www.sabrilyazid.org/homeOnt#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/" >

 <rdf:Description   rdf:about="http://www.sabrilyazid.org/homeOnt#Ly">
            <foaf:name>Lyazid</foaf:name>
</rdf:Description>
```

3.7.2. RDF:description with rdf:ID (another alternative)

An alternative way to describe a resource is by using **rdf:ID** in <rdf:Description>:

1. Use rdf:ID instead of rdf:about.
2. **When the resource doesn't need global visibility, rdf:ID keeps the RDF clean and readable.**
3. The full URI of a resource declared with rdf:ID is formed by concatenating the URI declared in the xml:base attribute and appending the # character followed by the identifier.
4. The rdf:ID attribute specifies an identifier for the resource currently being described. rdf:ID is scoped to the current RDF/XML document. It cannot reference a resource outside the document because:
   - **Local identifier:** rdf:ID defines a local name for a resource within the current RDF/XML file. It is not a full URI by itself.
   - **URI construction:** The complete URI of the resource is formed by concatenating the document's base URI (xml:base) with the # character and the value of rdf:ID. For example:

```
<rdf:RDF xml:base="http://www.sabrilyazid.org/homeOnt/">
 <rdf:Description rdf:ID="Sensor1">
  <iot:observedTemperature rdf:datatype="xsd:float">22.5</iot:observedTemperature>
 </rdf:Description>
</rdf:RDF>
```

The full URI of this resource becomes:

http://www.sabrilyazid.org/homeOnt#Sensor1

5. **Document-local scope:** Because rdf:ID depends on the current document's xml:base, it cannot reliably refer to a resource defined in another RDF/XML file. If you need a globally referenceable resource, you should use rdf:about with a full URI instead.

Caution

Using rdf:ID makes the resource local to the document, which means:

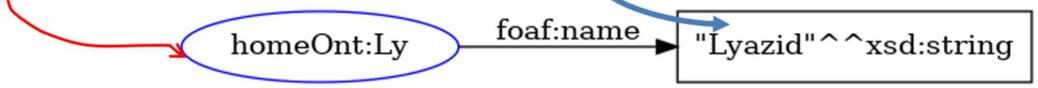
1. No global uniqueness: Other RDF documents cannot reference this resource because the full URI is implicitly tied to the document's xml:base.
2. Limited sharing: If multiple documents need to refer to the same entity (e.g., a sensor, a person), using rdf:ID prevents this. Only rdf:about with a full URI ensures cross-document references, enabling true sharing of resources.

So, while rdf:ID is convenient for small, self-contained RDF/XML files, it loses the main RDF objective of creating a globally interoperable graph.

3.7.3. Using xsd (XML Schema data types)

For a literal allows specifying the type of the value, but this is not always the best way to represent structured XML content. For more complex or structured data, it is recommended to use rdf:value, which allows embedding structured content while maintaining RDF compliance (see section: “Structured Value with rdf:value”). Therefore, the best manner to describe the resource Ly is as follow:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:homeOnt="http://www.sabrilyazid.org/homeOnt#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/">
   <rdf:Description   rdf:about="http://www.sabrilyazid.org/homeOnt#Ly ">
     <foaf:name> "Lyazid"^^xsd:string</foaf:name>
   </rdf:Description>
</rdf:RDF>
```



Namespaces:
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
homeOnt: http://www.sabrilyazid.org/homeOnt#
xsd: http://www.w3.org/2001/XMLSchema#
foaf: http://xmlns.com/foaf/0.1/

3.7.4. rdf:resource

Allows referencing a value of type *resource*. It is used to indicate that the object of an RDF triple is another resource identified by a URI, rather than a literal value.

In this example, KM is a robot and Jh is a person, with KM assisting Jh. Note that each resource is represented by an oval in the RDF graph, rather than as a literal.

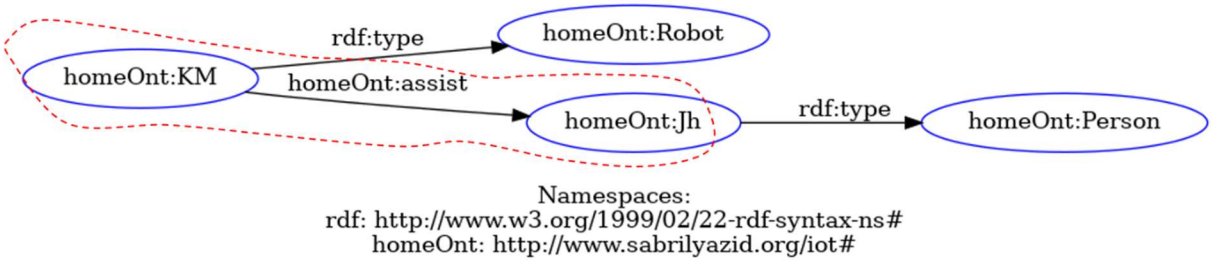
```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:homeOnt="http://www.sabrilyazid.org/iot#">

    <!-- Robot KM -->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#KM">
        <rdf:type rdf:resource="http://www.sabrilyazid.org/iot#Robot"/>
        <homeOnt:assist rdf:resource="http://www.sabrilyazid.org/iot#Jh"/>
    </rdf:Description>

    <!-- Person Jh -->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#Jh">
        <rdf:type rdf:resource="http://www.sabrilyazid.org/iot#Person"/>
    </rdf:Description>

</rdf:RDF>
```

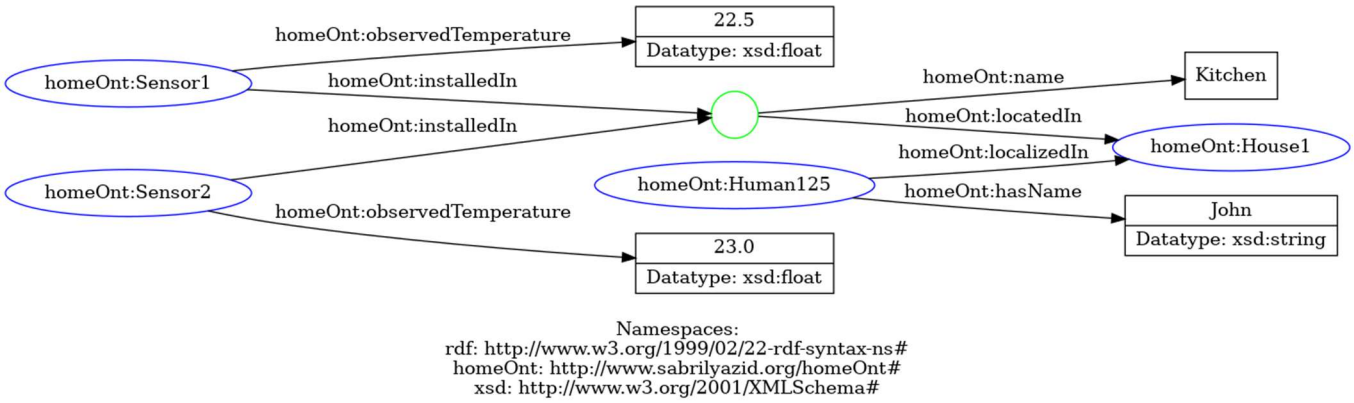
When describing KM, we use `rdf:about` to specify the resource, and likewise for Jh. To link the two descriptions, we use `rdf:resource` to indicate that KM assists Jh.



3.7.5. Blank Node with ID

It should be noted that during serialization, the system assigns a generic identifier to the blank node (see example the section Anonymous resource (blank node) and example))). This identifier is only known within the graph and therefore cannot be referenced from outside the graph. However, <rdf:Description> can include rdf:nodeID. rdf:nodeID is used to identify a blank node, allowing it to be referenced elsewhere in the RDF graph. Moreover, empty nodes (rdf:nodeID): Multiple resources can reference the same internal node. This allows common information, such as an observation, to be shared between multiple sensors without duplicating data.

The statement: Human125, named John, is located in House1. The sensors Sensor1 and Sensor2 are installed in the Kitchen (blank node room1), which is part of House1.
Is represented as follow:



1. Sensor1 and Sensor2 are installed in the same room, represented by the blank node room1.
2. The blank node room1 has properties describing the room, for example iot:name (Kitchen) and iot:locatedIn (House1).
3. An observation, such as the measured temperature, can be shared by both sensors if needed.
4. Using blank nodes allows the same resource to be referenced by multiple sensors without creating duplication in the RDF graph.

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:homeOnt="http://www.sabrilyazid.org/homeOnt#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
       <!-- Senors -->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/homeOnt#Sensor1">
        <homeOnt:observedTemperature rdf:datatype="xsd:float">22.5</homeOnt:observedTemperature>
        <homeOnt:installedIn rdf:nodeID="room1"/>
    </rdf:Description>
        <rdf:Description rdf:about="http://www.sabrilyazid.org/homeOnt#Sensor2">
        <homeOnt:observedTemperature rdf:datatype="xsd:float">23.0</homeOnt:observedTemperature>
        <homeOnt:installedIn rdf:nodeID="room1"/>
    </rdf:Description>
        <!-- Empty Room-->
    <rdf:Description rdf:nodeID="room1">
        <homeOnt:name>Kitchen</homeOnt:name>
        <homeOnt:locatedIn rdf:resource="http://www.sabrilyazid.org/homeOnt#House1"/>
    </rdf:Description>
        <!--   House1 -->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/homeOnt#House1"/>
        <!-- Human125 represents John-->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/homeOnt#Human125">
        <homeOnt:hasName rdf:datatype="xsd:string">John</homeOnt:hasName>
        <homeOnt:localizedIn rdf:resource="http://www.sabrilyazid.org/homeOnt#House1"/>
    </rdf:Description>
</rdf:RDF>
```

3.7.6. rdf:datatype & rdf:value

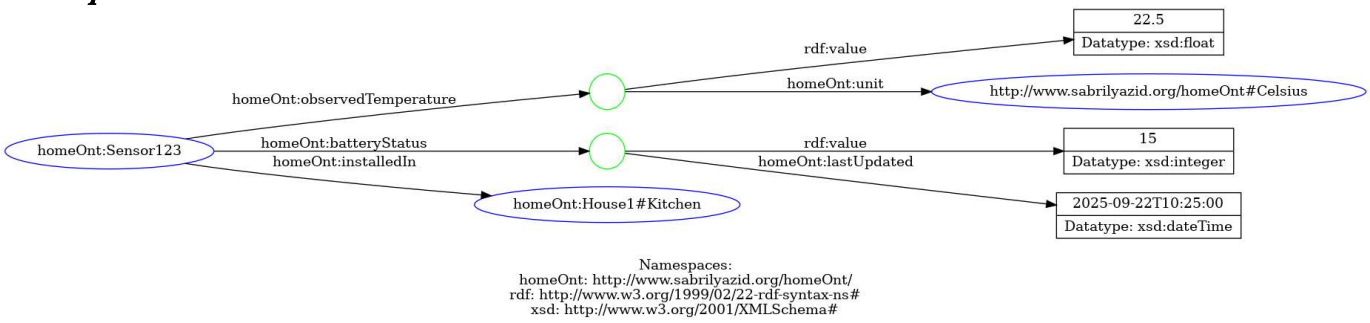
The rdf:value property in RDF is a standard property used to represent the value of a resource when that value doesn't fit naturally into a specific property of a vocabulary. It's particularly useful for encapsulating values within more complex structures, such as observations or measurements, where the value is a key component but additional context is also important.

Purpose of rdf:value

- Encapsulating Values: It allows you to encapsulate a value within a resource, providing a way to associate additional metadata with the value.
- Complex Structures: When representing complex data structures, rdf:value serves as a property to hold the main value, while other properties can provide supplementary information.
- Not a Replacement: `rdf:value` is not a replacement for specific properties within a vocabulary. It's used when no appropriate property exists.
- Standard Property**:** It's a standard property defined in the RDF specification, ensuring consistency across RDF data.

Datatypes in RDF are used with literals to represent concrete values such as strings, numbers, and dates. The datatype abstraction in RDF is compatible with XML Schema. Any datatype that conforms to this abstraction **can be used in RDF**, even if it is not originally defined in XML Schema.

Example



- `rdf:parseType="Resource"` is used for structured literals (temperature, battery status).
- Sensor location is a **resource reference**, not a literal.
- Each structured literal contains an <rdf:value> for the actual data and optionally other properties (unit, lastUpdated).
- This approach keeps the RDF graph clean and fully compliant.

3.7.7. Defining Resource Types with rdf:type in RDF/XML

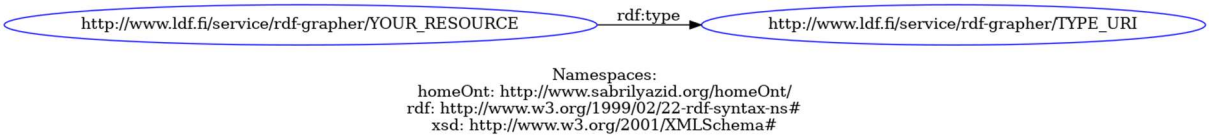
rdf:type is used to state that a resource is an instance of a class. There are two ways to represent this in an RDF graph using RDF/XML:

First way:

<rdf:Description rdf:about="YOUR_RESOURCE">

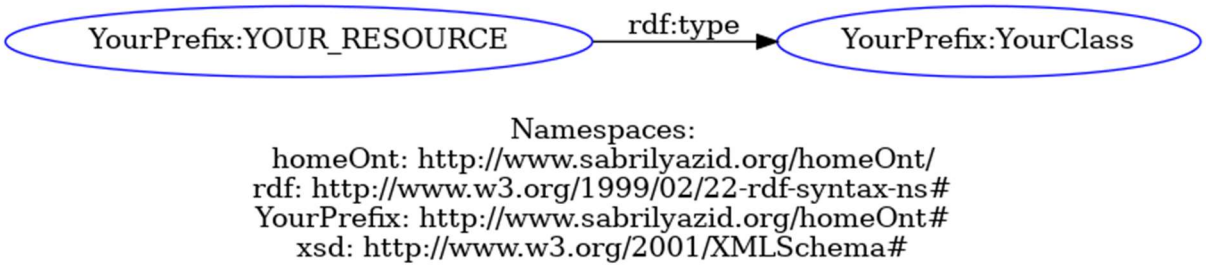
<rdf:type rdf:resource="TYPE_URI"/>

</rdf:Description>



Second way:

<YourPrefix:YourClass rdf:about=" YourPrefix:YOUR_RESOURCE"/>



In both cases, the RDF graph expresses that the resource is an instance of the given class. The first approach uses a generic <rdf:Description> with a <rdf:type> predicate, while the second directly uses the class element as the tag name.

3.7.8. Graph union

In the Semantic Web, RDF descriptions can be spread across multiple documents. For an intelligent agent to use this data, it must combine RDF graphs from different sources. Simply merging their triples is not enough, as blank nodes may share identical identifiers. Therefore, these blank nodes must be systematically renamed before the fusion.

3.7.9. Description of Groups of Resources Using RDF Containers and Collections

RDF provides mechanisms to describe groups of resources, relying on the concept of **containers**. Each element within a container is considered a member, which can be represented either as a resource or a literal. RDF defines three main types of containers:

- **rdf:Bag** – holds a list of elements in no particular order.
- **rdf:Seq** – represents a sequence of elements.

- **rdf:Alt** – indicates alternatives, allowing only a single choice to be selected.

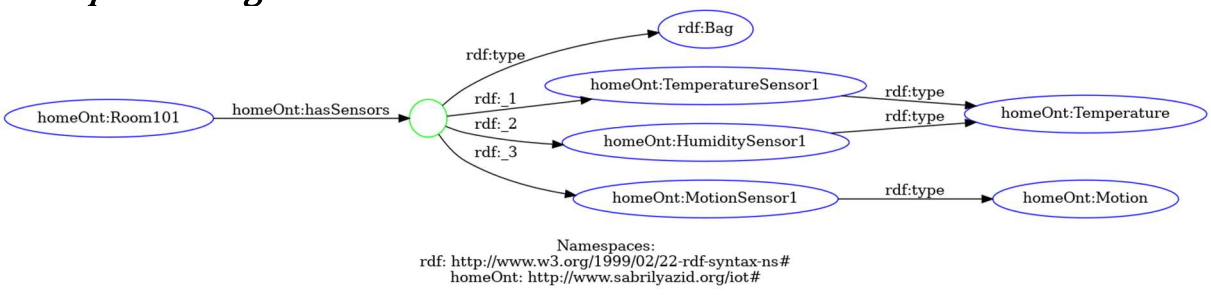
Comments on Containers:

1. rdf:_n is used to access each member of a container.
2. Properties like rdf:_1, rdf:_2, rdf:_3, etc., are automatically generated by the RDF parser to enumerate container members.
3. Applications are responsible for interpreting the sequence correctly.
4. Unlike rdf:Alt, only one member is identified by rdf:_1.
5. While rdf:li facilitates insertion of members in rdf:Bag, the parser uses rdf:_n to enumerate them.
6. There is no strict constraint on numbering.
7. Numbers do not need to be sequential.
8. Duplicate numbers for members are allowed.
9. The same applies to rdf:Seq and rdf:Alt.

Containers vs RDF Collections:

1. Containers allow adding new members in other RDF descriptions.
2. rdf:Collection is considered **closed**.
3. A blank node ensures that no external document can access it.
4. Collections use rdf:parseType="Collection".
5. rdf:rest describes the remaining elements in the list.
6. rdf:nil represents an empty list (end of collection).
7. rdf:first indicates the first member, enabling recursive list creation.
8. The order of elements follows the sequence of their appearance.
9. Collections prohibit adding additional members once defined.

Example rdf:Bag



1. **rdf:Bag**

- Used to represent an unordered collection of members.
- Members are added using <rdf:li> elements.

1. **IoT Context**

- Room101 has multiple sensors.
- Each sensor is identified by a URI.
- The order of sensors does not matter (that's why we use Bag, not Seq).

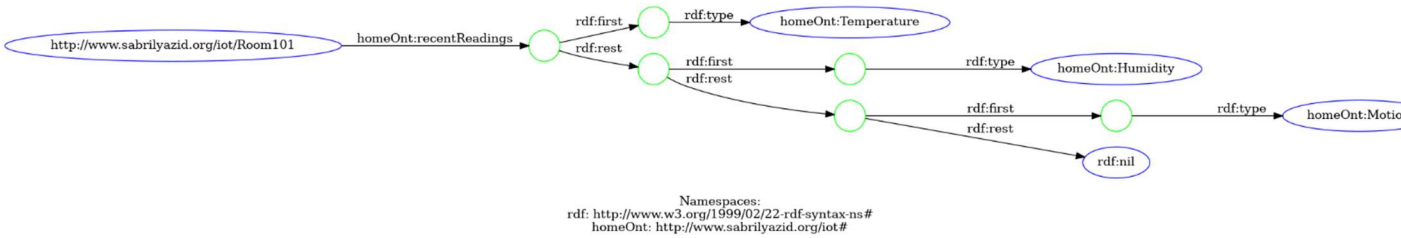
S

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:homeOnt="http://www.sabrilyazid.org/iot#">
        <!-- Description of the room -->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#Room101">
        <homeOnt:hasSensors>
            <!-- Bag of sensors -->
            <rdf:Bag>
                <rdf:li rdf:resource="http://www.sabrilyazid.org/iot#TemperatureSensor1"/>
                <rdf:li rdf:resource="http://www.sabrilyazid.org/iot#HumiditySensor1"/>
                <rdf:li rdf:resource="http://www.sabrilyazid.org/iot#MotionSensor1"/>
            </rdf:Bag>
        </homeOnt:hasSensors>
    </rdf:Description>
        <!-- Optional: describe the sensors themselves -->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#TemperatureSensor1">
        <rdf:type resource="http://www.sabrilyazid.org/iot#Temperature"></rdf:type>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#HumiditySensor1">
        <rdf:type resource="http://www.sabrilyazid.org/iot#Temperature"></rdf:type>
    </rdf:Description>
        <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#MotionSensor1">
            <rdf:type resource="http://www.sabrilyazid.org/iot#Motion"></rdf:type>
    </rdf:Description>
</rdf:RDF>
```

Example rdf:Collection

1. rdf:parseType="Collection"
    - Tells RDF/XML that the child elements form an ordered collection.
    - RDF will interpret this as an RDF list (rdf:first, rdf:rest, etc.) under the hood.
2. IoT Context
    - Room101 has a collection of recent readings.
    - The order of readings is preserved (first temperature, then humidity, then motion).
3. Difference from rdf:Bag
    - rdf:Bag = unordered collection.

- parseType="Collection" = ordered collection (RDF list).




```rdf:RDF
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:homeOnt="http://www.sabrilyazid.org/iot#">

    <!-- Description of the room -->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/iot/Room101">
        <homeOnt:recentReadings rdf:parseType="Collection">
            <homeOnt:Temperature>22.5</homeOnt:Temperature>
            <homeOnt:Humidity>45</homeOnt:Humidity>
            <homeOnt:Motion>false</homeOnt:Motion>
        </homeOnt:recentReadings>
    </rdf:Description>

</rdf:RDF>
```

Try your self

Use rdf:seq and rdf:alt

3.7.10. Simplifying an RDF property when its value is a literal.

1. When the **predicate has a simple literal value**, RDF/XML allows it to be **expressed as an attribute** of <rdf:Description>.
2. This makes the document more concise and readable.

<rdf:Description rdf:about="YourResource">

<YourPredicate>literal</YourPredicate>

</rdf:Description>

- Here, <YourPredicate> is a **child element** of <rdf:Description>.

- The predicate's value is directly the text content (a literal).

  "Simplified" form (as an attribute):

  <rdf:Description rdf:about="YourResource" YourPredicate="literal"/>

Example

<?xml version="1.0" encoding="utf-8" ?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:foaf="http://xmlns.com/foaf/0.1/" >

<foaf:Person rdf:about="http://www.sabrilyazid.org/homeOntt#Ly" foaf:name="Lyazid" />

</rdf:RDF>

Dans nombreux validateur on n'autorise cette serialisation, même si l'attibut n'est pas un littéral.

Concept / Classe : LivingRoom

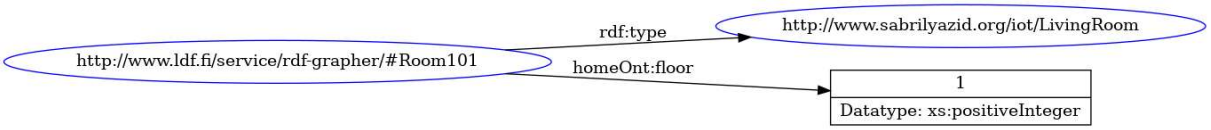
```
<rdf:Description rdf:about="http://www.sabrilyazid.org/iot/LivingRoom">
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Description>
```

Namespaces:
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: http://www.w3.org/2000/01/rdf-schema#
homeOnt: http://www.sabrilyazid.org/iot#
xs: http://www.w3.org/2001/XMLSchema#

Concept / Classe : LivingRoom (as attribute)

```rdf
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:homeOnt="http://www.sabrilyazid.org/iot#"
    xmlns:xs="http://www.w3.org/2001/XMLSchema#">

   <!-- Instance Room101 du concept LivingRoom -->
    <rdf:Description rdf:ID="Room101"
        rdf:type="http://www.sabrilyazid.org/iot/LivingRoom">
        <homeOnt:floor rdf:datatype="http://www.w3.org/2001/XMLSchema#positiveInteger">1</homeOnt:floor>
    </rdf:Description>
</rdf:RDF>
```



Namespaces:
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: http://www.w3.org/2000/01/rdf-schema#
homeOnt: http://www.sabrilyazid.org/iot#
xs: http://www.w3.org/2001/XMLSchema#

3.8. When RDF Reaches Its Limits: Towards More Expressive Models

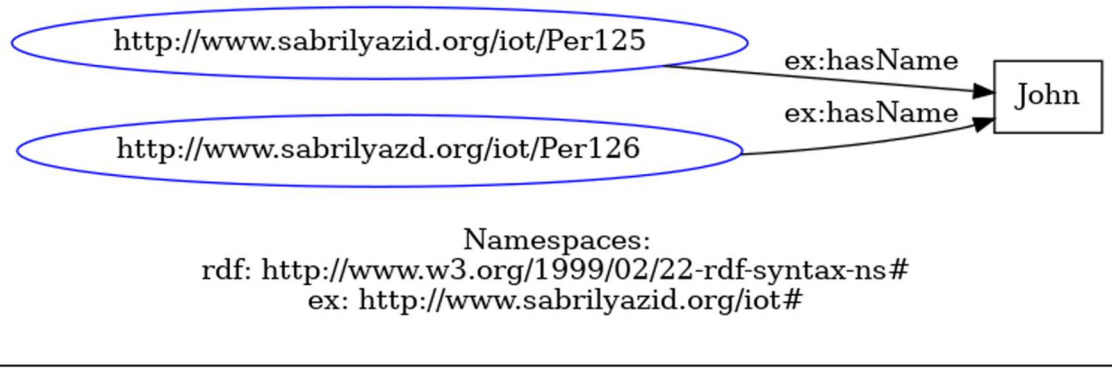
RDF/IoT Example	What cannot be inferred automatically	Possible Solution
`Per125 hasName "John"`	`"John" hasID Per125`	RDF is directed; the inverse must be explicitly stated.

Namespaces:
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
ex: http://www.sabrilyazid.org/iot#

RDF/IoT Example	What cannot be inferred automatically	Possible Solution
`ArtifactX remonteInformation ValeurY`	`ArtifactX rdf:type Sensor`	The type must be explicitly declared or a rule must be used.

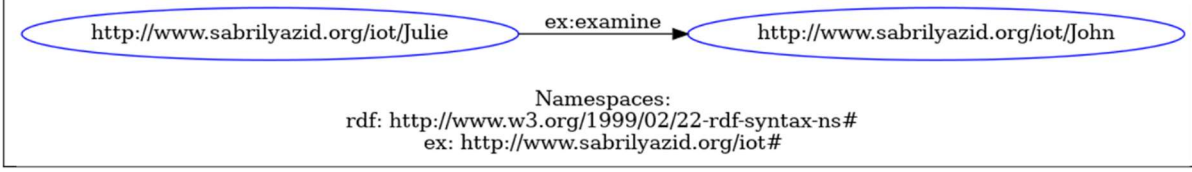
Namespaces:
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
ex: http://www.sabrilyazid.org/iot#

RDF/IoT Example	What cannot be inferred automatically	Possible Solution
`Per125 hasName "John"`, `Per126 hasName "John"`	Whether `Per125 ≠ Per126` or `Per125 = Per126`	Having the same name does not guarantee identity or difference.

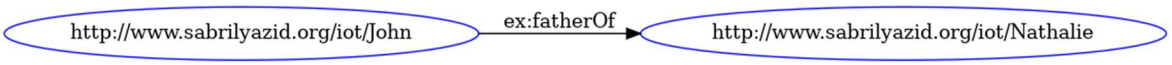


RDF/IoT Example	What cannot be inferred automatically	Possible Solution
`Robot1 communicatesWith John`	`John communicatesWith Robot1`	Symmetric relations must be explicitly declared or use `owl:SymmetricProperty`.

RDF/IoT Example	What cannot be inferred automatically	Possible Solution
`Julie examine John`	Julie is a doctor	Inferring an individual's type from an action requires explicit rules or a reasoner.

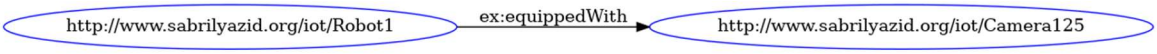


RDF/IoT Example	What cannot be inferred automatically	Possible Solution
John is the father of Nathalie	Nathalie is the daughter of John	Inverse relations must be explicitly declared.



Namespaces:
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
ex: http://www.sabrilyazid.org/iot#

RDF/IoT Example	What cannot be inferred automatically	Possible Solution
Robot1 equippedWith Camera125	Camera125 is carried by Robot1	Inverse relations in IoT must be explicitly declared.



Namespaces:
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
ex: http://www.sabrilyazid.org/iot#

3.9. Reification (rdf:Statement)

Turn a triple into an RDF resource so that we can attach metadata to it. We use the class rdf:Statement with three required properties:

- rdf:subject
- rdf:predicate
- rdf:object

- **Advantage**: standard RDF mechanism, no extra framework required.
- **Limitation**: verbose, not very efficient for large datasets, reasoning can be heavy.

Example

A sensor in Room12 reports that the **temperature is 35°C**.
We want to add metadata: *this fact was observed on 2025-09-25 by SensorX*.

<rdf:Description rdf:about="http://www.sabrilyazid.org/iot#Room12">

<homeOnt:hasTemperature>35</ homeOnt:hasTemperature>

</rdf:Description>

In the following :

1. The triple **(Room12, hasTemperature, 25)** is turned into a resource (rdf:Statement).
2. Extra facts (who observed it, when) are attached to that statement.

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:homeOnt="http://www.sabrilyazid.org/iot#">

<rdf:Statement rdf:about="http://www.sabrilyazid.org/iot#assertion1">

<rdf:subject rdf:resource="http://www.sabrilyazid.org/iot#Room12"/>

<rdf:predicate rdf:resource="http://www.sabrilyazid.org/iot#hasTemperature"/>

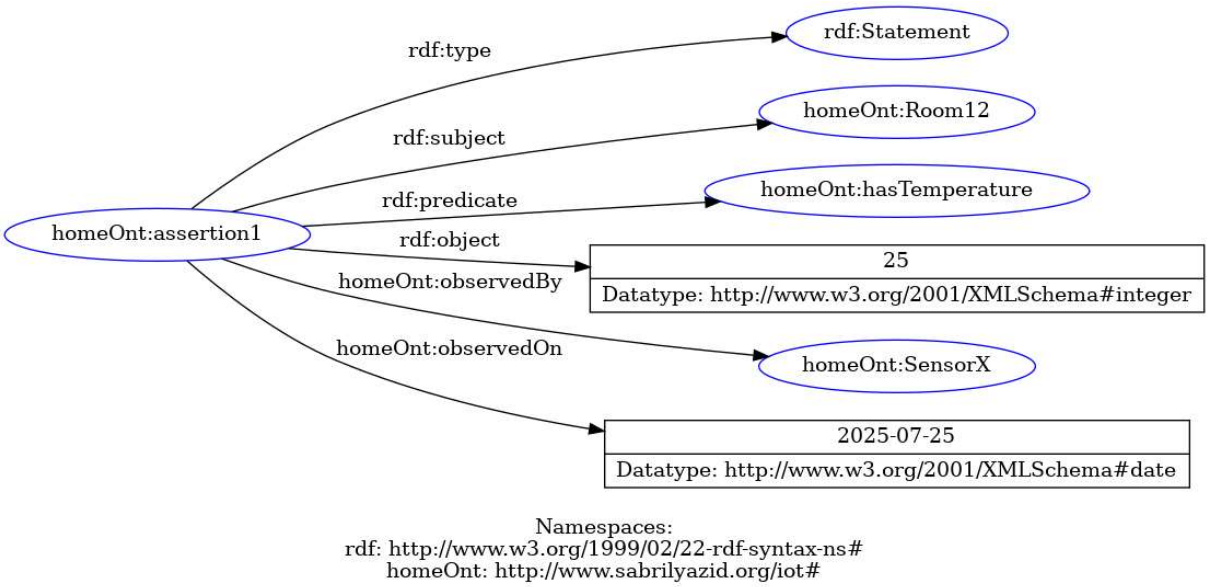
<rdf:object rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">25</rdf:object>

<homeOnt:observedBy rdf:resource="http://www.sabrilyazid.org/iot#SensorX"/>

<homeOnt:observedOn rdf:datatype="http://www.w3.org/2001/XMLSchema#date">2025-07-25</homeOnt:observedOn>

</rdf:Statement>

</rdf:RDF>



3.10. RDF(S) :Resource Description Framework (Schema)

RDF(S) is a standard language for representing data in graphs, allowing the definition of resources, properties, and relations.

RDFS Vocabulary:

- The RDFS vocabulary is defined in a different namespace than RDF.
  Prefix used: rdfs: to reference http://www.w3.org/2000/01/rdf-schema#
- RDFS is used together with RDF to enrich descriptions.
- Everything is described in RDF as a resource, and all are instances of the class rdfs:Resource. All other classes are subclasses of rdfs:Resource.

RDFS provides classes and properties from which vocabularies can be built.

1. The concept of a class is similar to Object-Oriented (OO) systems such as Java.
2. In Java, for example, it is possible to define a class Robot described by a property assists.
3. The property assists can have Person as its range. In other words, this structure indicates that each instance of Robot is linked to an instance of Person via the assists property.
4. In contrast, the W3C proposes the RDFS standard, which provides more flexibility and allows an independent description of properties and classes.
5. Resources are grouped into classes.
6. The set of instances of a class is called its extension.
7. A class is itself a resource.
8. Resources that are classes form a class called rdfs:Class.
9. Classes are themselves resources, usually identified by IRIs (URIs, URLs)

10. **rdfs:Literal**
    This is the class of literal values such as strings and integers.
11. **rdfs:Datatype**
    This is the class of data types.

3.11. Two different classes may share the same extension.

A class may define properties that another class does not. Consider two classes: Agent and Artifact. A robot can be regarded as an instance of the class Agent and, at the same time, an instance of the class *Artifact*. Indeed, the class *Agent* may also have instances of type *Person*, which is different from the class *Artifact*.

3.11.1. Domain & Range (RDFS Vocabulary)

- **rdfs:domain**: any resource that has a given property is considered an instance of one or more classes.
- **rdfs:range**: any value of a property is considered an instance of one or more classes.

Formalization:

- **P rdfs:domain C**
    - *P* is a property, *C* is a class.
    - All subjects of triples where *P* is the property are instances of *C*.
    - $\forall X,Y \ (P(X,Y) \Rightarrow C(X))$
- **P rdfs:range C**
    - *P* is a property, *C* is a class.
    - All objects of triples where *P* is the property are instances of *C*.
    - $\forall X,Y \ (P(X,Y) \Rightarrow C(Y))$
- **rdf:Property**

    $\forall x,y \ \ P(x , y) \wedge (P, \text{rdfs:range}, C) \Rightarrow \ (y, \text{rdf:type } C)$

    In RDF(S), every property is a resource of type rdf:Property.
    This means that a property is not just an “abstract link,” but a first-class object in the RDF model, which can have its own metadata (such as domain, range, label, etc.).

```
<rdf:Property rdf:about="http://www.sabrilyazid.org/homeOnt#assists">
    <rdfs:domain rdf:resource ="http:// www.sabrilyazid.org/homeOnt# Agent"/>
     <rdfs:domain rdf:resource ="http:// www.sabrilyazid.org/homeOnt# Artifact"/>
     <rdfs:range rdf:resource ="http:// www.sabrilyazid.org/homeOnt# Person"/>
</rdf:Property>
```

The **range** of the relation assists is foaf:Person.
Therefore, in any RDF triple where assists is the predicate, it implies that the **Object** of the triple is of type foaf:Person (see the queries), and its **Subject** may be two kind of class Agent and Artifact.

Example RDFS Vocabulary

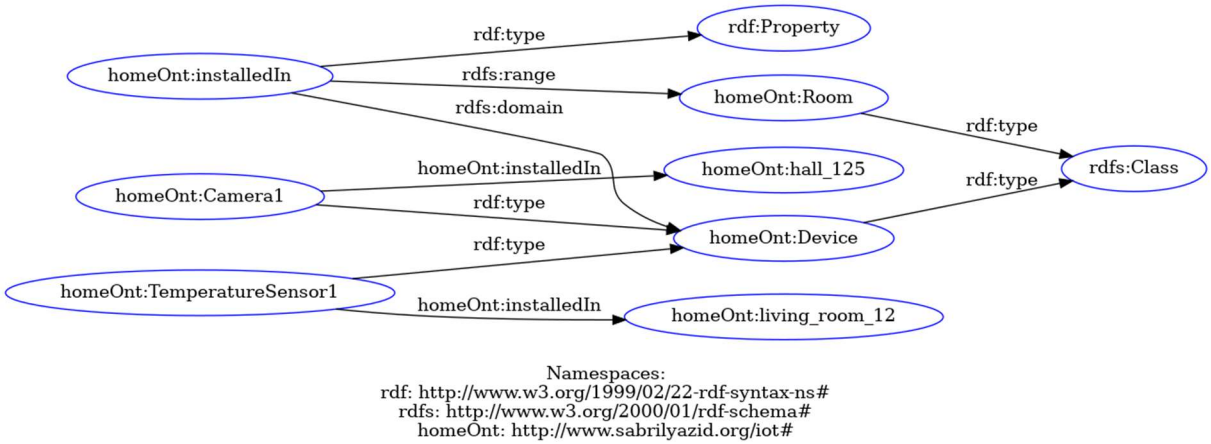
```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:homeOnt="http://www.sabrilyazid.org/iot#">

    <!-- Define Classes -->
    <rdfs:Class rdf:about="http://www.sabrilyazid.org/iot#Device"/>
    <rdfs:Class rdf:about="http://www.sabrilyazid.org/iot#Room"/>

    <!-- Define Property -->
    <rdf:Property rdf:about="http://www.sabrilyazid.org/iot#installedIn">
        <rdfs:domain rdf:resource="http://www.sabrilyazid.org/iot#Device"/>
        <rdfs:range rdf:resource="http://www.sabrilyazid.org/iot#Room"/>
    </rdf:Property>

    <!-- Instances of Device -->
    <homeOnt:Device rdf:about="http://www.sabrilyazid.org/iot#TemperatureSensor1">
        <homeOnt:installedIn rdf:resource="http://www.sabrilyazid.org/iot#living_room_12"/>
    </homeOnt:Device>
    <homeOnt:Device rdf:about="http://www.sabrilyazid.org/iot#Camera1">
        <homeOnt:installedIn rdf:resource="http://www.sabrilyazid.org/iot#hall_125"/>
    </homeOnt:Device>

</rdf:RDF>
```



Explanation

- rdfs:Class defines concepts: Device and Room.
- rdf:Property defines installedIn with a domain (Device) and a range (Room).
- Instances:
    - TemperatureSensor1 is a Device installed in living_room_12 with is a Room.
    - Camera1 is a Device installed in hall_125 wich is a Room.
      Etc.

P rdfs:subPropertyOf P'.

∀X,Y P(X,Y) ⇒ P' (X , Y), P ⊑ P'

2. rdfs:subPropertyOf defines a subproperty relationship between two properties.
3. If a property P1P_1P1 is declared as a subproperty of P2P_2P2, then any triple using P1P_1P1 also implies a triple with P2P_2P2.

<rdf:Property rdf:about="http://www.sabrilyazid.org/homeOnt#amiDe">

<rdfs:subPropertyOf rdf:resource="http://xmlns.com/foaf/0.1/knows"/>

</rdf:Property>

This means:

- The property homeOnt:amiDe is a subproperty of foaf:knows.

subClassOf

(x , rdf:type , C) ⋀ (C , rdfs:subclassOf , C') ⇒ (x , rdf:type , C')

∀ X C(X) ⋀ (C ⊑ C') ⇒ C'(X).

∀ CC'C'' (C' ⊑ C') ⋀ (C' ⊑ C'') ⇒ (C' ⊑ C'').

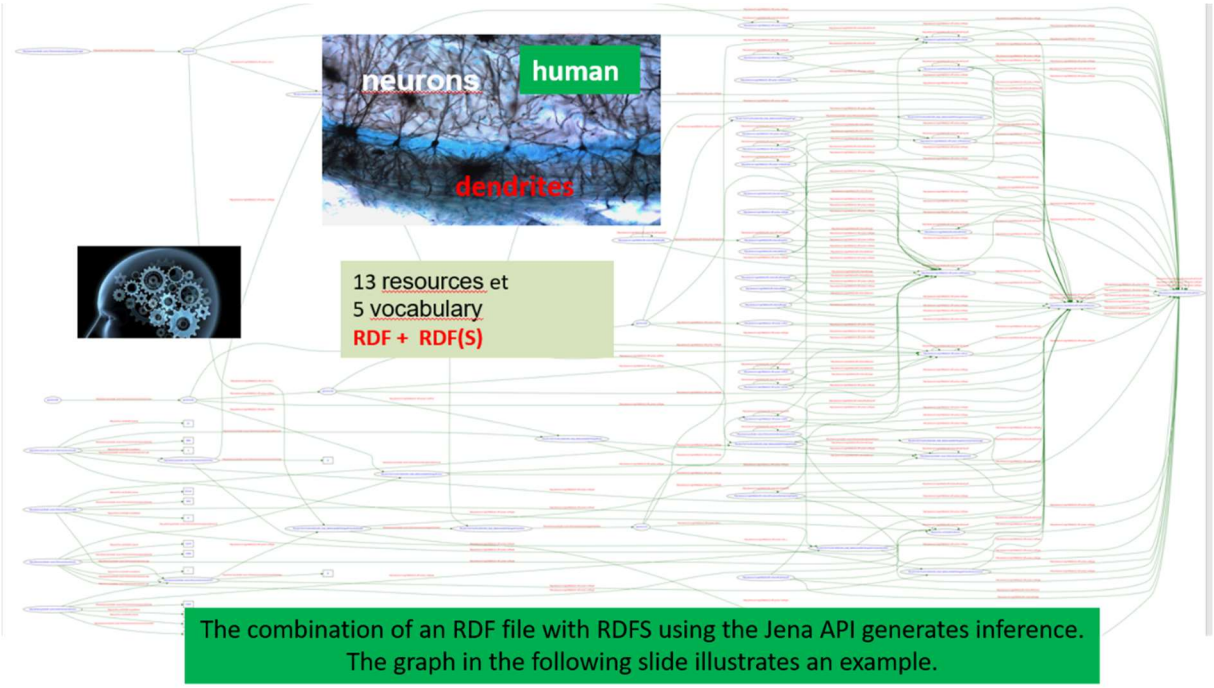
```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:ex="http://example.org/iot#">

    <!-- Device is a subclass of Artifact -->
    <rdfs:Class rdf:about="http://example.org/iot#Device">
        <rdfs:subClassOf rdf:resource="http://example.org/iot#Artifact"/>
    </rdfs:Class>

    <!-- Room is a subclass of Space -->
    <rdfs:Class rdf:about="http://example.org/iot#Room">
        <rdfs:subClassOf rdf:resource="http://example.org/iot#Space"/>
    </rdfs:Class>

</rdf:RDF>
```

According to our example above, therefore living_room_12 is an instance of Space, and TemperatureSensor1 is an instance of Artifact.



3.11.2. Data extraction from the Web – Query Languages

1. RDQL (RDF Data Query Language)
2. SeRQL (pronounced *Circle*, Sesame RDF Query Language)
3. SPARQL – A W3C (World Wide Web Consortium) standard since January 15, 2008 https://www.w3.org/TR/rdf-sparql-query/ and https://www.w3.org/TR/sparql11-query/)
4. https://jena.apache.org/documentation/fuseki2/

3.12. SPARQL a recursive acronym

(SPARQL, pronounced *Sparkle* — meaning *scintillation* in English), stands for **SPARQL Protocol and RDF Query Language**.

SPARQL is built upon three core components:

- **Query Language**
- **Response Format**: XML (W3C Recommendation, new edition, March 21, 2013)

- **Protocol**: Based on WSDL 2.0 (XML Schema) for HTTP and SOAP

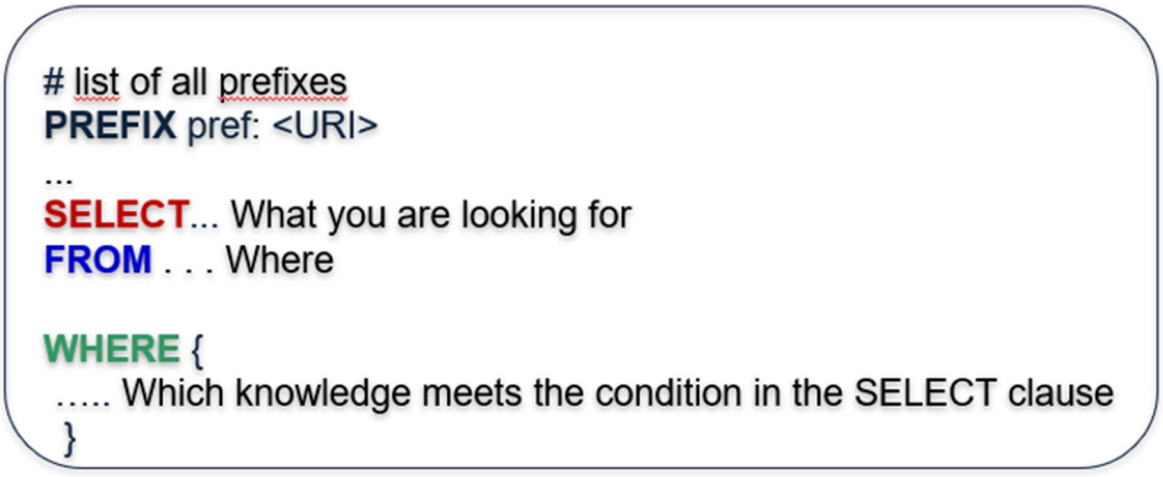
SPARQL: Protocol & Query Language
The specification http://www.w3.org/TR/rdf-sparql-protocol/ describes how to query a SPARQL endpoint/processor.

3.13. SPARQL Query Types

A SPARQL query typically begins with a list of prefixes to define namespaces, for example PREFIX pref: <URI>. The SELECT clause specifies the variables or data you want to retrieve, often accompanied by a FROM clause to indicate the target graph. The WHERE clause then defines the triple patterns or conditions that the data must satisfy in order to match the query. Together, these components allow you to precisely extract and filter knowledge from an RDF graph:

- **SELECT query**: Retrieves values that match a triple pattern.
- **ASK query**: Checks whether a triple pattern exists in the graph.
- **DESCRIBE query**: Returns a description of resources matching the pattern.
- **CONSTRUCT query**: Builds a new RDF graph based on the pattern.

All these queries are based on the **triple pattern** <S P O> (Subject, Predicate, Object).
Each component can be a **variable**, introduced with ? or $.



According to the figure above:

1. Specifies the **variable bindings** to be returned in the query results.

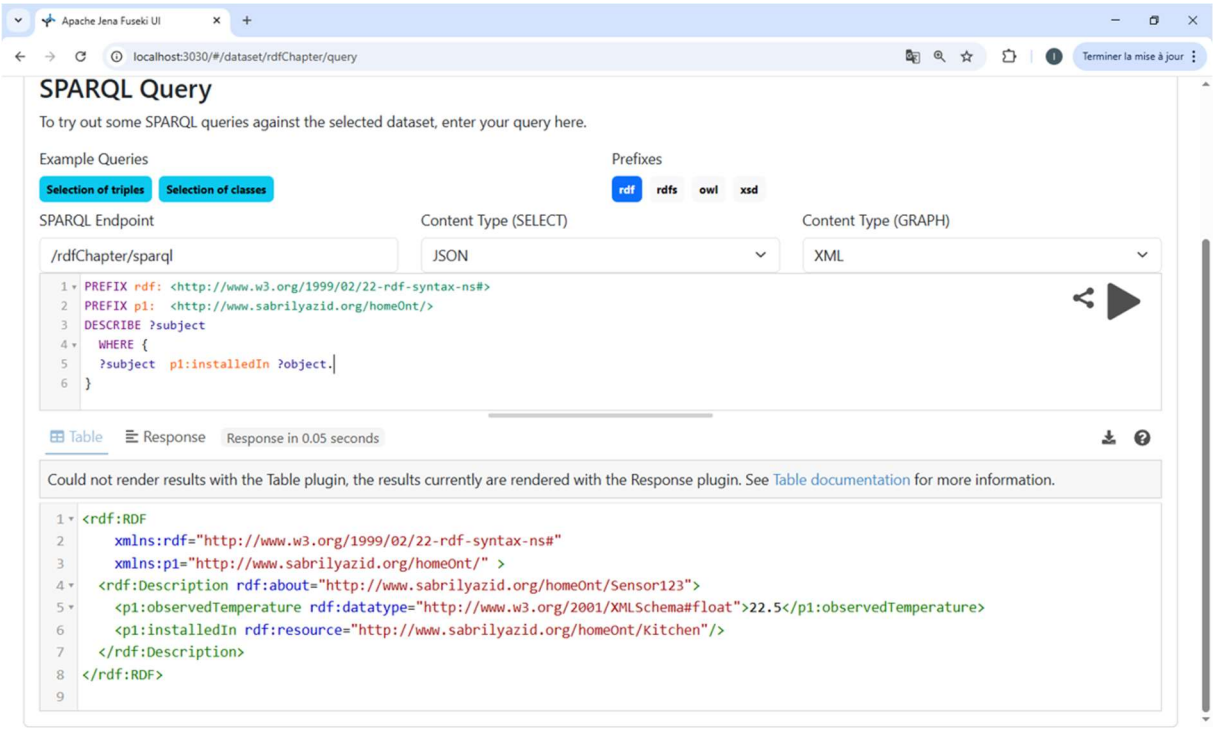
2. The **FROM** clause indicates which graph to query (named graph) and is **optional** if the target dataset is already known.

3. The **WHERE** clause contains the **graph patterns** that define the desired results. (Note: the word **WHERE** is not mandatory.)

3.13.1. DESCRIBE Query in SPARQL

The DESCRIBE form returns an RDF graph that provides information about a resource. Instead of giving only variable bindings (like SELECT), it returns a description of the resource(s) specified.

1. Ignores the specific data graph.
2. Ignores the exact variables to request.
3. Solution: DESCRIBE is used to provide a description of a resource.
4. The result of a DESCRIBE query is an **RDF graph**.

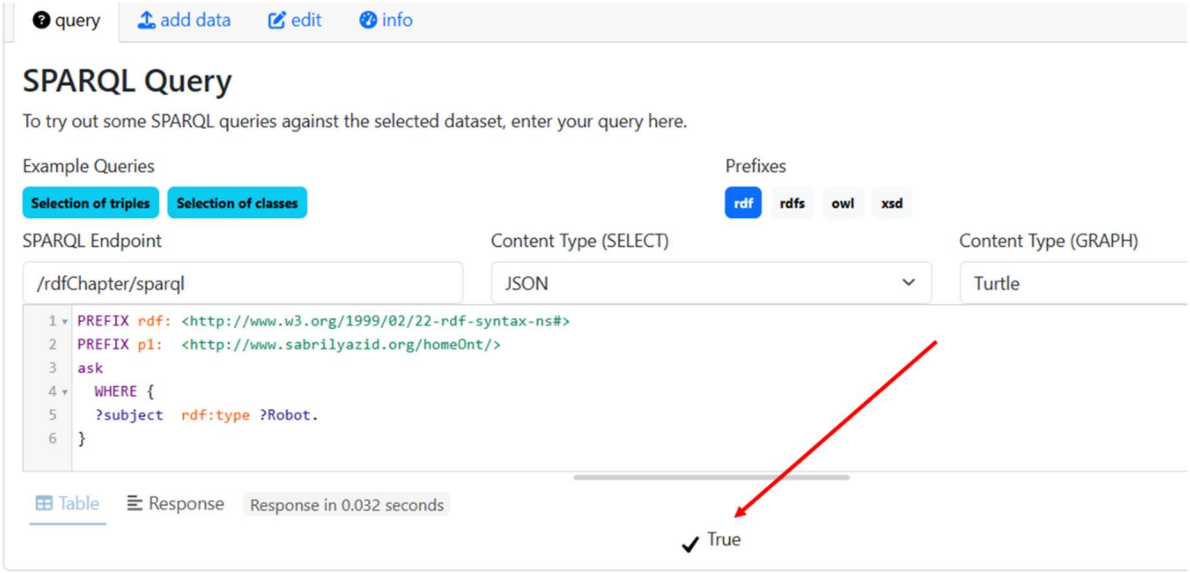


3.13.2. ASK Clause

Applications use the ASK form to check if a query pattern has at least one match. It does not return details about the results, only a true or false indicating whether a solution exists.

A SPARQL ASK query returns true or false depending on whether the graph pattern in the query matches any data in the knowledge base.

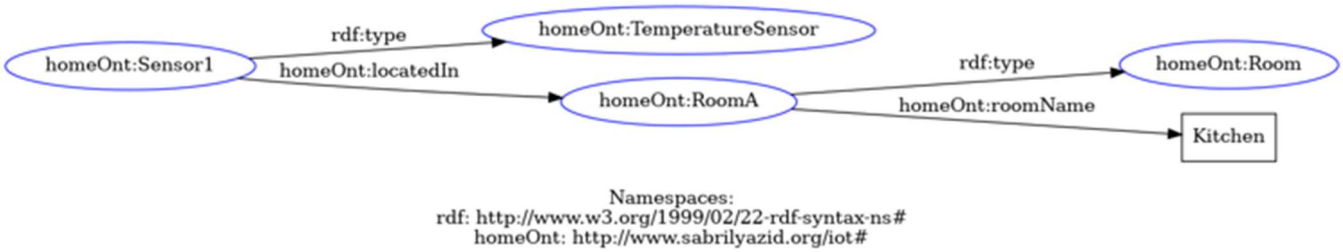
Example*:* Check if there is at least one instance of Robot in the dataset.



3.13.3. Construct Clause

When you run a CONSTRUCT query in SPARQL, the result is a new RDF graph (not a table like SELECT, not true/false like ASK).

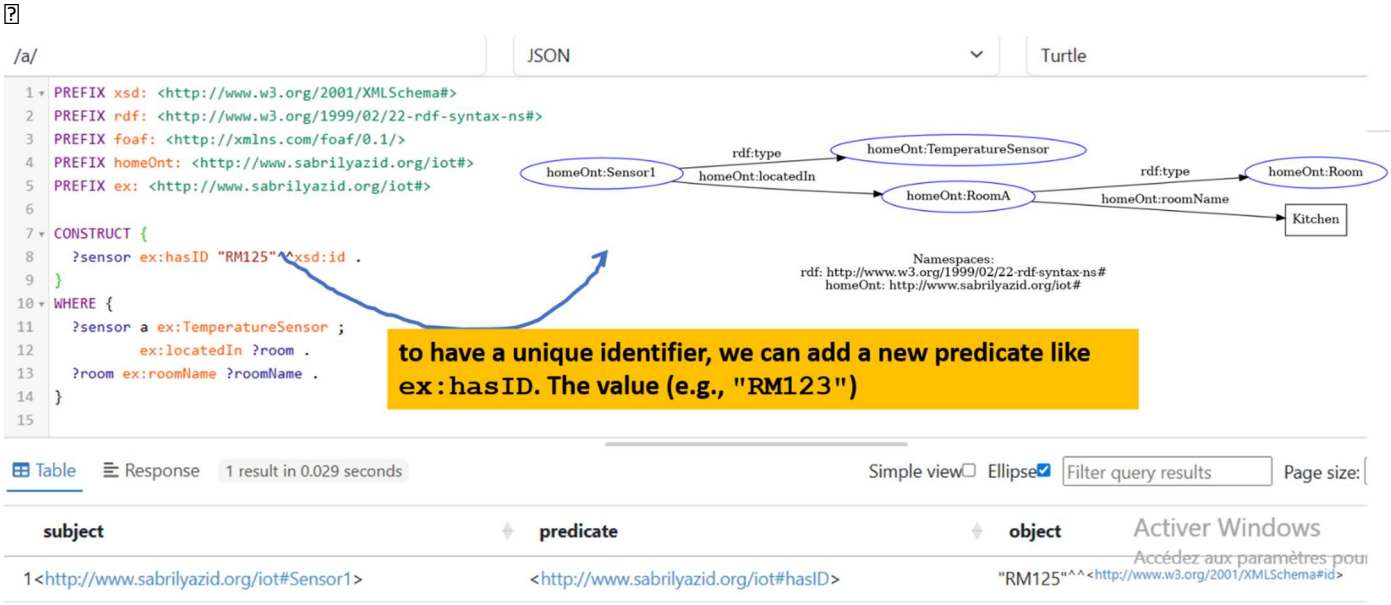
Suppose we start with this rdf graph



The CONSTRUCT query in SPARQL produces a single RDF graph defined by a graph template. For each solution found in the WHERE clause, variables are substituted into the template, and the resulting triples are combined into one RDF graph using set union.

If a substitution generates an invalid triple — for example, a triple with an unbound variable or with a literal in the subject or predicate position — that triple is discarded and does not appear in the output graph. The template can also include ground triples (triples without variables), which are always included in the final RDF result.

Example (adding an ID predicate):
Suppose we want to enrich each sensor with a new identifier. We can use CONSTRUCT to add an ex:hasID property to all temperature sensors:



3.13.4. Select **clause**

The SELECT query form returns a set of variables together with their values (bindings). It works by projecting the chosen variables from the query results, while also allowing the introduction of new variable bindings within the solution.

When the SELECT clause specifies a list of variable names, only those variables and their corresponding bindings are included in the result. Using SELECT * is a shorthand to return all in-scope variables at that stage of the query. Variables that appear only in FILTER expressions, in the right-hand side of a MINUS, or inside subqueries are not included.

The use of SELECT * is restricted: it can only be applied when the query does **not** include a GROUP BY clause.

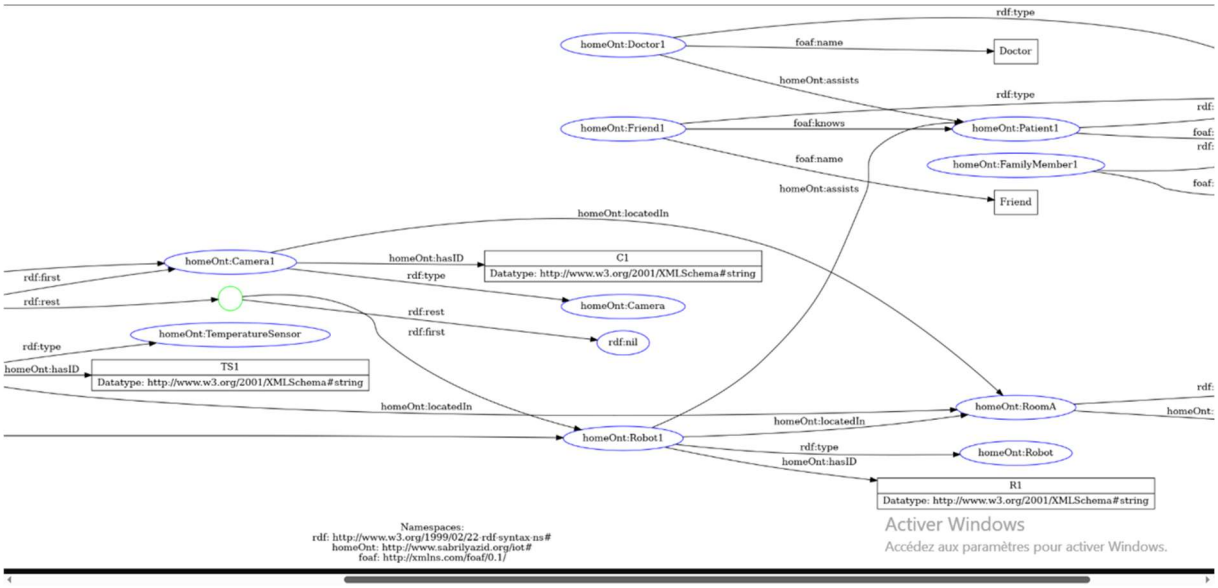
Example:

```
SELECT *    (do  not use * )
WHERE {
  ?sensor a ex:TemperatureSensor ;
        ex:locatedIn ?room .
}
GROUP BY ?room
```

3.14. Select query Examples

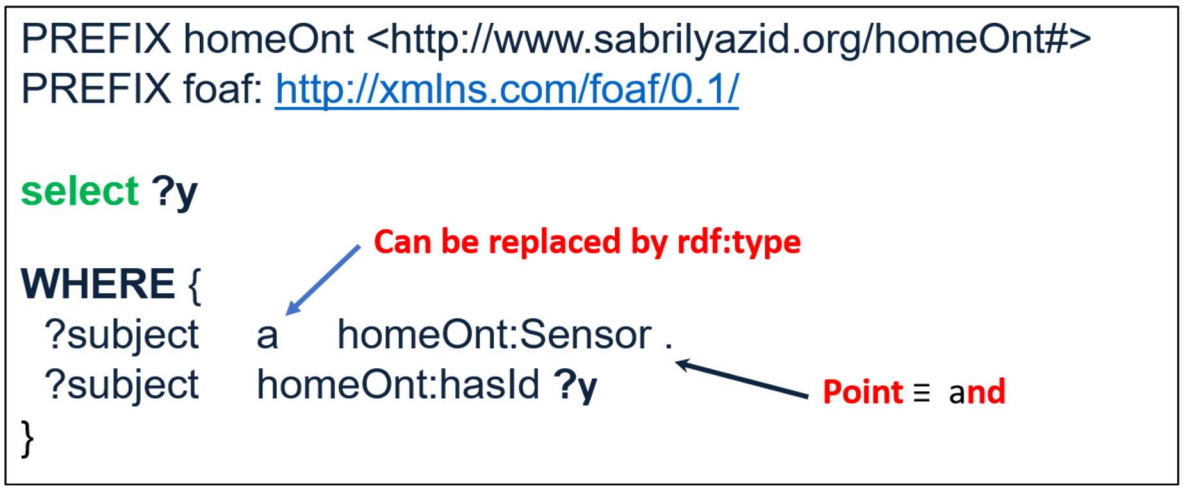
In the upcoming sessions, we will perform several SPARQL queries on the RDF file named ***rdf_smartHome*** to better understand how the data can be explored and manipulated. To gain a deeper understanding of these concepts, I strongly encourage you to carefully review the Supervised Exercises and the lab sessions. The tutorials will help you reinforce the theoretical aspects through guided exercises, while the labs will give you the opportunity to practice with concrete examples. Following these materials alongside the course will provide you with the necessary foundation to master RDF and SPARQL querying.

An excerpt from the RDF graph is presented below to illustrate the structure of the data.



Example 1 (rdf:type ≡ a)

The terms rdf:type and a are semantically equivalent. Both are used to indicate the type (or class) of a resource.



Example 2 Find the sensor IDs

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX homeOnt: <http://www.sabrilyazid.org/iot#>
select ?y
WHERE {
  ?subject     a     homeOnt:Camera .
  ?subject     homeOnt:hasID ?y
}
```

Caution: Depending on the variables used, we may simply say *"search for the IDs of the sensors."* However, attention must be paid: if the vocabulary being used does not carry explicit semantic meaning in a language understandable to the reader, ambiguity may arise. For example, if we do not know what "Camera" means in this context, we should express the query more explicitly.

In this case, the query should be formulated as:

"Search among the triples for the objects (?y) linked to a subject by the predicate homeOnt:hasID, where the subject is of type (a) homeOnt:Camera."

In the query, we do not need to duplicate the variable ?subject, because it already refers to the same subject within the same query block.

For example, instead of writing it like this (with repetition):

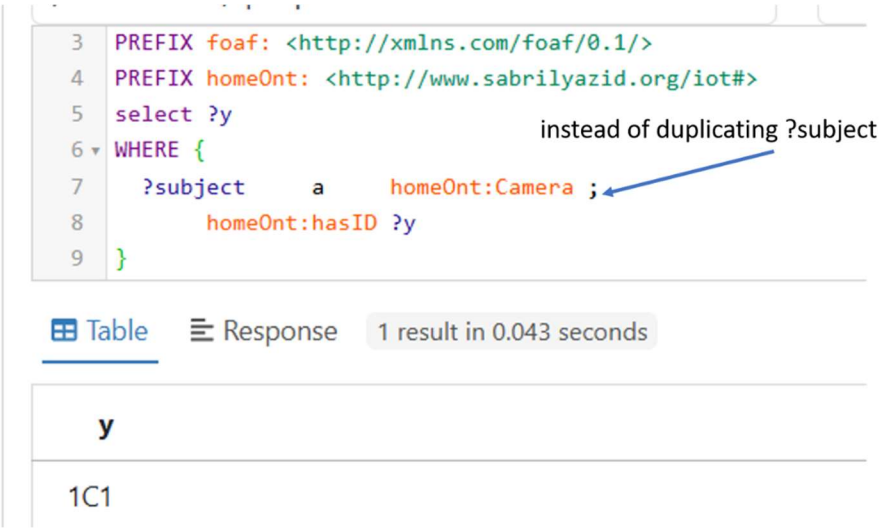
?subject a homeOnt:Camera .

? subject homeOnt:hasID ?id .

we can combine the two triples into one block, since they share the same subject ?camera:

?camera a homeOnt:Camera ;

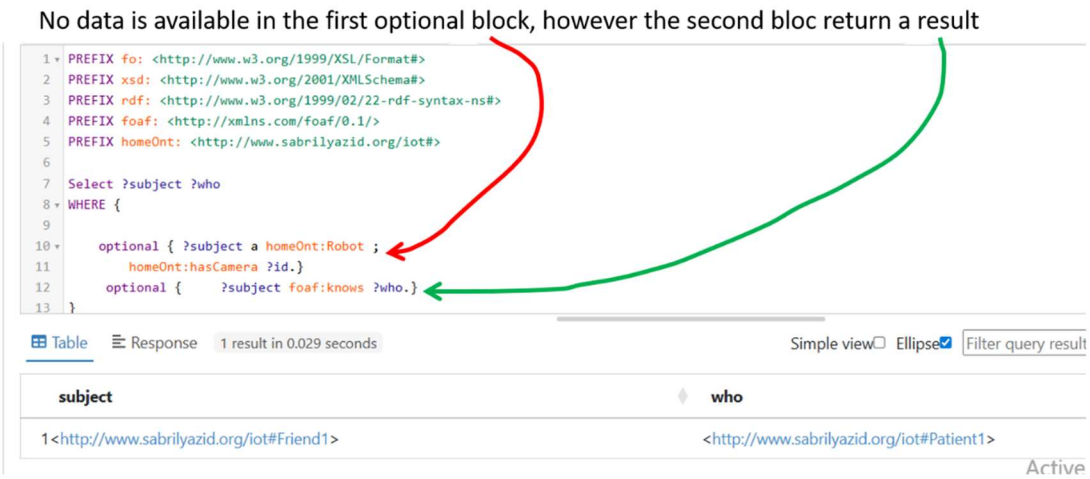
homeOnt:hasID ?id .



Example 3 OPTIONAL

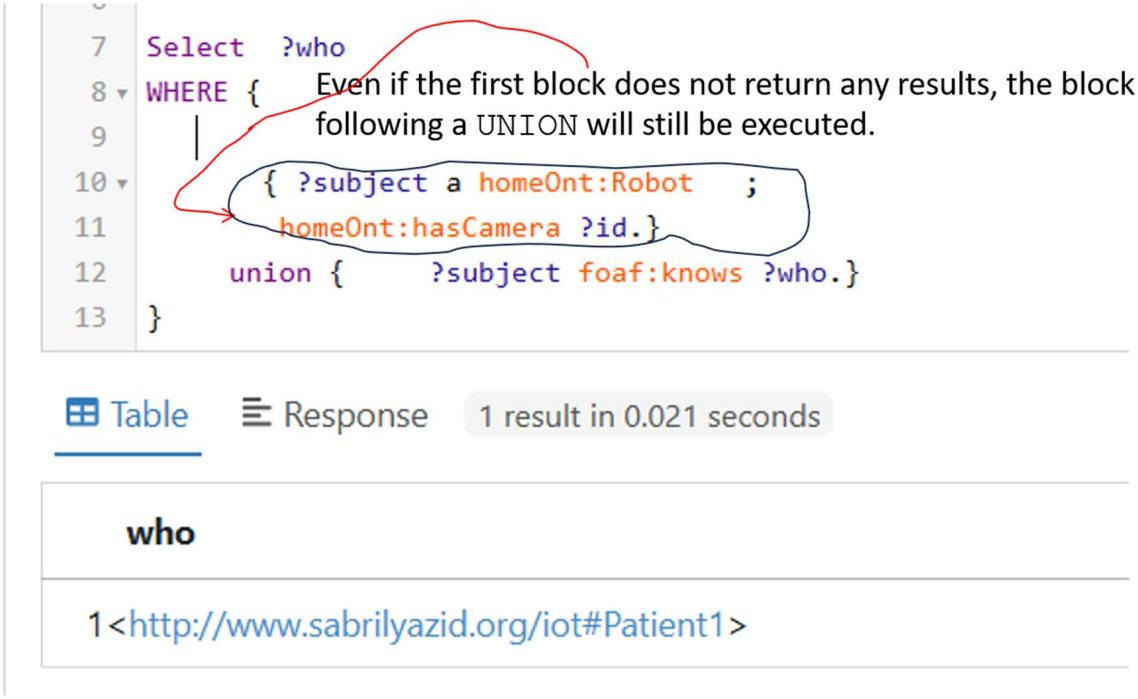
In a SPARQL query with two blocks, one mandatory and the other optional, note that if the first (mandatory) block does not return any result, then the optional block is not executed. This behavior is logical: an OPTIONAL clause only applies if the mandatory part of the query has already matched something. If nothing matches in the required part, then the optional parts are simply skipped

However, if you enclose both blocks within OPTIONAL, then each optional block will be evaluated one by one.



Example 4 Expressing mutually exclusive graph patterns.

In SPARQL, the UNION keyword is used to combine two or more graph patterns so that a solution matches either one pattern or the other. This allows you to query for mutually exclusive or alternative conditions in your RDF data.



Key point: Unlike OPTIONAL, UNION ensures that all alternative patterns are checked independently, regardless of whether other blocks produce results.

Example xsd datatype

In SPARQL, you can query RDF literals that have a specific **XML Schema datatype** (XSD), such as xsd:string, xsd:integer, etc.

Example: Suppose we want to find all roomName values that are typed as xsd:string in our RDF graph:

```
PREFIX fo: <http://www.w3.org/1999/XSL/Format#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX homeOnt: <http://www.sabrilyazid.org/iot#>


SELECT ?roomName
WHERE {
  ?room homeOnt:roomName ?roomName .
  FILTER(datatype(?roomName) = xsd:string)
}
```

Explanation:

- ?roomName is the object of homeOnt:roomName.
- FILTER(datatype(?roomName) = xsd:string) ensures that only literals typed as xsd:string are returned.

Minus vs not Exist

- NOT EXISTS is used to filter out solutions for which a given pattern exists.
- It allows you to exclude results that match a specific graph pattern.
- MINUS removes all solutions that match the pattern in the MINUS clause.
- It behaves like a set difference between the main pattern and the pattern inside MINUS.

```
SELECT ?person
WHERE {
  ?person a foaf:Person .
  FILTER NOT EXISTS { ?person foaf:knows ?friend . }
}
```

Table | Response | 3 results in 0.065 seconds

person
1<http://www.sabrilyazid.org/iot#Patient1>
2<http://www.sabrilyazid.org/iot#Doctor1>
3<http://www.sabrilyazid.org/iot#FamilyMember1>

```
PREFIX fo: <http://www.w3.org/1999/XSL/Format#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX homeOnt: <http://www.sabrilyazid.org/iot#>
SELECT ?person
WHERE {
  ?person a foaf:Person .
  MINUS { ?person foaf:knows <http://www.sabrilyazid.org/iot#Patient1> . }
}
```

Table | Response | 3 results in 0.06 seconds

person
1<http://www.sabrilyazid.org/iot#Patient1>
2<http://www.sabrilyazid.org/iot#Doctor1>
3<http://www.sabrilyazid.org/iot#FamilyMember1>

Showing 1 to 3 of 3 entries

Goal: Find all people who **do not know anyone**.

Explanation 1:

- ?person a foaf:Person selects all individuals.
- FILTER NOT EXISTS { ?person foaf:knows ?friend } removes anyone who has a foaf:knows relationship.
- The result will include persons like Doctor1 or FamilyMember1 if they do not have foaf:knows links.

Explanation 2:

- The main pattern matches all foaf:Person.
- The MINUS clause removes anyone who knows Patient1.
- The result will be all persons except Friend1 in your dataset.

Example : Property path. SPARQL 1.1

Syntax Form	Property Path Expression Name	Matches
iri	PredicatePath	An IRI. A path of length one.
^*elt*	InversePath	Inverse path (object to subject).
elt1 / *elt2*	SequencePath	A sequence path of *elt1* followed by *elt2*.
elt1 | *elt2*	AlternativePath	A alternative path of *elt1* or *elt2* (all possibilities are tried).
*elt**	ZeroOrMorePath	A path that connects the subject and object of the path by zero or more matches of *elt*.
elt+	OneOrMorePath	A path that connects the subject and object of the path by one or more matches of *elt*.
elt?	ZeroOrOnePath	A path that connects the subject and object of the path by zero or one matches of *elt*.
!*iri* or !(iri_1| ...|iri_n)	NegatedPropertySet	Negated property set. An IRI which is not one of iri_i. !*iri* is short for !(*iri*).
!^*iri* or !(^iri_1| ...|^iri_n)	NegatedPropertySet	Negated property set where the excluded matches are based on reversed path. That is, not one of iri_1...iri_n as reverse paths. !^*iri* is short for !(^*iri*).
!(iri_1| ...|iri_j|^iri_{j+1}| ...|^iri_n)	NegatedPropertySet	A combination of forward and reverse properties in a negated property set.
(*elt*)		A group path *elt*, brackets control precedence.

Property path operators include:

- / → sequence (path along multiple predicates)
- | → alternative (either predicate)
- * → zero or more occurrences
- + → one or more occurrences
- ? → zero or one occurrence

Suppose we want people who either **know Patient1** or are **assisting Patient1**:

```
SELECT ?person
WHERE {
  ?person (foaf:knows | homeOnt:assists) <http://www.sabrilyazid.org/iot#Patient1> .
}
```

Table | Response | 3 results in 0.032 seconds

	person
1	<http://www.sabrilyazid.org/iot#Friend1>
2	<http://www.sabrilyazid.org/iot#Robot1>
3	<http://www.sabrilyazid.org/iot#Doctor1>

Showing 1 to 3 of 3 entries

1. (foaf:knows | homeOnt:assists) means **either predicate** can be used.
2. The query will return all persons who either know or assist Patient1.

Example: ^:inverse

This RDF fragment describes a **person** using the foaf:Person class:

```
<foaf:Person rdf:about="http://www.sabrilyazid.org/iot#Friend1">
    <foaf:name>Friend</foaf:name>
    <foaf:knows rdf:resource="http://www.sabrilyazid.org/iot#Patient1"/>
</foaf:Person>
```

Explanation:

- The resource Friend1 is an instance of foaf:Person.
- foaf:name assigns the literal "Friend" as the person's name.
- foaf:knows links this person to another resource, Patient1, indicating that Friend1 knows Patient1.

In terms of a graph:

- Friend1 is a node.
- There is an edge labeled foaf:knows pointing from Friend1 to Patient1.

- Another edge labeled foaf:name points to the literal "Friend".

This example can be used to demonstrate forward and inverse property paths in SPARQL:

- Forward: Who does Friend1 know? → Patient1.
- Inverse (^foaf:knows): Who knows Patient1? → Friend1.

```
SELECT        ?p1   ?p2
   WHERE  {
        ?p1  ^foaf:knows       ?p2.
        ?p1   foaf:name ?name.
     }
```

Who knows `Patient1`?

Table | Response | 1 result in 0.069 seconds | Simple view ☐ Ellipse ☑ Filter que

p1	p2
1<http://www.sabrilyazid.org/iot#Patient1>	<http://www.sabrilyazid.org/iot#Friend1>

howing 1 to 1 of 1 entry

Example Collection

In SPARQL, using the rdf_smartHome dataset where a collection:

```
    <!-- sensors Collection -->
    <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#AllSensors">
        <homeOnt:hasSensors rdf:parseType="Collection">
            <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#Sensor1"/>
            <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#Camera1"/>
            <rdf:Description rdf:about="http://www.sabrilyazid.org/iot#Robot1"/>
        </homeOnt:hasSensors>
    </rdf:Description>
```

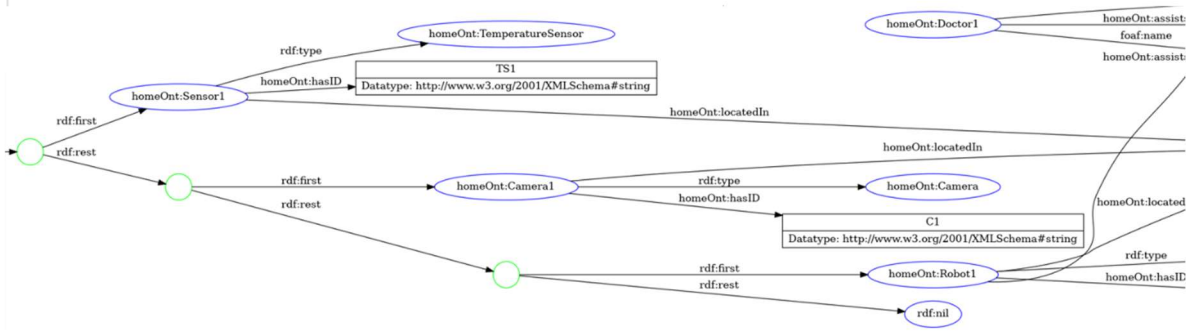
Texte Grille Auteur

1. ?list is the head of your RDF collection (the list of sensors).
2. rdf:rest* traverses zero or more rdf:rest links (the chain of the list).
3. rdf:first gets the item at each position in the list.
4. The query returns each member of the collection as ?sensor.

```
9
10 SELECT DISTINCT ?sensor
11 WHERE {
12   ?list rdf:rest*/rdf:first ?sensor .
13 }
14
```

Table | Response | 3 results in 0.034 seconds

sensor
1<http://www.sabrilyazid.org/iot#Sensor1>
2<http://www.sabrilyazid.org/iot#Camera1>
3<http://www.sabrilyazid.org/iot#Robot1>



Multiple paths to the same item:

- The property path rdf:rest*/rdf:first allows **zero or more** steps along rdf:rest.
- If the same sensor appears in multiple positions (or if the list is not properly terminated), SPARQL can traverse **multiple paths to the same ?sensor**, producing duplicates.

DF Collections and Property Path Notation

1. rdf:rest{3}/rdf:first
    - rdf:rest{3} means follow the rdf:rest link exactly 3 times from the head of the list.
    - /rdf:first then retrieves the item (value) at that position.
    - So this returns the third element (or third “pair”) in the RDF list.
2. rdf:rest{3}
    - Simply refers to the third node in the linked list (without accessing its value via rdf:first).
3. rdf:rest{2,3}/rdf:first
    - {2,3} is a range: it matches 2 or 3 steps along rdf:rest.
    - /rdf:first retrieves the items at positions 2 and 3 in the list.

Example of merging RDFS with RDF

Understanding FOAF:knows vs homeOnt :friendOF

1. foaf:knows
    - Represents any acquaintance or known relationship between two people.
    - Two people can “know” each other without being close friends.
2. Friendship (subclass or property)
    - If we define a property like :friendOF as a subproperty of foaf:knows, then:
        - Every friendship implies knowing each other:
        - x :friendOF y ⇒ x foaf:knows y
        - But the reverse is not necessarily true:

x foaf:knows y ⇏ x :friendOF y

```
<foaf:Person rdf:about="http://www.sabrilyazid.org/iot#Friend1">
    <foaf:name>Friend</foaf:name>
    <foaf:knows rdf:resource="http://www.sabrilyazid.org/iot#Patient1"/>
</foaf:Person>
<foaf:Person rdf:about="http://www.sabrilyazid.org/iot#Friend1">
    <homeOnt:freindOF rdf:resource="http://www.sabrilyazid.org/iot#Doctor1"/>
</foaf:Person>
```

```
SELECT ?per1 ?per2
WHERE {
 ?per1 foaf:knows  ?per2 .
}
```

Table | Response | 2 results in 0.023 seconds | Simple view | Ellipse | Filter quer

per1	per2
1<http://www.sabrilyazid.org/iot#Friend1>	<http://www.sabrilyazid.org/iot#Patient1>
2<http://www.sabrilyazid.org/iot#Friend1>	<http://www.sabrilyazid.org/iot#Doctor1>

Friend1 knows Doctor1 because he is his friend

```
<rdf:Property rdf:about="http://www.sabrilyazid.org/iot#freindOF">
    <rdfs:subPropertyOf rdf:resource="http://xmlns.com/foaf/0.1/knows" />
</rdf:Property>
```

SubProperty effects

RDFS Vocabulary

```
    <rdf:Property rdf:about="http://www.sabrilyazid.org/iot#freindOF">
        <rdfs:subPropertyOf rdf:resource="http://xmlns.com/foaf/0.1/knows" />
    </rdf:Property>
    <rdf:Property rdf:about="http://www.sabrilyazid.org/iot#helps">
        <rdfs:subPropertyOf rdf:resource="http://www.sabrilyazid.org/iot#assists"/>
    </rdf:Property>
    <rdf:Property rdf:about="http://www.sabrilyazid.org/iot#monitors">
        <rdfs:subPropertyOf rdf:resource="http://www.sabrilyazid.org/iot#helps"/>
    </rdf:Property>
```

Robot1 uses only minotors relationship

```
<homeOnt:Robot rdf:about="http://www.sabrilyazid.org/iot#Robot1">
    <homeOnt:hasID rdf:datatype="http://www.w3.org/2001/XMLSchema#string">R1</homeOnt:hasID>
    <homeOnt:locatedIn rdf:resource="http://www.sabrilyazid.org/iot#RoomA"/>
    <homeOnt:monitors rdf:resource="http://www.sabrilyazid.org/iot#Patient1"/>
</homeOnt:Robot>
```

Doctor1 assists Patient1 and Freind1 helps Patient1

```
    <foaf:Person rdf:about="http://www.sabrilyazid.org/iot#Doctor1">
        <foaf:name>Doctor</foaf:name>
        <homeOnt:assists rdf:resource="http://www.sabrilyazid.org/iot#Patient1"/>
    </foaf:Person>
    <foaf:Person rdf:about="http://www.sabrilyazid.org/iot#Friend1">
        <foaf:name>Friend</foaf:name>
        <foaf:knows rdf:resource="http://www.sabrilyazid.org/iot#Patient1"/>
         <foaf:helps rdf:resource="http://www.sabrilyazid.org/iot#Patient1"/>
    </foaf:Person>
    <foaf:Person rdf:about="http://www.sabrilyazid.org/iot#Friend1">
        <homeOnt:freindOF rdf:resource="http://www.sabrilyazid.org/iot#Doctor1"/>
        <foaf:knows rdf:resource="http://www.sabrilyazid.org/iot#Patient1"/>
    </foaf:Person>
```

```
SELECT ?per1 ?per2
WHERE {
 ?per1 homeOnt:assists  ?per2 .
}
```

Press CTRL - <spacebar> to autocomplete

Table | Response | 2 results in 0.024 seconds | Simple view | Ellipse | Filter qu

per1	per2
1<http://www.sabrilyazid.org/iot#Robot1>	<http://www.sabrilyazid.org/iot#Patient1>
2<http://www.sabrilyazid.org/iot#Doctor1>	<http://www.sabrilyazid.org/iot#Patient1>

```
SELECT ?per1 ?per2
WHERE {
 ?per1 homeOnt:helps  ?per2 .
}
```

Table | Response | 1 result in 0.02 seconds | Simple view | Ellipse | Filter qu

per1	per2
1<http://www.sabrilyazid.org/iot#Robot1>	<http://www.sabrilyazid.org/iot#Patient1>

```
SELECT ?per1 ?per2
WHERE {
 ?per1 foaf:helps  ?per2 .
}
```

Press CTRL - <spacebar> to autocomplete

Table | Response | 1 result in 0.024 seconds | Simple view | Ellipse | Filter que

per1	per2
1<http://www.sabrilyazid.org/iot#Friend1>	<http://www.sabrilyazid.org/iot#Patient1>

```
SELECT ?per1 ?per2
WHERE {
 ?per1 homeOnt:monitors  ?per2 .
}
```

Press CTRL - <spacebar> to autocomplete

Table | Response | 1 result in 0.037 seconds | Simple view | Ellipse | Filter qu

per1	per2
1<http://www.sabrilyazid.org/iot#Robot1>	<http://www.sabrilyazid.org/iot#Patient1>

Showing 1 to 1 of 1 entry

3.15. Topics Remaining to Explore with RDF

1. Cryptography – securing RDF data and ensuring integrity.
2. Temporal concepts – representing time-dependent information in RDF.
3. Numeric computations – performing calculations over RDF data.
4. RDF validation with SHACL (Shapes Constraint Language) – enforcing constraints and data quality.
5. SPARQL and Web Applications – integrating queries into web-based systems.
6. Aggregate Algebra – set functions such as COUNT, SUM, AVG, MIN, MAX.
7. Advanced SPARQL clauses – HAVING, VALUES, ORDER BY, projection of variables, DISTINCT, REDUCED, as well as OFFSET and LIMIT.

8. Other topics – additional features and advanced usages in RDF and SPARQL.

3.16. RDFS vs OWL

RDFS (RDF Schema):

- Allows you to describe the semantics of information, defining classes, properties, and simple hierarchies.
- Limitations of RDFS:
    1. Cardinality constraints cannot be specified.
        - Example: There is no way to express that a chair can only be occupied by one person at a time.
    2. Negation or disjunction of classes cannot be expressed.
        - Example: You cannot express that a person has not taken their medication, or that a Robot and a Human are distinct.
    3. Default values for properties cannot be assigned.

OWL (Web Ontology Language):

- Extends RDFS with richer semantics, allowing:
    - Specifying cardinality constraints (owl:cardinality, owl:minCardinality, owl:maxCardinality).
    - Expressing class disjointness, union, intersection, and complement.
    - Assigning default or inferred values to properties via reasoning.
- OWL enables more powerful reasoning over ontologies, making it suitable for applications that require strict semantic definitions and automated inference.

3.17. Named Graphs and SPARQL

An IRI associated with an RDF graph is called the name of the graph. Such graphs are referred to as named graphs.
Named graphs formalize the intuitive idea that the contents of an RDF document (a graph) on the Web can be considered “named” by the URI of the document. This provides several benefits:

- It simplifies provenance management for pieces of data.
- It enables fine-grained access control over the source data.
- It allows trust management, for example by applying digital signatures to the data in a named graph.

Originally, RDF reification was intended to provide these facilities, but this approach proved problematic.

When SPARQL was introduced, the need to refer explicitly to named graphs became clear.

- Without named graphs, queries can return results without knowing the provenance of resources.
- With named graphs, the graph name is available for each triple retrieved.
- Each graph in a dataset can thus have a unique name (e.g., a URI or a file address).

Example 1 Named graph

If you have an external RDF file and you've loaded it into a named graph in Fuseki, you can query that graph with SPARQL to retrieve information about its contents. Here's a concrete example.

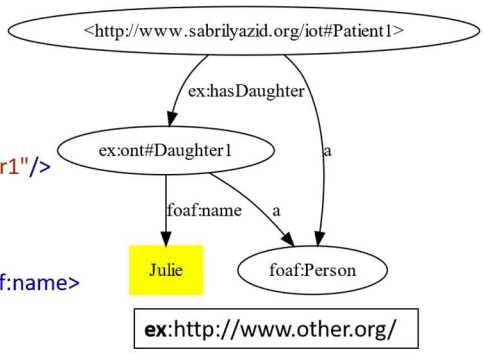
Let's create a complete RDF/XML file that represents the knowledge that Patient1 has a daughter named Julie, which you can load into a named graph in Fuseki:

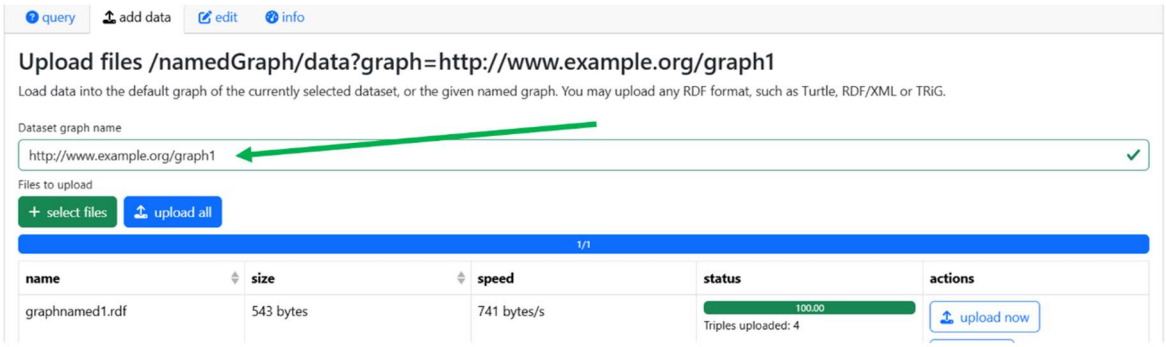
1. foaf:Person defines a person.
2. foaf:name gives the patient's name.
3. ex:hasDaughter uses an XML literal to store the daughter's name (Julie).
4. This file can be loaded into a named graph in Fuseki, e.g., http://www.example.org/graph1.

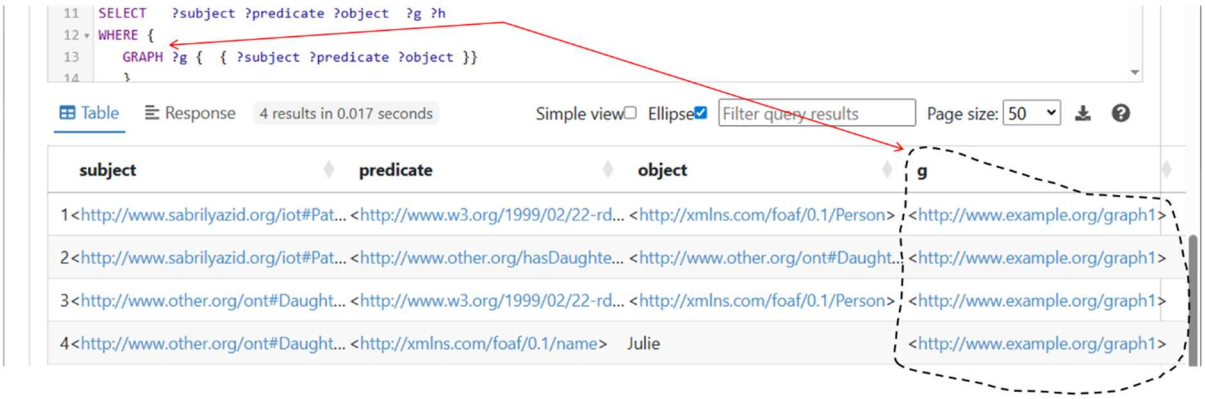
```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:ex="http://www.other.org/"
   xmlns:foaf="http://xmlns.com/foaf/0.1/">

   <foaf:Person rdf:about="http://www.sabrilyazid.org/iot#Patient1">
      <ex:hasDaughter rdf:resource="http://www.other.org/ont#Daughter1"/>
   </foaf:Person>
     <foaf:Person rdf:about="http://www.other.org/ont#Daughter1">
      <foaf:name
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Julie</foaf:name>
   </foaf:Person>

</rdf:RDF>
```



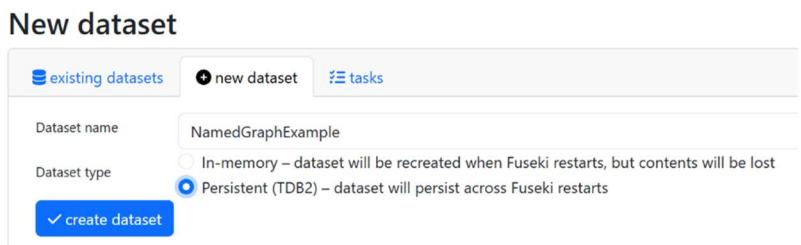


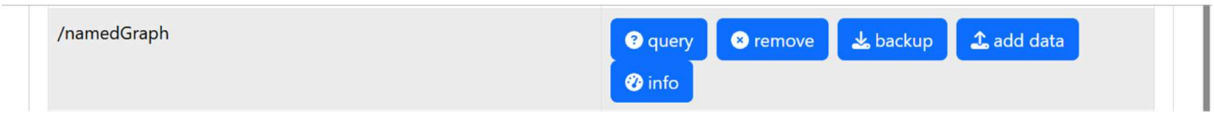


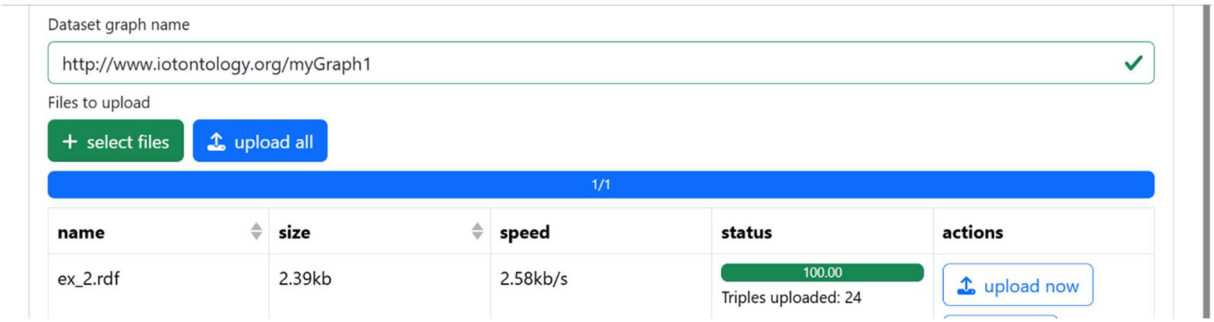
Example 2 Named graph

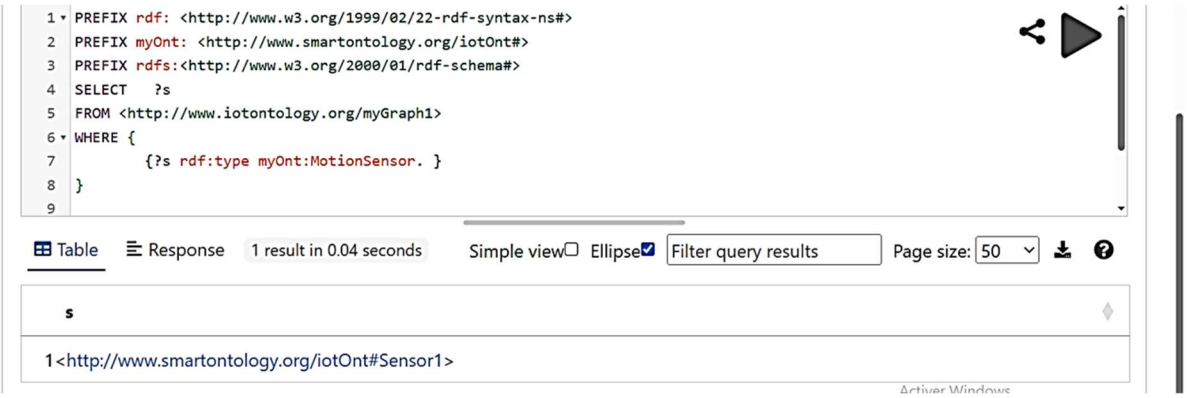
Load into Fuseki the file ex2_.rdf provided.

When naming a graph, you must give it its URI:









Expressiveness Limitations of RDF Schema (RDF-S)

RDF Schema (RDF-S) has several significant limitations in terms of expressiveness. For instance, it cannot specify cardinality constraints on a property, i.e., define how many values a property may have. A concrete example is a chair that can only be occupied by one person at a time. Moreover, RDF-S does not support expressing the negation of a class or statement. For example, it cannot represent that a person refuses to take their medication, or that a human is not a robot.

It is also unable to express disjunction between classes, such as indicating that the concepts Robot and Human are distinct. Other limitations include the inability to assign default values to attributes or to define inverse or transitive properties. These restrictions demonstrate that, while RDF-S is useful for structuring simple knowledge, it is insufficient for representing more complex scenarios that require advanced logical rules.

4. Chapter 4 Knowledge Representation and Reasoning: Logic, Pragmatics, and Ontologies

4.1. Introduction

Knowledge Representation (KR) is a central field in Artificial Intelligence, focusing on how to model facts, knowledge, and reasoning about the world. Logical approaches, such as propositional logic and predicate logic, allow expressing truth values, inference rules, and formal reasoning. Non-logical approaches, including conversational implicatures and pragmatic inferences, capture implicit meaning beyond strict logical form. Philosophers such as Grice emphasized how communication relies on both explicit and implicit information. Similarly, Dreyfus argued that human reasoning includes tacit knowledge that machines struggle to replicate. Description Logics (DL), as fragments of First-Order Logic (FOL), balance expressiveness and decidability. They are the foundation of ontology languages such as OWL. These frameworks make it possible to automate reasoning about knowledge while avoiding undecidability. In the context of AI and the Semantic Web, KR enables agents to interpret, infer, and reason intelligently with structured knowledge.

4.2. Knowledge Representation: Logical and Non-Logical Approaches

The Proposition

- A proposition (or assertion) is a statement that carries meaning.
- It corresponds to an assertion of an elementary statement that can be either true or false.
- Determining the truth value of sentences expressed in natural language as logical propositions can be delicate.

Example:
The air conditioner does not turn on.
The truth value of this statement depends on the state of the world. The question is: is the air conditioner broken, or is the room temperature simply at a level that triggers the air conditioner to stop?

Key Points:

1. Statements provide real information about the world.
2. One must be able to judge the truth of a statement, which requires understanding its meaning.

3. A statement is assumed true if it conforms to the facts of the world (the state of the universe).
4. A statement must satisfy certain truth conditions (i.e., vericonditional conditions).

Paraphrase

- Paraphrasing involves reformulating a statement without changing its meaning.

Examples:

- Case 1: John takes medication. → John is sick.
- Case 2: Sara, John's daughter, often accompanies John to the hospital. → John is Sara's father.

4.2.1. Logical Consequence/ Equivalence/ Conversational

An assertion A is a logical consequence of an assertion B if and only if, in every situation where A is true, B is also true. This is denoted as: A ⊨ B.

Logical Equivalence
Two propositions are said to be *logically equivalent* if their truth values (true or false) are always the same.

Conversational Implicature (Paul Grice)
Conversational implicature involves identifying the speaker's communicative intention, that is, an implicit deduction that can be drawn from a statement.

Example:

1. Sara knocks on John's door.
2. John does not answer.
3. Sara thinks that John is alone.

The Gricean Typology of Implicatures (causal theory of perception)

4.3. Grice Paul (1913-1988) : British philosopher of language and linguist, specializing in pragmatics and linguistics.

Grice first introduced the notion of implicature in , in an article on the causal theory of perception. Although the article's primary focus was not linguistic communication, Grice established a set of distinctions that would later become central to his theory.

He begins with the examination of an utterance such as:

"That looks red to me."

According to Grice, this utterance conveys a complex inference, suggesting that the speaker knows or believes that the object is not red, or that someone else has expressed doubt, or that the speaker is unsure of the object's color. The key idea is that the utterance conveys more than a simple assertion: it communicates implicit information or inferences beyond what is explicitly stated.

To analyze these inferences, Grice proposes a typology of non-logical inferences that a statement can generate, in addition to presuppositions already studied. He distinguishes:

- **The conventional implicatures**, which depend on specific linguistic terms.
- **The generalized conversational implicatures**, which arise in conversation according to typical situations.
- **The particularized conversational implicatures**, which depend on the specific context of the utterance.

Grice also defines criteria to distinguish these inferences:

- Their impact on the truth value of the statement.
- Dependence on specific terms.
- Persistence when the statement is reformulated (**detachability**).
- Possibility of cancellation.
- Context dependence.

4.4. Analysis of the Connection with the Sara and John Example

Recognized Communicative Intention

Grice emphasizes that for a message to convey an unnatural meaning, the listener must recognize the speaker's communicative intention.

In our example:

- Sara knocks on John's door.
- John does not answer.
- Sara thinks John is alone.

Here, Sara implicitly infers John's state from his lack of response. The information she derives depends on her interpretation of the signs (John's silence), which corresponds to the recognition of implicit intention according to Grice.

Unnatural Meaning and Implicature
Grice distinguishes between natural meaning (e.g., a directly observed fact, such as a photograph) and unnatural meaning (which implies an intention to be understood).

In this example, John's silence is not a purely natural fact; Sara interprets it as a sign that John is alone. This inference relies on conversational implicature, as she deduces an implicit belief from observable behavior.

Compatibility with Grice's Theory
As in Grice's drawing vs. photo example, information is successfully conveyed only if the interpretation accounts for the implicit communicative intention. Here, the communicative intention is not explicit, but Sara's inference follows the same pattern: the listener understands the situation by recognizing an implicit intention.

Silence Is Not Direct Evidence
John may not respond for many reasons: he could be busy, asleep, distracted, or simply not hear Sara. Sara's belief that John is alone is a personal and contextual interpretation—it is **not a logical consequence of silence**.

Role of Conversational Implicature
Conversational implicature does not involve deducing a certain truth; rather, it allows the listener to extract probable or suggested information from context and the assumed cooperation between speaker and hearer.

In our example, Sara infers that John is alone, but this remains an **implicit assumption**, not a fact.

Distinction between Inference and Implication
According to Grice, unnatural meaning requires the hearer to recognize the communicative intention. Here, John has no explicit intention to communicate that he is alone. Therefore, Sara's conclusion is an interpretation, not a certainty.

4.5. Peripheral Awareness according to Hubert L. Dreyfus (1929-2017)

According to American philosopher Hubert L. Dreyfus, human beings are capable of processing information that is not explicitly specified by the context. For example, if we assume that Sara knows John is not asleep at this time, she can deduce that John is in an emergency situation.

Dreyfus was particularly interested in phenomenology, existentialism, the philosophy of psychology, as well as the philosophy of literature and artificial intelligence. He argued that computers, lacking bodies, childhood experiences, and cultural practices, cannot become truly intelligent. Furthermore, since human knowledge is partly tacit, it cannot be fully articulated or incorporated into a computer program.

With the rapid development of deep learning, generative AI, and robots equipped with increasingly sophisticated sensors—especially from 2022 onwards—these arguments are being questioned. Nevertheless, the idea that general artificial intelligence is impossible because computers are **not situated in the world** is still defended by philosophers such as Ragnar Fjelland. He notes that while deep learning and big data represent the latest approaches, proponents claim they could eventually achieve artificial general intelligence (AGI).

A closer examination reveals that, despite impressive AI developments for specific tasks (sometimes referred to as narrow or applied AI), we are still far from achieving AGI. In principle, this may be impossible, echoing Dreyfus's argument that computers, being fundamentally detached from human embodied experience, cannot replicate general human intelligence.

The Central Question

How can we represent facts and knowledge expressed in natural language in the form of logical expressions?

4.6. Some Definitions & Basic Concepts

4.6.1. Reasoning

Reasoning is the ability to analyze reality, to perceive the relationships between beings and objects—whether present or absent—and to understand facts. It can also be defined as a sequence of propositions connected according to logical principles and organized in such a way as to reach a conclusion. More broadly, reasoning can be seen as a discourse which, starting from hypotheses, seeks to demonstrate that certain knowledge is true.

4.6.2. Syllogism

A syllogism, a term introduced by Aristotle, is a form of deductive reasoning that connects at least three propositions. The first two are called **premises**, while the third, derived from these premises, is the **conclusion**.

4.6.3. Axiom

An axiom is a proposition accepted as true without proof. All reasoning is built upon one or more axioms. A model may then be used to verify or refute the truth of a given proposition. Frequently, a proposition is itself composed of atomic propositions, for example:

- *"Sara knocks on the door"* is an atomic proposition.
- *"Sara thinks John is alone"* is a contingent statement, which may be true or false.
- *"John is not alone"* is not atomic and should instead be expressed as: **NOT (John is alone)**, where **NOT** is the symbol of negation.

4.7. Predicate Logic: First-Order Logic

Predicate logic is a branch of classical logic initiated by Aristotle and developed further through the work of Charles Pierce and Gottlob Frege. Its purpose is to represent the meaning of statements expressed in natural language.

Thanks to the work of Richard Montague, a philosopher and logician of the 1960s, Noam Chomsky's view of natural language as a formal system was extended by introducing semantics into predicate calculus. Unlike Chomsky, Montague made it possible to formalize the truth conditions of statements by expressing them in the language of predicate logic.

Translating a sentence first requires identifying the predicates and their arguments, then linking the predicates together.

Choosing the Predicate

Consider the following statements:

- (EN1) *David turns on the television.*
- (EN2) *David turns on the television in the living room using the remote control.*

In linguistics, Davidsonian and neo-Davidsonian theories hold that a verb like *to turn on* expresses a relation between two arguments, while adverbs act as modifiers of the verb (Montague, 1973).

Davidson goes further by arguing that a verb expressing an action, such as *to turn on*, actually involves three arguments. Two are **nominal arguments**, and the third is an **event argument** (*e*), which is implicitly quantified using the existential quantifier ∃. Adverbs, in this framework, are treated as predicates of the event argument.

Thus, the two statements can be represented as follows:

- ∃e(TurnOn(e, David, Television))

- ∃e(TurnOn(e, David, Television)) ∧ In(LivingRoom, e) ∧ With(RemoteControl, e)

where *e* is a variable representing an event.

The overall meaning of (EN1) then becomes: *There exists an event e such that e represents David turning on the television.*

Constants vs. Variables

Take the statements: *Sara knocks on the door* and *John does not answer.*

The terms *Sara* and *John* denote two individuals in the world, called **terms** in predicate logic. These terms are **constants**, usually denoted by lowercase letters (e.g., *s* → *Sara*, *j* → *John*). In predicate logic, a constant always denotes the same individual in the world.

By contrast, the individual *door* is arbitrary — it could be any door in the domain of description (front door, bedroom door, etc.). In this case, we use a **variable**, for example *x*, to denote the door anonymously, without specifying which door.

Variables (*x, y, z*, etc.) therefore allow us to refer to individuals in the world anonymously, while constants always refer to the same specific individual.

4.8. Predicate Logic (PL)

Predicates are represented by symbols, usually borrowed from natural language, although other symbols can also be used. The arguments of a predicate appear in parentheses and are separated by commas. The interpretation of a predicate corresponds to a subset of individuals in the domain that satisfy the property represented by that predicate.

Model and Interpretation Function

A model $\mathcal{M}$ used to represent the states of the world is composed of two elements. First, a set of individuals, called the domain and denoted D , which makes it possible to model the entities of the world. Then, an interpretation function I that maps the individuals of the world to the vocabulary of the knowledge representation language, in this case predicate logic.

Syntax of Predicate Logic

The syntax of predicate logic is based on several sets. First, we have a set of variables

$V = \{x, y, z, \dots\dots\}$. Then, a set of individual constants $C = \{a, c_1, \dots\}$, followed by a set of predicates $P = \{assists, fatherOf, installedIn, \dots\}$. In addition, logical connectives are used to construct complex formulas: conjunction (∧), inclusive disjunction (∨), implication (⇒), equivalence (⇔), negation (ך), as well as parentheses for grouping expressions. Two quantifiers are also used: the universal quantifier (∀), which designates an arbitrary variable

representing all individuals, and the existential quantifier (∃), which designates an arbitrary variable representing at least one individual.

Examples

Consider the following formula:

$\exists x\big(Human(x)\big)\wedge assists(Kompai,x)\wedge Robot(Kompai)$ …….(1)

This formula can be translated as: *a robot named Kompai assists at least one human*, or equivalently, *there exists a human who is assisted by the robot Kompai*. In this expression, **Kompai** is a constant, the variable (x) represents a human (for example John or David), and the predicate **assists** is a binary predicate.

Another example is given by the formula: $\forall x(assists(Kompai,x) \longrightarrow Human(x)) \wedge)$

Or again:

$$\neg\exists x(Robot(Kompai) \wedge Human(x) \ \wedge \ assists(Kompai,x))$$

which means that Kompai assists only humans.

4.9. Description Logics (DL) & W3C Web Ontology Language

The ultimate goal of Artificial Intelligence is to design systems that exhibit intelligent behavior.
To achieve this, three elements are essential:

1. Acquiring knowledge about the world or the specific application domain.
2. Representing that knowledge in a structured form.
3. Reasoning with the knowledge to derive conclusions and make decisions.

Hence, a suitable logical formalism is needed to represent knowledge and enable reasoning.

Knowledge representation approaches can broadly be divided into two categories:

- Logic-based formalisms
- Non-logic-based formalisms

The main distinction lies in their treatment of reasoning (entailment/implication)

First-Order Logic (FOL) is often used as a rule-based language. Its advantage is the ability to express both general statements and specific facts, and to make inferences (e.g., through quantifiers that refer to sets of objects without having to enumerate them).

At first glance, FOL appears adequate for describing ontologies. However, it is also too expressive and complex:

- Some forms of reasoning in FOL are undecidable.
- FOL representations can become heavy and cumbersome.

Therefore, researchers have sought fragments of logic that avoid these issues while preserving reasoning power. This leads us to Description Logics (DL):

- A family of knowledge representation languages.
- Most DLs are well-defined fragments of FOL.
- They are decidable, meaning reasoning procedures always terminate.
- They support inference, allowing new knowledge to be derived from existing facts.

In DL, knowledge bases are typically divided into:

- **TBox** (Terminological Box): the conceptual schema (classes, roles).
- **ABox** (Assertional Box): the instance data (individuals, facts).

Together, the TBox and ABox form the **knowledge base.**

4.9.1. Description Logics (DL) vs First-Order Logic (FOL)

It is important to distinguish Description Logics (DL) from First-Order Logic (FOL).

Problem with First-Order Logic (FOL):

- Very expressive: It allows the formalization of virtually anything (complex relations, recursion, cardinalities, etc.).
- But: Full FOL is undecidable → we cannot guarantee that a reasoning algorithm will always terminate with an answer (sometimes it may loop infinitely).

Therefore, Description Logics (DL) were designed to strike a balance between:

- Expressivity: The ability to represent rich knowledge.
- Computability (decidability): Ensuring that inference algorithms always terminate in finite time.

To achieve this, certain aspects of FOL were restricted.

Syntactic restriction (example 1)

- No arbitrary quantification over multiple variables (it is usually limited to a single variable linked by a role).

  FOL :

  $$\forall x \forall y (canExchangeDataWith(x, y) \wedge canExchangeDataWith(y, x) \Rightarrow Symmetric(canExchangeDataWith))$$

  DL (allowed):
  Symmetry is expressed through a **role property**:

  $$CanExchangeDataWith \equiv (CanExchangeDataWith) - 1$$

  (Interpreted as: if device x can exchange data with device y, then device y can exchange data with device x).

  Only one variable is “bound” at a time through the role.

- No higher-arity functions

  FOL:

  $$Treats(x, Therapy(y, z))$$

  Here, the predicate *Therapy(y,z)* has arity 2 (for example: a therapy defined by a drug y and a dosage z), and it is used as an argument of another function (too complex).

  DL (allowed):
  We keep only **binary roles**:

  $$Treats(x, t)\ or\ Therapy(t)$$

  There is no function *Therapy(y,z)*, only a therapy instance t.

Syntactic restriction (example 2)

$$Prescribes\big(doctor, Treatment(patient, drug)\big)$$

Here, *Treatment(patient, drug)* is a function of arity 2 (patient + drug). It is then used as an argument of *Prescribes*. This kind of nested structure is **not allowed** in DL.

DL (allowed form)

We only keep **binary roles**:

$$Prescribes(doctor, t) \land Treatment(t) \land AppliesTo(t, patient) \land UsesDrug(t, drug)$$

Interpretation:

- ✓ The doctor prescribes a therapy t.
- ✓ t a Treatment.
- ✓ T applies to a patient.
- ✓ t uses a drug.

- **Choice of logical** operators
- We keep conjunction (⊓), disjunction (⊔), limited negation (¬), existential quantification (∃), and universal quantification (∀).
- We add cardinality restrictions (≥, ≤).
- But we avoid combinations that make the system undecidable

1. Description Logic (DL) Formalism

Description Logic (DL) is inspired by semantic networks and conceptual graphs. It is a decidable fragment of first-order logic, designed to represent domain knowledge in a structured and formal way. DL is based on three fundamental entities: **concepts** (or classes), **roles** (or relations), and **individuals** (or instances).

By convention, concepts start with a capital letter, such as *Robot*, *Human*, *House*, or *Disease*. Roles, on the other hand, start with a lowercase letter, such as *assists*, *hasAssistant*, or *hasDisease*. The basic syntax includes two forms:

- **C(a)** or **a:C**, which expresses that an individual *a* is an instance of a concept *C*.
- **R(a,b)**, which expresses that there is a relation *R* between individuals *a* and *b*.

4.9.2. The TBox: Vocabulary and Axioms

The TBox (Terminological Box) introduces the terminology of the domain, i.e., the vocabulary consisting of concepts and roles. It represents intentional knowledge**,** meaning the general structure of the domain.

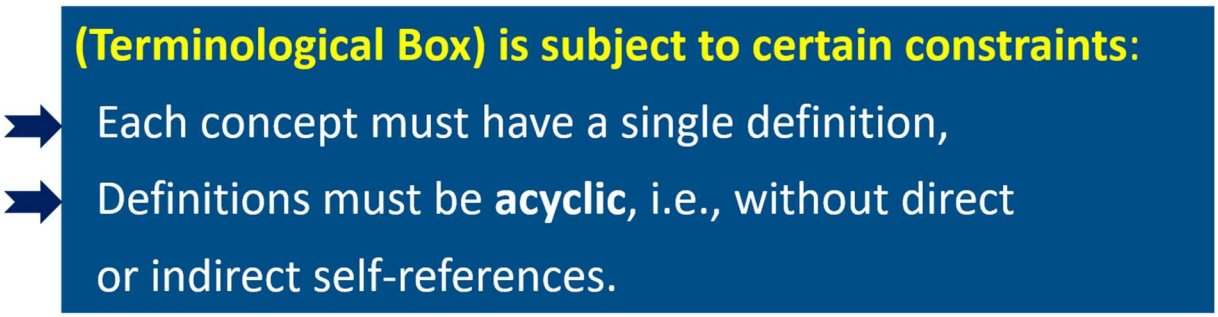
The basic form in a TBox **is a** concept definition axiom**,** built from already defined concepts. For example:

$$Robot \equiv Agent \sqcap \neg Human$$

This axiom defines the concept *Robot* as any agent that is not human. It expresses a logical equivalence, providing both necessary and sufficient conditions.

A role such as **nearTo(Person,Object)** allows us to define a proximity relation between a person and an object.

The TBox relies on a set of logical constructors such as
$\wedge$,$\vee$,$\sqsubseteq$,$\sqsupseteq$,$\perp$,$\equiv$, $\sqcup$



4.9.3. The ABox: Facts and Assertions

The ABox (Assertional Box) contains factual assertions, i.e., knowledge about specific instances in the domain. For example:

- Agent(agent1)
- Robot(Kompai)
- Human(David)

The assertion implicitly expresses, via the TBox, that *agent1* is a robot:

$$\neg Human \sqcap Agent(agent1)$$

Thus, the ABox describes concrete facts, consistent with the definitions given in the TBox. Together, the TBox and ABox form the knowledge base, where reasoning can combine logical and non-logical aspects.

Formal Semantics

Let $\mathcal{I}$ be an interpretation model and a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ is the TBox and $\mathcal{A}$ the ABox component. An individual a is an instance of concept C with respect to the knowledge base $\mathcal{K}$ and we write $\mathcal{K} \models a : C$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ holds for all models of $\mathcal{K}$.

The pair of individuals (a, b) is an instance of role r with respect to $\mathcal{K}$ and we write $\mathcal{K} \models (a, b) : r$ if $\langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in r^{\mathcal{I}}$ holds for all models of $\mathcal{K}$.

4.9.4. Minimal Description Logic

Minimal Description $\mathcal{AL}$ logics form a hierarchical family of formalisms, where each language extends the previous one by adding new logical constructors. Thus, each step toward a more expressive logic enables the representation of richer knowledge, while still preserving decidability for automated reasoning

$\mathcal{AL}$ Introduced by $Schmidt - Schau\beta$ and $G.Smolka$ in 1991 as a minimal language with practical interest. Other languages of this family are extensions of $\mathcal{AL}$ logic.

minimal logic since it is the least expressive of the logics

C,D $\rightarrow$ A (atomic concept)
- $\bot$ *bottom concept*, represents the impossible concept
- $\top$ *top concept*, represents the universal concept
- $\neg A$ atomic negation
- $C \sqcap D$ intersection of concepts
- $\exists R.\top$ value restriction
- $\forall R.C$ limited universal quantifier

By definition every class $C \sqsubseteq \top$

Example

- Robot, Artifact, Human → (atomic concepts)
- hasAssistant → (role)
- Negation is only allowed on an atomic concept.
  ¬Robot : Represents the class of instances that are not *Robot*
- The existential quantifier cannot be used with a concept.

However, the universal concept $\top$ is admitted.
Human $\sqcap \exists$hasAssistant.$\top$
represents the class of individuals having an assistant.

The following description is not allowed in minimal logic $\mathcal{AL}$:
Human $\sqcap \exists$hasAssistant.*Robot*.
There is no way to express that a human is assisted by at least one Robot.
Note that we cannot express which assistant it is.

- The impossible concept is used to impose a restriction on a relation.

Example : representing humans : having no assistants
Human $\sqcap \forall$*hasAssistant*.$\bot$

- The interpretation $(\forall \mathcal{R}.C)$ clearly means :

♦ That an individual i is related only to class C through relation $\mathcal{R}$.
♦ It is possible that individuals not participating at all in relation $\mathcal{R}$ can also be represented by this description.

4.9.5. Semantics of the $\mathcal{AL}$ language

To define semantics, we consider an interpretation $\mathcal{I}$ and a non-empty interpretation domain $\triangle^{\mathcal{I}}$. An interpretation function assigns to each atomic concept description $\mathcal{A}$ a subset $A^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}}$, and for each role R a binary relation $R^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$. Thus, this interpretation function is described as follows for other descriptions :

- $\top^{\mathcal{I}} = \triangle^{\mathcal{I}}$;
- $\bot^{\mathcal{I}} = \varnothing$;
- $\neg A^{\mathcal{I}} = \triangle^{\mathcal{I}} \setminus A^{\mathcal{I}}$;
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \sqcap D^{\mathcal{I}}$;
- $(\forall R.C)^{\mathcal{I}} = \{a \in \triangle^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \longrightarrow b \in C^{\mathcal{I}}\}$;
- $(\exists R.\bot)^{\mathcal{I}} = \{a \in \triangle^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}}\}$;

Equivalent Classes

Two classes $\mathcal{C}$ and $\mathcal{D}$ are equivalent, and we write $\mathcal{C} \equiv \mathcal{D}$, if $C^{\mathcal{I}} \equiv C^{\mathcal{I}}$ for every interpretation $\mathcal{I}$.

- **For example** :

The description of the class of humans assisted by human doctors (assuming there could one day be robot doctors) can be expressed in two different ways :

$\forall assist.Human \sqcap \forall assist.Doctor$ is equivalent to expressing
$\forall assist.(Human \sqcap Doctor)$

■ **Reminder** : With the minimal logic we cannot describe the class of non-human doctors.

$\neg(Human \sqcap Doctor)$

4.10. Extensions of Description Logics

($\mathcal{AL}$) is generally not expressive enough to represent the ontologies we use in practice. There are three ways to increase the expressiveness of logic $\mathcal{AL}$:

- Add role constructors ;
- Add concept constructors ;
- Define constraints on the interpretation of roles ;

- Description logics come in several languages differing in their degree of expressivity and complexity
- Logic $\mathcal{AL}$ can be extended using the following constructors :

$$(\mathcal{ALC}|\mathcal{S})[\mathcal{H}]|[\mathcal{SR}])[\mathcal{O}][\mathcal{I}][\mathcal{F}|\mathcal{N}|\mathcal{Q}].$$

Note that the sub-language of $\mathcal{AL}$ obtained by eliminating atomic negation is called $\mathcal{FL}^-$, and the sub-language of $\mathcal{FL}^-$ obtained by eliminating restricted existential quantification is called $\mathcal{FL}0$. These two languages were the first to serve as a basis for establishing theoretical results on the complexity of subsumption.

Languages	Constructors	Concepts
$\mathcal{AL}$	Names of atomic concepts	A, C
	Atomic roles	$\mathcal{R}_1, \mathcal{R}_2$
	Intersection (concept conjunction)	A ⊓ C
	Universal quantifier (value restriction)	$\forall \mathcal{R}.C$
	Existential quantifier	$\exists \mathcal{R}$
	Top (subsumes all concepts)	$\top$
	Bottom (has no instance)	$\bot$
	Negation	$\neg A$
$\mathcal{C}$	Unrestricted negation	$\neg\exists \mid \neg\forall \ldots$
$\mathcal{U}$	Union (Disjunction)	$A \sqcup C$
$\mathcal{E}$	Typed existential quantifier	$\exists \mathcal{R}.C$
$\mathcal{R}$	Role conjunction	$\mathcal{R}_1, \mathcal{R}_2$
$\mathcal{S}$	Role transitivity	$\mathcal{R}_1, \mathcal{R}_2$ and $\mathcal{R}_3$
$\mathcal{N}$	Cardinalities	$(\geq n\mathcal{R})(\leq n\mathcal{R})$
$\mathcal{H}$	Role hierarchy	$\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$
$\mathcal{I}$	Inverse role	$\mathcal{R}^-$
$\mathcal{Q}$	Qualified number restriction	$(\geq n\mathcal{R}.A)(\leq n\mathcal{R}.A)$
$\mathcal{O}$	Named individuals $\{a_1, a_2, a_3, \ldots, a_n\}$	$\{a_1^I, a_2^I, a_3^I, \ldots, a_n^I\}$

Table – Correspondence between constructors of different description languages

The logic $\mathcal{ALU}$

Thus, the logic $\mathcal{ALU}$ is nothing more than the logic $\mathcal{AL}$ enriched with the constructor $\sqcup$.

- The union of concepts (indicated by the letter $\mathcal{U}$) is written as follows : C $\sqcup$ D and interpreted as $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \sqcup D^{\mathcal{I}}$.
- To describe that a Robot only has Screens or Microphones, it can be expressed as follows :
  Robot $\sqcap \forall has.(Screen \sqcup Microphone)$.
  The individuals described by this class must belong to the class Robot. Then, if an individual of the class Robot uses the role *has*, it can only be linked to instances of the classes Screen or Microphone. Note that a Robot individual may not use the property *has*.

The logic $\mathcal{ALE}$

To increase the expressiveness of $\mathcal{AL}$, the letter $\mathcal{E}$ is added, indicating that an individual must have at least one relation with another individual, specifying the class of the latter. In contrast, with $\mathcal{AL}$, the existential quantifier can only be used with the universal concept and cannot be associated with an arbitrary concept.

- The semantics of the full existential quantifier is expressed as : $(\exists R.C)^{\mathcal{I}} = \{a \in \triangle^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
- Thus, we can represent the class of humans who have at least one assistant Robot as :
  Human $\sqcap \exists assist$.Robot

The functional relation $\mathcal{ALF}$

1. A property (relation) defined as a functional property means that an individual of a class can be associated with one and only one individual of another class. Formally, for all individuals x, y, z, if
   P(x,y) and P(x,z) $\Rightarrow$ y = z
2. In the TBox component, a relation *assist* can be defined as functional as follows : $\top \sqsubseteq \leq 1 assist$

The inverse relation : $\mathcal{ALI}$

1. Provides the ability to express that a role is the inverse of another role.
2. The semantics of this constructor is expressed as follows :
   $(R^{-})^{\mathcal{I}} = \{(x,y) \mid (y,x) \in R^{\mathcal{I}}\}$
3. For example, the role *assistedBy* is the inverse role of *assist*. Assist (Kompai, John) is equivalent to saying assistedBy (John, Kompai), and we write :
   assist $\equiv$ assistedBy^{-}.

Cardinality Restriction $\mathcal{ALN}$

- This allows expressing the number of times an individual participates in a property (role). This constructor is indicated by the letter $\mathcal{N}$ and the two connectors $\leq n\mathcal{R}$ and $\geq n\mathcal{R}$. The semantics of cardinality restriction is expressed as :
  1. $(\geq n\mathcal{R})^{\mathcal{I}}=\{a \in \triangle | \mid \{b \mid (a,b) \in (\mathcal{R})^{\mathcal{I}} \mid \geq n\}$
  2. $(\leq n\mathcal{R})^{\mathcal{I}}=\{a \in \triangle | \mid \{b \mid (a,b) \in (\mathcal{R})^{\mathcal{I}} \mid \leq n\}$
  3. For example, describing that a person has fewer than 2 assistants, without specifying who : $\text{Human} \sqcap \leq 2 hasAssistant.\top$

Qualified Cardinality Restriction $\mathcal{ALQ}$

- It is characterized by the addition of the letter $\mathcal{Q}$. This constructor allows expressing not only the number of times an individual participates in a relation but also to which class it is linked. The semantics is expressed as :
  1. $(\geq n\mathcal{R}.C)^{\mathcal{I}}=\{a \in \triangle | \mid \{b \mid (a,b) \in (\mathcal{R})^{\mathcal{I}} \wedge b \in C^{\mathcal{I}} \mid \geq n\}$
  2. $(\leq n\mathcal{R}.C)^{\mathcal{I}}=\{a \in \triangle | \mid \{b \mid (a,b) \in (\mathcal{R})^{\mathcal{I}} \wedge b \in C^{\mathcal{I}} \mid \leq n\}$
- Example, describing that a person has at most 2 assistant robots : $\text{Human} \sqcap \leq 2 hasAssistant.Robot.$
- Example, describing that a robot is equipped with a single touchscreen : $\text{Robot} \sqcap \leq 1 equipped.Touchscreen.$
- Example, describing that a robot is equipped with exactly 2 cameras : $\text{Robot} \sqcap \leq 2 equipped.Camera \sqcap \geq 2 equipped.Camera.$

Full Negation $\mathcal{ALC}$

1. This constructor is designated by the letter $\mathcal{C}$.
2. Allows expressing that an individual is not assisted by a Robot : $\neg\exists assist.Robot$ $\neg(Human \sqcap Agent)$

Named Individual Enumeration $\mathcal{ALO}$

1. It is enough to add the letter $\mathcal{O}$
2. Example : named individuals can be added as follows : $\{Alice, John\}$ or $\{Kompai, Pepper\}$

Role Inclusion (Hierarchies) $\mathcal{ALH}$

1. It is enough to add the letter $\mathcal{H}$
2. Example : if we define a hierarchy of roles, e.g., $monitors \sqsubseteq assist \sqsubseteq help$, it means every individual related via *assist* is also related via *help*.

Caution

New DL languages are obtained by adding different constructors to the basic language $\mathcal{AL}$:
Example : concept union $\rightarrow \mathcal{ALU}$
Full existential quantification + cardinality restriction $\rightarrow \mathcal{ALEN}$
Some constructors are not independent :
Extended negation does not increase expressivity.
$\mathcal{ALC}$ and $\mathcal{ALUE}$ are equivalent in expressivity.
Full existential quantification and concept union are not necessary if extended negation is available.
To understand this properly, it is important to first recognize that there are several logical equivalences in these languages :

$$\neg(C \sqcup D) \equiv \neg C \sqcap \neg D$$

$$\neg(C \sqcap D) \equiv \neg C \sqcup \neg D$$

$$\neg\exists R.A \equiv \forall R.\neg A$$

$$\neg\forall R.A \equiv \exists R.\neg A$$

$$\neg\neg C \equiv C$$

Intuitive interpretation of the equivalences :
Example : Class of parents without daughters $\rightarrow$
$\neg\exists aEnfant.Femme \equiv \forall aEnfant.\neg Femme$
Class of those whose children are not all daughters $\rightarrow$
$\neg\forall aEnfant.Femme \equiv \exists aEnfant.\neg Femme$
Consequences :
Concept union can be expressed using only negation and intersection :
$C \sqcup D \equiv \neg(\neg C \sqcap \neg D)$
Existential quantification can be expressed using universal quantification :
$\exists R.C \equiv \neg\forall R.\neg C$
These formulations are correct but more complex and less intuitive.

4.11. Inferences in Description Logic (DL)

Description logic inferences manipulate elements from both TBox and ABox components. To ensure predictable system behavior, these inferences should be decidable.

TBox inferences:

- **Subsumption:** Determines if one concept is more general than another.

  A concept C is **subsumed** by a concept D if every instance of C is also an instance of D.

  $T \sqsubseteq C \sqsubseteq D$

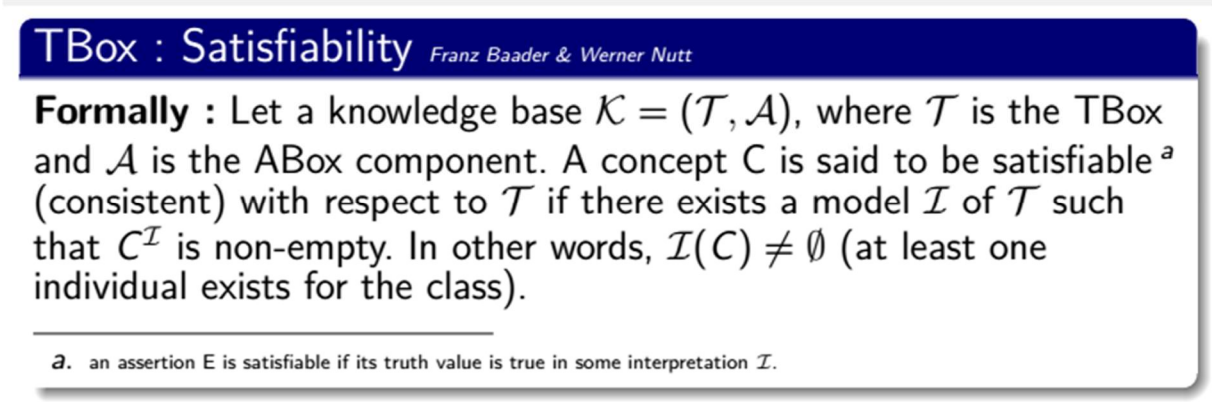
  TemperatureSensor ⊑ Sensor (Every TemperatureSensor is by definition a Sensor).

- **Satisfiability:** Checks whether a concept can possibly have instances.

  A concept is satisfiable with respect to a terminology if there exists at least one possible entity in the domain that can belong to that concept.

  Concept: Sensor ⊓ ¬Sensor

  Interpretation: This concept is unsatisfiable, because no IoT device can be both a Sensor and not a Sensor.



- **Equivalence:** Identifies concepts that are semantically identical.

  Two concepts C and D are equivalent if they have exactly the same instances in all models of the terminology.

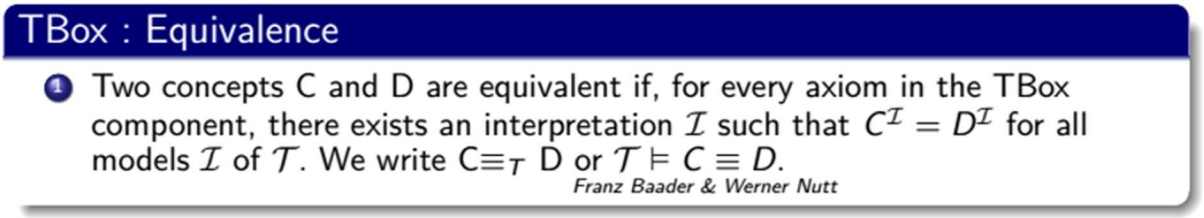
  TempMonitor ≡ TemperatureSensor

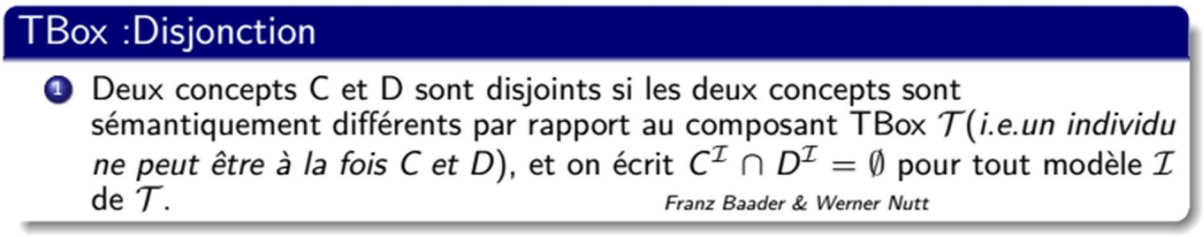
- **Disjointness:** Ensures that certain concepts have no shared instances.

  Two concepts C and D are disjoint if no entity can belong to both concepts simultaneously.

  $Camera \sqcap Thermostat \sqsubseteq \bot$

No IoT device can be both a Camera and a Thermostat at the same time





4.11.1. ABox inferences:

- **Instance checking:** Verifies if a specific individual belongs to a concept.

  Verifying whether a particular individual belongs to a given concept according to a knowledge base : K ⊨ a : C

  Individual a is an instance of concept C in all models of the knowledge base K.

  Knowledge Base (TBox + ABox):

  - TBox: SmartSensor ≡ Sensor ⊓ ∃monitors.Temperature
  - ABox: Temperature(device1)

  Question: Is device1 an instance of SmartSensor?

  Answer: Yes, because Temperature ⊑ SmartSensor according to the TBox.

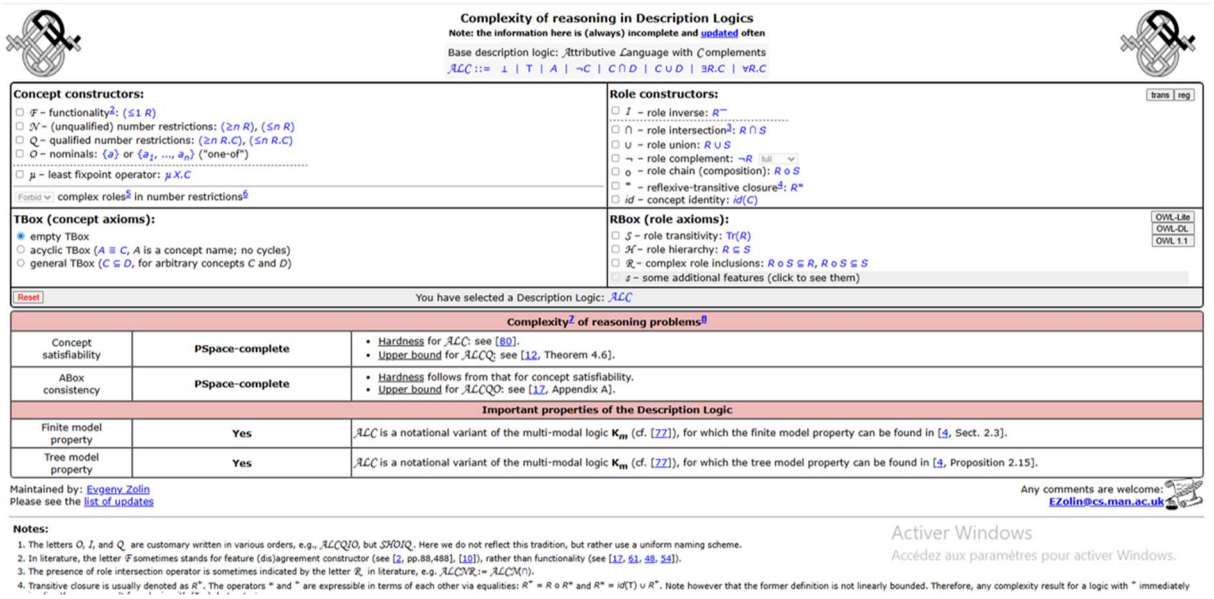
- **Consistency checking:** Ensures the knowledge base has no contradictions.

  Ensure that the ABox does not contain any assertion C(a) for which it is impossible for a to belong to the class C.

  If the ABox contains SmartSensor(device1) and also implies device1 ⊑ ¬SmartSensor via the TBox definitions, the ABox is inconsistent.

  Consistency checking guarantees that all instance assertions in the ABox are logically possible with respect to the defined concepts and roles.

- **Role verification:** Checks relationships between individuals. Involves checking whether the roles in an assertion R(a,b) are valid, i.e., whether a and b correspond to existing individuals.
- **Realization:** Identifies the most specific concepts an individual belongs to.



Complexity of reasoning in Description Logics

Note: the information here is (always) incomplete and updated often

Base description logic: $\mathcal{A}$ttributive $\mathcal{L}$anguage with $\mathcal{C}$omplements

$\mathcal{ALC} ::= \bot \mid \top \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C$

Concept constructors:

- $\mathcal{F}$ – functionality[2]: $(\leq 1\ R)$
- $\mathcal{N}$ – (unqualified) number restrictions: $(\geq n\ R)$, $(\leq n\ R)$
- $\mathcal{Q}$ – qualified number restrictions: $(\geq n\ R.C)$, $(\leq n\ R.C)$
- $\mathcal{O}$ – nominals: $\{a\}$ or $\{a_1, \ldots, a_n\}$ ("one-of")
- μ – least fixpoint operator: $\mu X.C$

Forbid complex roles[5] in number restrictions[6]

Role constructors: trans reg

- $\mathcal{I}$ – role inverse: R^-
- $\sqcap$ – role intersection[3]: $R \sqcap S$
- $\cup$ – role union: $R \cup S$
- $\neg$ – role complement: $\neg R$ (all)
- $\circ$ – role chain (composition): $R \circ S$
- * – reflexive-transitive closure[4]: R^*
- id – concept identity: $id(C)$

TBox (concept axioms):

- empty TBox
- acyclic TBox ($A \equiv C$, A is a concept name; no cycles)
- general TBox ($C \sqsubseteq D$, for arbitrary concepts C and D)

RBox (role axioms): OWL-Lite OWL-DL OWL 1.1

- $\mathcal{S}$ – role transitivity: $\mathrm{Tr}(R)$
- $\mathcal{H}$ – role hierarchy: $R \sqsubseteq S$
- $\mathcal{R}$ – complex role inclusions: $R \circ S \sqsubseteq R$, $R \circ S \sqsubseteq S$
- $\mathcal{s}$ – some additional features (click to see them)

Reset — You have selected a Description Logic: $\mathcal{ALC}$

Complexity[7] of reasoning problems[8]		
Concept satisfiability	PSpace-complete	• Hardness for $\mathcal{ALC}$: see [80]. • Upper bound for $\mathcal{ALCQ}$: see [12, Theorem 4.6].
ABox consistency	PSpace-complete	• Hardness follows from that for concept satisfiability. • Upper bound for $\mathcal{ALCQO}$: see [17, Appendix A].
Important properties of the Description Logic		
Finite model property	Yes	$\mathcal{ALC}$ is a notational variant of the multi-modal logic $\mathbf{K}_m$ (cf. [77]), for which the finite model property can be found in [4, Sect. 2.3].
Tree model property	Yes	$\mathcal{ALC}$ is a notational variant of the multi-modal logic $\mathbf{K}_m$ (cf. [77]), for which the tree model property can be found in [4, Proposition 2.15].

Maintained by: Evgeny Zolin
Please see the list of updates

Any comments are welcome: EZolin@cs.man.ac.uk

Notes:

1. The letters $\mathcal{O}$, $\mathcal{I}$, and $\mathcal{Q}$ are customary written in various orders, e.g., $\mathcal{ALCQIO}$, but $\mathcal{SHOIQ}$. Here we do not reflect this tradition, but rather use a uniform naming scheme.
2. In literature, the letter $\mathcal{F}$ sometimes stands for feature (dis)agreement constructor (see [2, pp.88,488], [10]), rather than functionality (see [17, 61, 48, 54]).
3. The presence of role intersection operator is sometimes indicated by the letter $\mathcal{R}$ in literature, e.g. $\mathcal{ALCNR} := \mathcal{ALCN}(\sqcap)$.
4. Transitive closure is usually denoted as R^+. The operators * and $^+$ are expressible in terms of each other via equalities: $R^+ = R \circ R^*$ and $R^* = id(\top) \cup R^+$. Note however that the former definition is not linearly bounded. Therefore, any complexity result for a logic with * immediately

Activer Windows
Accédez aux paramètres pour activer Windows.

4.12. Towards OWL (Web Ontology Language)

Description Logics (DLs) are formal languages with a well-defined semantics. They clarify the relationship between the syntax of the language and the models representing a domain. DLs formalize the meaning of the language, enabling the verification of class consistency, ontology coherence, and implication relationships. The ability to provide such reasoning support was a key design goal for OWL. Description Logic had a strong influence on the design of OWL, particularly in the formalization of its semantics, the choice of language constructors, and the integration of data types and data values.

OWL (Web Ontology Language) was developed as a new ontology language for the Semantic Web. Ontologies are intended to help automate (through intelligent agents) the processes of accessing information on the Web. In particular, ontologies provide structured vocabularies that explain the relationships between different terms, allowing intelligent agents—and humans—to interpret their meaning flexibly and unambiguously. Terms defined within ontologies can be used in semantic markup to describe the content and functionality of resources accessible on the Web.

Introduction to OWL

To preserve both the expressiveness of first-order logic (FOL) and ensure decidability, it is necessary to use an appropriate modeling language. It is in this context that OWL 1 and OWL 2 are situated.

The goal is to represent ontologies using logic, relying on description logic, which lies between propositional logic and first-order logic. This approach allows maintaining the ability to represent complex knowledge while enabling automated reasoning over that knowledge.

OWL (Web Ontology Language) was developed by the W3C Web-Ontology Working Group to represent information about categories of objects and their relationships—in other words, to formalize ontologies. It also allows describing detailed information about the objects themselves, i.e., data. OWL integrates into the Semantic Web language stack, including XML and RDF, while remaining compatible with pre-existing languages such as SHOE, OIL, and DAML-ONT.

4.13. OWL Predecessors

Before the development of OWL, several earlier ontology languages and frameworks laid the groundwork for representing knowledge on the Web. These predecessor languages introduced key concepts such as frames, XML syntax, and the use of URIs, while addressing limitations in expressiveness and reasoning capabilities. Understanding these earlier efforts—SHOE, DAML-ONT, OIL, and DAML+OIL—provides insight into the evolution and design choices that shaped OWL.

4.13.1. SHOE (Simple HTML Ontology Extensions)

SHOE was the first attempt to create an ontology language for the Web. It was based on frames with XML syntax and could be integrated into existing HTML documents. SHOE introduced the use of URIs for names and allowed the import of other ontologies with version management and compatibility.
However, SHOE was not based on RDF and lacked formal semantics, which influenced the design of OWL

4.13.2. DAML-ONT

DAML-ONT was developed by the DARPA program in 1999 to extend RDF with constructors inspired by object-oriented and frame-based languages. It relied on RDFS but suffered from insufficient semantic specification, leading to ambiguities for both humans and machines. Properties had a global scope, which complicated their understanding.

4.13.3. OIL (Ontology Inference Layer)

OIL, designed in parallel by a group of European researchers, combined elements of description logic (SHIQ), frame-based languages, and Web standards such as XML and RDF. It did not focus on RDF semantics, which were still being defined at the time..

4.13.4. DAML+OIL

DAML and OIL merged to create DAML+OIL, which served as the basis for OWL. This merger allowed the strengths of both languages to be combined but also introduced challenges, particularly with the use of RDF/XML as the official syntax of OWL.

4.13.5. Main Features of OWL

- Based on Description Logic: OWL allows ontologies to be formalized and logical inferences to be made.
- URIs for concepts, individuals, and relations: As in RDF, each entity has a unique identifier.
- Class hierarchy: Classes are organized according to subsumption and can be defined by intersections, unions, or complements.
- Property hierarchy and restrictions: OWL allows defining property domains and ranges, as well as transitive, symmetric, functional, or inverse properties.
- Control over individuals: It is possible to indicate class membership for individuals, declare disjoint classes or individuals, and specify the existence or absence of property values.
- Automated reasoning: OWL is compatible with reasoning engines (Pellet, Fact++) to infer new knowledge from data and ontologies.

4.14. The OWL Language and Its Evolutions

1. OWL, proposed by the W3C in 2004, initially consisted of three sub-languages: OWL-Lite, OWL-DL, and OWL-Full.
2. OWL-Lite corresponds to the SHIF(D) family, suitable for simple ontologies with limited restrictions.
3. OWL-DL corresponds to the SHOIN(D) family, offering higher expressive power, with a distinction between roles linking two individuals and roles linking an individual to a literal (Data property).
4. OWL-Full combines OWL-DL and RDF, ensuring full compatibility with RDF. It is not decidable.

In 2009, OWL 2 was launched, enhancing expressiveness and replacing the division into three sub-languages with profiles:

- **OWL 2 EL:** Optimized for ontologies with a large number of classes and properties. Inferences (consistency, subsumption, classification) can be performed in polynomial time.

- **OWL 2 QL:** Suited for ontologies with many individuals, similar to UML or Entity-Relationship models, ideal for interfacing a knowledge base with a database.
- **OWL 2 RL:** Restricts ontologies to forms that can be translated into first-order logic rules.

4.15. Ontologies in OWL 2

Between 2009 and 2012, OWL 2 was developed to overcome the limitations of the first version, OWL 1. OWL 1 DL was based on the expressive description logic SHOIN(D) and was compatible with the existing Semantic Web, particularly RDF. However, it was difficult to satisfy both of these constraints simultaneously: sufficient expressiveness for complex ontologies and compatibility with RDF standards. Two approaches had been considered: introducing sub-languages such as OWL Lite and OWL DL, and extending the RDF model theory.

In OWL 2, every document constitutes an ontology, identified by a unique URI. An ontology can define prefixes to simplify notation and can import other ontologies to reuse existing definitions. In functional syntax, an ontology is represented by the term Ontology(...), which contains all statements describing the classes, properties, and relationships of the ontology.

To define class hierarchies, two types of axioms are possible: class declaration (which simply establishes the existence of a class) and the direct use of the property rdfs:subClassOf. OWL introduces the specific vocabulary owl:Class because the interpretation of classes is more restrictive than in RDF (thus rdfs:Class is not used).

Similarly, OWL allows defining property hierarchies, distinguishing two types:

- **ObjectProperty:** The property value is an individual belonging to a class.
- **DataProperty:** The property value is a literal (text, number, date, etc.).

To construct the ABox, OWL provides three types of assertions:

- **ClassAssertion:** Specifies the class to which an individual belongs.
- **ObjectPropertyAssertion:** Links an individual to another individual.
- **DataPropertyAssertion:** Links an individual to a literal value.

This organization allows modeling both the terminological structure (TBox) and specific facts (ABox) in a coherent and formal language, ready to be exploited by reasoning engines.

4.16. RDF(S) vs OWL : Semantics of the OWL 2 Model Theory

OWL offers richer expressive capabilities than RDF(S). For example, in OWL, it is possible to declare that two classes are equivalent or disjoint, and this can be done for an unlimited number of classes simultaneously using the terms EquivalentClasses and DisjointClasses. In RDF, representing disjoint classes is not straightforward: one must use an anonymous entity whose members are listed, which is less intuitive.

OWL also allows specifying the non-existence of a relationship between two individuals, which is not possible in RDF. For instance, one can indicate that an individual named John does not have a robot assistant and cannot perform certain tasks alone.

In OWL 2, facts about individuals provide detailed information about specific entities. It is thus possible to specify the classes to which an individual belongs as well as the values of its properties, which can be either other individuals (via ObjectProperty) or literals (via DataProperty), depending on the type of property.

4.16.1. Property Chains in OWL and SROIQ

A property chain (or role chain) allows the composition of multiple relationships to automatically infer a new relationship. Formally, these relationships are defined in the **RBox** (the role box) of SROIQ logic, which forms the formal basis of OWL 2 DL.

The RBox contains all the properties (roles) of the ontology, their hierarchies, their characteristics (transitive, symmetric, functional, etc.), and property chains.

Thus, a property chain is not a simple link in the ABox (individuals) but a rule for composing roles that applies to all individuals according to the SROIQ model.

Semantics of the logic $\mathcal{SROIQ}$

Let $\mathbf{C}$ be a set of concept names including a subset of nominals $\mathbf{N}$, $\mathbf{R}$ a set of role names including the universal role U,
and $\mathbf{I} = \{a, b, c, \dots\}$ a set of individual names. The set of roles is
$\mathbf{R} \cup \{R^- | \mathbf{R} \in \mathbf{R}\}$, where R^- denotes the inverse of role R.

An interpretation $\mathcal{I} = (\triangle^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a set $\triangle^{\mathcal{I}}$ called the domain of $\mathcal{I}$ and a valuation $\cdot^{\mathcal{I}}$ that assigns to each role name R a binary relation $R^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$; for the universal role U, this is the universal relation $\triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$; and to each concept name C a subset $C^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}}$, such that $C^{\mathcal{I}}$ is a singleton if $C \in \mathbf{N}$; finally, each individual name a is interpreted as an element $a^{\mathcal{I}} \in \triangle^{\mathcal{I}}$.

The inverse role is interpreted semantically as usual :
$(R^-)^{\mathcal{I}} = \{\langle y, x\rangle \mid \langle x, y\rangle \in R^{\mathcal{I}}\}$

TBox / ABox / RBox ⇝ OWL2

The knowledge base $\mathcal{SROIQ}(D)$ consists of a triplet $\langle \mathcal{T}, \mathcal{A}, \mathcal{R} \rangle$, where $\mathcal{T}$ denotes the set of class axioms, $\mathcal{A}$ denotes the set of assertions, and $\mathcal{R}$ denotes the set of roles and simple roles. A role is called simple if it is not transitive and does not subsume any transitive roles.

For more details, visit : `https://www.w3.org/TR/owl2-direct-semantics/`

4.16.2. IoT Example

Context:

- Smart home with connected objects:
  - MotionSensor (motion sensor)
  - Camera (camera)
  - AlarmSystem (alarm system)

Properties in the RBox:

1. hasSensor: links a House to a Sensor
2. triggers: links a Sensor to an Event
3. activates: links an Event to an Action

Definition of a property chain in the RBox:

hasSensor o triggers o activates ⊑ causesAlerthasSensor

Interpretation:

- If House M has a Sensor (hasSensor),
- And hasSensor triggers an Event (triggers),
- And triggers activates an Action (activates),
- Then M causes an alert (causesAlert).

OWL automatically infers this relationship through the chain defined in the RBox, enabling efficient reasoning over the individuals in the ABox.

4.16.3. Example : “Property Chains” in OWL 2

In OWL 2, a property chain allows the definition of a new property from a sequence of existing properties.

Concrete example:

- There is a property hasParent linking an individual to their parents.
- In the previous example, a property hasAncestor was created to link a person to all their ancestors, regardless of the length of the hasParent chain.
- Here, we want to be more specific: create a property hasGrandparent that links an individual only to their grandparents, i.e., exactly two hops via hasParent.

Syntaxe fonctionnelle (Functional-Style Syntax)

```
SubObjectPropertyOf(
  ObjectPropertyChain( :hasParent :hasParent )
  :hasGrandparent
)
```

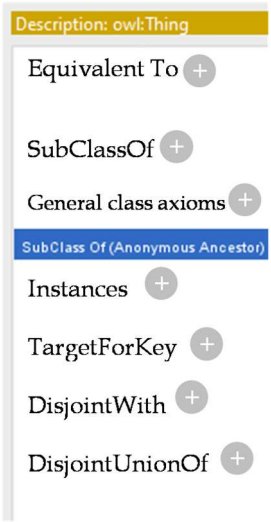
Meaning:

1. ObjectPropertyChain(:hasParent :hasParent): indicates a chain composed of two successive hasParent relationships.
2. SubObjectPropertyOf(... :hasGrandparent): any chain of two hasParent relationships becomes an instance of the hasGrandparent property.

In other words:
If A has parent B and B has parent C, then OWL automatically infers that A has grandparent C via hasGrandparent.

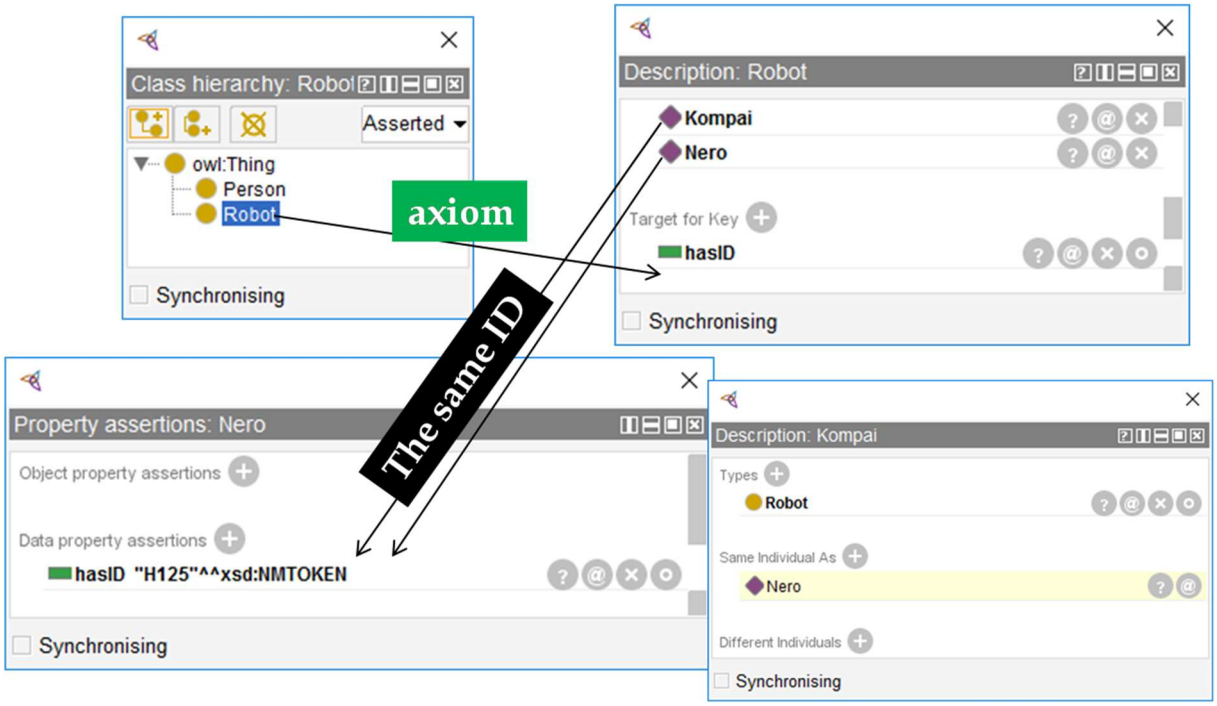
4.16.4. Axiomes TBox

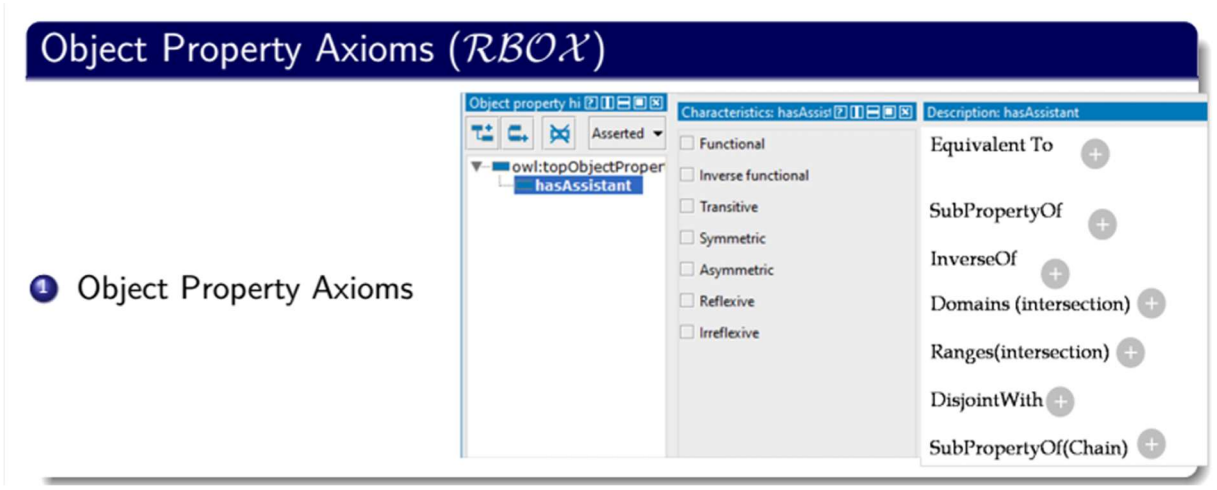
1. A class is defined using the vocabulary element owl:Class
2. rdfs:Class is not used (the semantic interpretation of classes in OWL is more restrictive than in RDF)
3. The use of rdfs:subClassOf is retained to define a class hierarchy



4.16.5. Example of the hasKey Property

Example created with Protégé (http://protege.stanford.edu/) . The reasoning engine infers via the ***hasKey*** property that the individual Kompai and the individual Nero reference the same Robot individual.





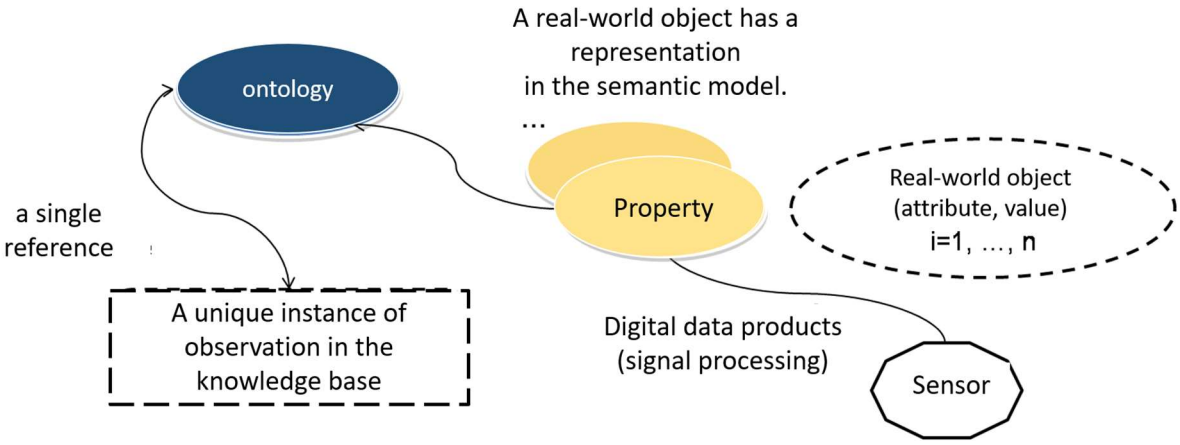
4.16.6. Open World vs. Closed World Assumptions

1. Under the Closed World Assumption (CWA), a statement is considered true if its negation cannot be proven.
2. Under the Open World Assumption (OWA), which underlies OWL reasoning (logical interpretation), the absence of a fact in the knowledge base implies that the fact is unknown rather than false.

3. Both assumptions are useful for reasoning in different applications.
4. OWL 2 introduces Negative Property Assertions, which allow explicitly stating that a fact does not exist. This can be applied to both Datatype and ObjectProperty types, partially mitigating the limitations of the OWA.

4.16.7. Unique Name Assumption

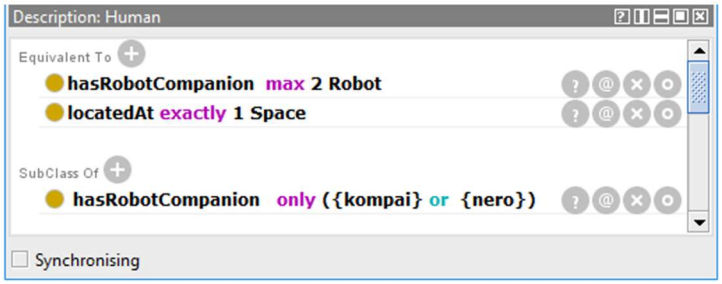
- Unique Name Assumption (UNA) is the hypothesis that different names refer to different individuals.
- Formally, if two individuals aaa and bbb satisfy a≠ba \neq ba□=b, then their interpretations aI≠bIa^{\mathcal{I}} \neq b^{\mathcal{I}}al□=bI.
- In a database, individuals with different names are indeed distinct. In OWL, if two individuals have different names, it is possible to infer that they could actually be the same individual unless UNA is explicitly applied.
- The adapted figure from Sabri et al., 2013, illustrates the need to apply the UNA principle in IoT (Internet of Things) applications.



Example 1

Example : Reasoning under the UNA Assumption

Considérons la TBox suivante :
Class : Robot
Class : Space
ObjectProperty : hasRobotCompanion & ObjectProperty : locatedAt
Class : Human EquivalentTo :
locatedAt exactly 1 Space $\rightsquigarrow^{DL}$ $located = 1.Space$
hasRobotCompanion max 2 Robot $\rightsquigarrow^{DL}$ $hasRobotCompanion \sqcap \leq 2.Robot$
SubClassOf : hasRobotCompanion only(({*kompai*}) or ({*nero*}))
$\rightsquigarrow^{DL}$ $hasRobotCompanion \sqcap \forall hasRobotCompanion(\{kompai\} \sqcup \{nero\})$
The following figure shows the definition of these axioms in Protégé



Example 2

Example : Reasoning under the UNA Assumption

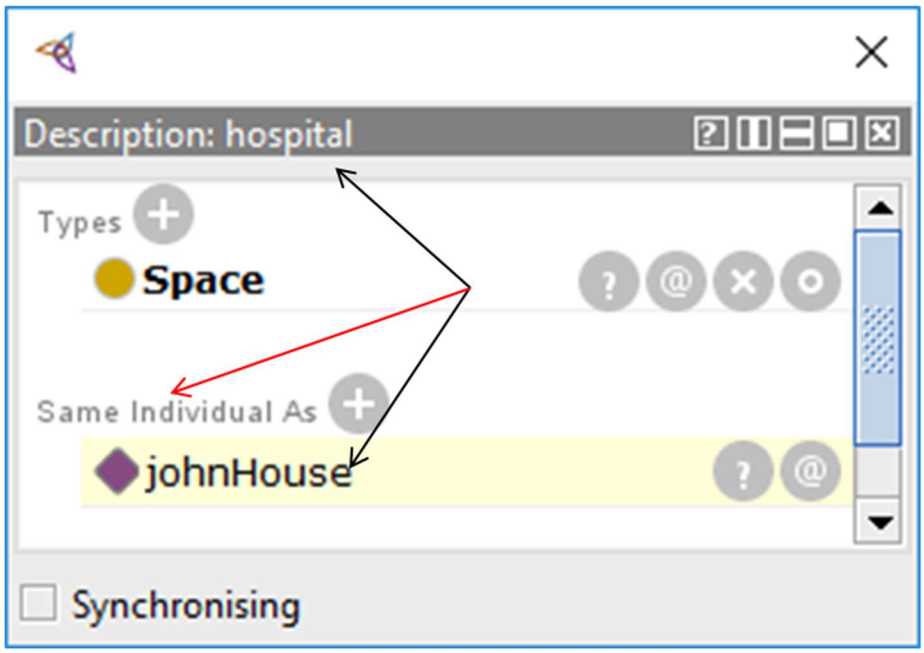
Consider the following TBox :

Individual : **hospital Types : Space** $\leftrightsquigarrow^{DL}$ $Space(hospital)$
Individual : **johnHouse Types : Space** $\leftrightsquigarrow^{DL}$ $Space(johnHouse)$
Individual : david Types : Human $\leftrightsquigarrow^{DL}$ $Human(david)$
Individual : john Types : Human $\leftrightsquigarrow^{DL}$ $Human(john)$

Facts :

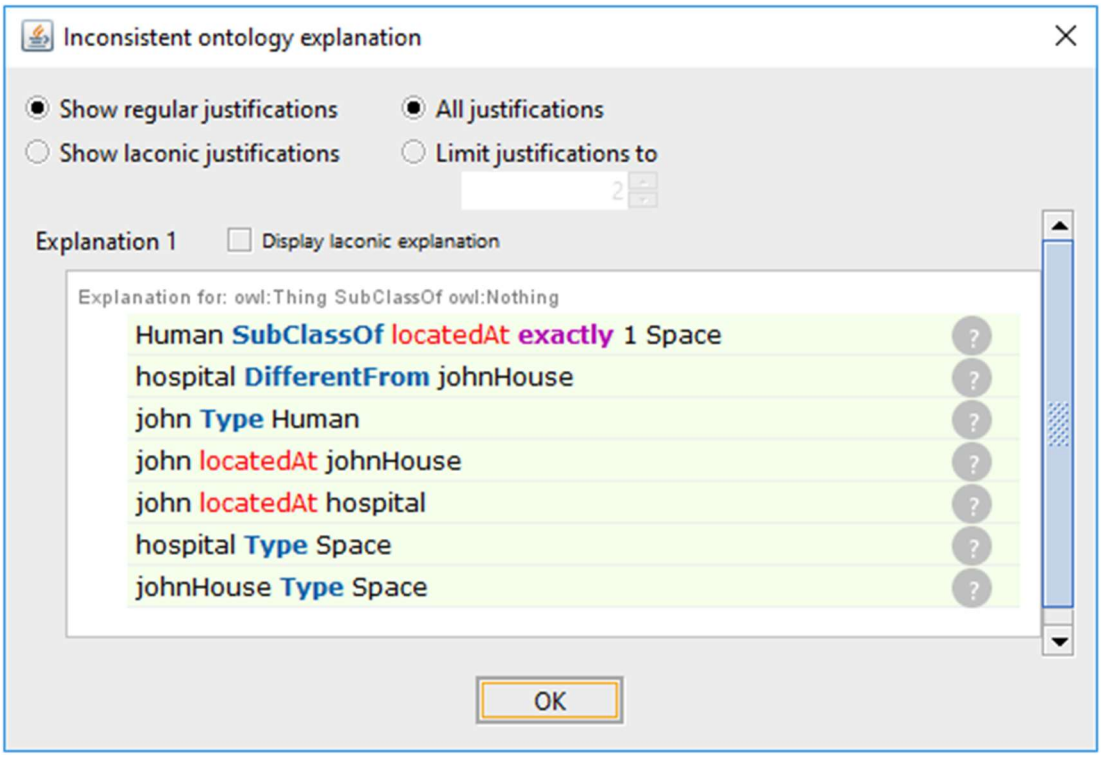
hasRobotCompanion kompai, $\leftrightsquigarrow^{DL}$ $hasRobotCompanion(john, kompai)$
hasRobotCompanion nero, $\leftrightsquigarrow^{DL}$ $hasRobotCompanion(john, nero)$
hasRobotCompanion tutu, $\leftrightsquigarrow^{DL}$ $hasRobotCompanion(john, tutu)$
locatedAt johnHouse $\leftrightsquigarrow^{DL}$ $locatedAt(john, johnHouse)$
locatedAt hospital $\leftrightsquigarrow^{DL}$ $locatedAt(john, hospital)$
Individual : kompai Types : Robot $\leftrightsquigarrow^{DL}$ $Robot(kompai)$
DifferentFrom : nero $\leftrightsquigarrow^{DL}$ $\{kompai\} \sqsubseteq \neg\{nero\}$
Individual : nero Types : Robot $\leftrightsquigarrow^{DL}$ $Robot(nero)$
DifferentFrom : kompai $\leftrightsquigarrow^{DL}$ $\{nero\} \sqsubseteq \neg\{kompai\}$
Individual : tutu Types : $\leftrightsquigarrow^{DL}$ $Robot(tutu)$

The following figure shows that the inference engine concludes that the individuals johnHouse and hospital are the same. Indeed, on one hand, we have the axiom (locatedAt exactly 1 Space) which requires that a Human individual can be located in only one place, and on the other hand, assertions in the ABox indicate that the individual john is at two different locations hospital and johnHouse (The temporal aspect is not considered in this reasoning) which therefore forces the conclusion that the two spaces actually refer to the same place; otherwise, the ontology would be inconsistent.



4.16.8. OWL:SameIndividualAs

If we make explicit that the two individuals (hospital, johnHouse) are different, formally hospital ⊑ ¬ johnHouse. Consequently, an inconsistency will be raised; the figure illustrates the explanation provided by the Pellet reasoner.
Note that we could also have used the axiom: ⊓≤1 located.Space.



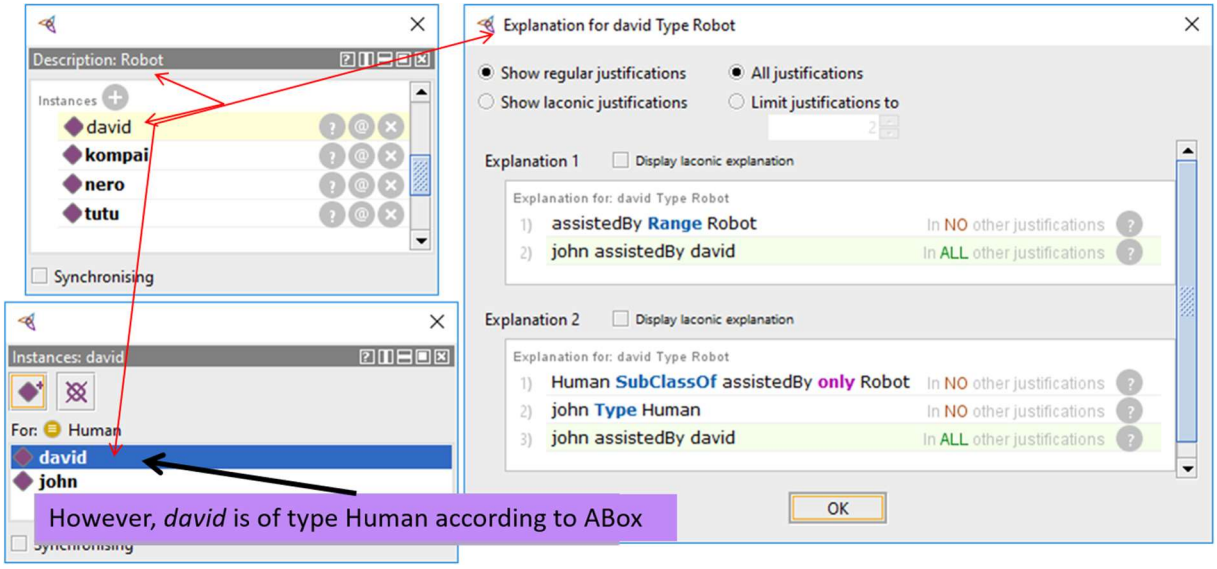
Querying the location of the individual *david* will not yield an answer. Indeed, under the open-world assumption, the fact that the property *locatedAt* does not link any space to the individual *david* does not cause any problem in OWL, and the answer would be unknown. It is partly logical and reasonable not to be able to provide a precise location for *david* (since it is unknown).

4.16.9. Reasoning based on Domain & Range

1. Human ⊑ ∀ assistedBy.Robot: Expresses that a human can only be assisted by a Robot.
2. ⊤ ⊑ ∀ assistedBy.Robot: Expresses that the range of the property *assistedBy* is Robot. That is, any property in the universe (domain) using *assistedBy* must have as its value an individual of type Robot.
3. assistedBy(john, david): Expresses that *john* is assisted by *david*.

From (1) to (3), we can infer that *david* is a robot (the next slide explains this implication). In a closed-world scenario, this reasoning is rejected due to constraint violation.

4. Human ⊓ Robot ⊑ ⊥: Expresses that the classes *Robot* and *Human* are disjoint. This prevents a human from assisting another human. Consequently, this axiom raises an inconsistency if assertion (3) is added to the knowledge base



In ambient intelligence systems

The environment is dynamic, and consequently, first-order logic formalisms and ontologies cannot represent facts whose truth value is not known in advance or may continuously change. Consider the case where *john*, of type Human, must be assisted by at least one robot;
formally in DL we write: $\textbf{\textit{Human}} \sqsubseteq \exists \textbf{\textit{assistedBy.Robot}}$.

If no value links the property *assistedBy* to *john*, then trying to determine the number of robots assisting *john* will not be possible within ontology-based reasoning. Such reasoning may in some cases lead to erroneous or contradictory deductions. In contrast, in systems based on the closed-world assumption, the answer would unambiguously be 0.

4.17. Reasoning Engines

OWL 2 DL: Pellet. https://github.com/stardog-union/pellet

FaCT++: https://bitbucket.org/dtsarkov/factplusplus

HermiT: https://github.com/phillord/hermit-reasoner

OWL 2 EL: https://github.com/liveontologies/elk-reasoner

OWL 2 QL: https://github.com/ontop/ontop/

OWL 2 RL: RDFox. http://www.cs.ox.ac.uk/isg/tools/RDFox/

5. Bibliography

1. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2002.
2. Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes*. W3C Recommendation, 2001. Available at: http://www.w3.org/TR/2002/WD-xmlschema-2-20010502/.
3. Sabri, Lyazid. *Modèles sémantiques et raisonnements réactif et narratif, pour la gestion du contexte en intelligence ambiante et en robotique ubiquitaire* (Thèse de doctorat, Université Paris-Est Créteil Val-de-Marne). (2013). Disponible sur HAL: https://theses.hal.science/tel-01332358
4. Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. *DAML+OIL (March 2001) reference*

description. W3C Note, 18 December 2001. Available at: http://www.w3.org/TR/2001/NOTE-daml+oil-reference-20011218.

5. I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. van Harmelen, M. Klein, S. Staab, R. Studer, and E. Motta. *OIL: The Ontology Inference Layer*. Technical Report IR-479, Vrije Universiteit Amsterdam, Faculty of Sciences, September 2000.
6. Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. *OWL web ontology language semantics and abstract syntax*. W3C Working Draft, 31 March 2003. Available at: http://www.w3.org/TR/2003/WD-owl-semantics-20030331/.
7. Frank van Harmelen, Ian Horrocks, and Peter F. Patel-Schneider. *A model-theoretic semantics for DAML+OIL (March 2001)*. W3C Note, 18 December 2001. Available at: http://www.w3.org/TR/2001/NOTE-daml+oil-model-20011218.
8. D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P.F. Patel-Schneider. *OIL: An ontology infrastructure for the semantic web*. IEEE Intelligent Systems, 16(2):38–45, 2001.
9. I. Horrocks and P.F. Patel-Schneider. *The generation of DAML+OIL*. In Proc. of the 2001 Description Logic Workshop (DL 2001), volume 49 of CEUR Electronic Workshop Proceedings, 2001.
10. I. Horrocks, P.F. Patel-Schneider, and F. van Harmelen. *Reviewing the design of DAML+OIL: An ontology language for the semantic web*. In Proc. of the 18th Nat. Conf. on Artificial Intelligence (AAAI 2002), pages 792–797. AAAI Press, 2002.
11. I. Horrocks, P.F. Patel-Schneider, and F. van Harmelen. *From SHIQ and RDF to OWL: The making of a web ontology language*. J. of Web Semantics, 1(1):7–26, 2003.
12. J. Doyle and R.S. Patil. *Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services*. Artificial Intelligence, 48:261–297, 1991.
13. P.F. Patel-Schneider, P. Hayes, and I. Horrocks. *OWL Web Ontology Language semantics and abstract syntax*. W3C Recommendation, 10 February 2004. Available at: http://www.w3.org/TR/owl-semantics/.
14. I. Horrocks, O. Kutz, U. Sattler. *The Even More Irresistible SROIQ*, in Proc. KR 2006, Lake District, UK, 2006.
15. Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of Knowledge Representation. Foundations of Artificial Intelligence*. Elsevier, 2008.
16. J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, 1984.
17. M. Ross Quillian. *Semantic memory*. In Marvin Minsky, editor, *Semantic Information Processing*, chapter 4, pages 227–270. MIT Press, 1968.
18. Peter F. Patel-Schneider and Boris Motik, editors. *OWL 2 Web Ontology Language: Mapping to RDF Graphs*. W3C Recommendation, 2009. Available at: http://www.w3.org/TR/owl2-mapping-to-rdf/.
19. W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 2009. Available at: http://www.w3.org/TR/owl2-overview/.

20. Marvin Minsky. *A framework for representing knowledge*. Artificial Intelligence Memo, A.I. Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.
21. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, second edition, 2007.
22. D. Davidson. *Causal Relations*. In *The Journal of Philosophy*, volume 64, pages 691–703, 1967.
23. Hebeler, J., Fisher, M., Blace, R., & Perez-Lopez, A. (2009). *Semantic Web Programming*. Foreword by Mike Dean, Principal Engineer, BBN Technologies. Wiley.

6. External Links

1. Berners-Lee, T. (1998, September). *Semantic Web Road Map*. Disponible sur : http://www.w3.org/DesignIssues/Semantic.html
2. Berners-Lee, T. (1998, March). *Evolvability*. Disponible sur : http://www.w3.org/DesignIssues/Evolution.html
3. Berners-Lee, T. (1998, September). *What the Semantic Web Can Represent*. Disponible sur : http://www.w3.org/DesignIssues/RDFnot.html
4. Dumbill, E. (2000, November 1). *The Semantic Web: A Primer*. Disponible sur : http://www.xml.com/pub/a/2000/11/01/semanticweb/
5. van Harmelen, F., & Fensel, D. (1999). *Practical Knowledge Representation for the Web*. Disponible sur https://www.cs.vu.nl/~frankh/postscript/IJCAI99-III.html
6. Hendler, J. (2001, March–April). *Agents and the Semantic Web*. *IEEE Intelligent Systems, 16*, 30–37. Preprint disponible sur : http://www.cs.umd.edu/users/hendler/AgentWeb.html
7. Palmer, S. (2001, June). *The SemanticWeb, Taking Form*. Disponible sur : http://infomesh.net/2001/06/swform/
8. Palmer, S. (2001). *The SemanticWeb: An Introduction*. Disponible sur : http://infomesh.net/2001/swintro/
24. Swartz, A. (n.d.). *The SemanticWeb in Breadth*. Disponible sur : http://logicerror.com/semanticWeb-long