Outline

Outline

Outline

Elements of an ontology language

- Syntax
  - Alphabet
  - Languages constructs
  - Sentences to assert knowledge
- Semantics
  - Formal meaning
- Pragmatics
  - Intended meaning
  - Usage

Static vs. dynamic aspects

The aspects of the domain of interest that can be modeled by an ontology language can be classified into:

- Static aspects
  - Are related to the structuring of the domain of interest.
  - Supported by virtually all languages.
- Dynamic aspects
  - Are related to how the elements of the domain of interest evolve over time.
  - Supported only by some languages, and only partially (cf. services).

Before delving into the dynamic aspects, we need a good understanding of the static ones.
In this course we concentrate essentially on the static aspects.

Outline

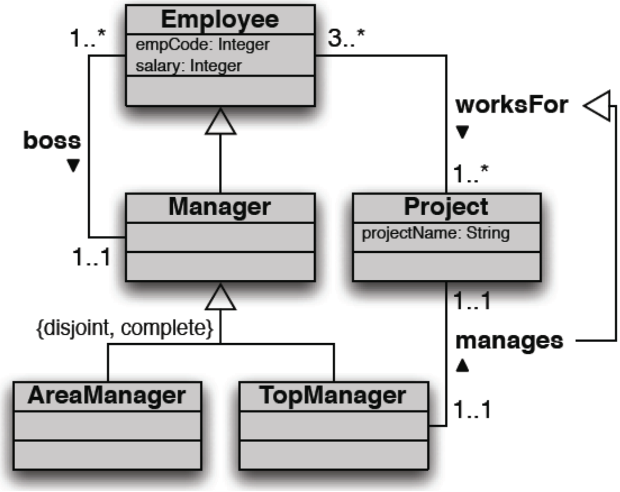
Intensional level of an ontology language

An ontology language for expressing the intensional level usually includes:

- Concepts
- Properties of concepts
- Relationships between concepts, and their properties
- Axioms
- Queries

Ontologies are typically rendered as diagrams (e.g., Semantic Networks, Entity-Relationship schemas, UML Class Diagrams).

Example: ontology rendered as UML Class Diagram



Concepts

Definition (Concept)

A concept is an element of an ontology that denotes a collection of instances (e.g., the set of "employees").

We distinguish between:

- Intensional definition:
  - specification of name, properties, relations,...
- Extensional definition:
  - specification of the instances

Concepts are also called classes, entity types, frames.

Definition (Property)

A property is an element of an ontology that qualifies another element (e.g., a concept or a relationship).

Property definition (intensional and extensional):

- Name
- Type: may be either
  - atomic (integer, real, string, enumerated,...), or
    e.g., eye-color $\rightarrow$ { blu, brown, green, grey }
  - structured (date, set, list,...)
    e.g., date $\rightarrow$ day/month/year
- The definition may also specify a default value.

Properties are also called attributes, features, slots, data properties

Definition (Relationship)

A relationship is an element of an ontology that expresses an association among concepts.

We distinguish between:

- Intensional definition:
  specification of involved concepts
  e.g., worksFor is defined on Employee and Project
- Extensional definition:
  specification of the instances of the relationship, called facts
  e.g., worksFor(domenico, tones)

Relationships are also called associations, relationship types, roles, object properties.

Axioms

Definition (Axiom)

An axiom is a logical formula that expresses at the intensional level a condition that must be satisfied by the elements at the extensional level.

Different kinds of axioms/conditions:

- subclass relationships, e.g., $\mathsf{Manager} \sqsubseteq \mathsf{Employee}$
- equivalences, e.g., $\mathsf{Manager} \equiv \mathsf{AreaManager} \sqcup \mathsf{TopManager}$
- disjointness, e.g., $\mathsf{AreaManager} \sqcap \mathsf{TopManager} \equiv \perp$
- (cardinality) restrictions, e.g., each Employee worksFor at least 3 Project
- ...

Axioms are also called assertions.
A special kind of axioms are definitions

Extensional level of an ontology language

At the extensional level we have individuals and facts:

- An instance represents an individual (or object) in the extension of a concept.
  e.g., domenico is an instance of Employee
- A fact represents a relationship holding between instances.
  e.g., worksFor(domenico, tones)

Outline

Comparison with other formalisms

- Ontology languages vs. knowledge representation languages:
  - Ontologies are knowledge representation schemas.
- Ontology vs. logic:
  - Logic is the tool for assigning semantics to ontology languages.
- Ontology languages vs. conceptual data models:
  - Conceptual schemas are special ontologies, suited for conceptualizing a single logical model (database).
- Ontology languages vs. programming languages:
  - Class definitions are special ontologies, suited for conceptualizing a single structure for computation.

Classification of ontology languages

- Graph-based
  - Semantic networks
  - Conceptual graphs
  - UML class diagrams, Entity-Relationship diagrams
- Frame based
  - Frame Systems
  - OKBC, XOL
- Logic based
  - Description Logics (e.g., $\mathcal{SHOIQ}$, $\mathcal{DLR}$, DL-Lite , OWL,...)
  - Rules (e.g., RuleML, LP/Prolog, F-Logic)
  - First Order Logic (e.g., KIF)
  - Non-classical logics (e.g., non-monotonic, probabilistic)

Outline

Outline

Modeling the domain of interest

We aim at obtaining a description of the data of interest in semantic terms.
One can proceed as follows:

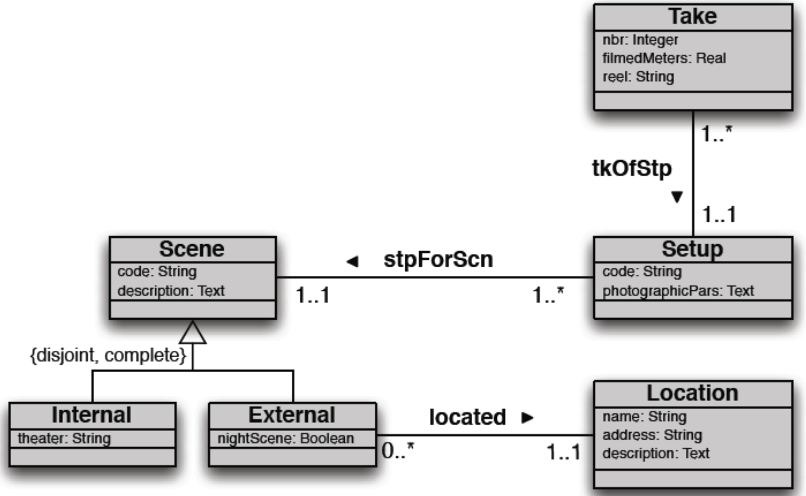
1. Represent the domain of interest as a conceptual schema, similar to those used at design time to design a database.
2. Formalize the conceptual schema as a logical theory, namely the ontology.
3. Use the resulting logical theory for reasoning and query answering.

Let's start with an exercise

Exercise

Requirements: We are interested in building a software application to manage filmed scenes for realizing a movie, by following the so-called "Hollywood Approach".
Every scene is identified by a code (a string) and is described by a text in natural language.
Every scene is filmed from different positions (at least one), each of this is called a setup. Every setup is characterized by a code (a string) and a text in natural language where the photographic parameters are noted (e.g., aperture, exposure, focal length, filters, etc.). Note that a setup is related to a single scene.
For every setup, several takes may be filmed (at least one). Every take is characterized by a (positive) natural number, a real number representing the number of meters of film that have been used for shooting the take, and the code (a string) of the reel where the film is stored. Note that a take is associated to a single setup.
Scenes are divided into internals that are filmed in a theater, and externals that are filmed in a location and can either be "day scene" or "night scene". Locations are characterized by a code (a string) and the address of the location, and a text describing them in natural language.
Write a precise specification of this domain using any formalism you like!

Solution 1: Use conceptual modeling diagrams (UML)!



Solution 1: Use conceptual modeling diagrams - Discussion

Good points:

- Easy to generate (it's the standard in software design).
- Easy to understand for humans.
- Well disciplined, well-established methodologies available.

Bad points:

- No precise semantics (people that use it wave hands about it).
- Verification (or better validation) done informally by humans.
- Machine incomprehensible (because of lack of formal semantics).
- Automated reasoning and query answering out of question.
- Limited expressiveness.[a]

[a] Not really a bad point, in fact.

Use logic!!!

Alphabet

$Scene(x)$, $Setup(x)$, $Take(x)$, $Internal(x)$, $External(x)$, $Location(x)$, $stpForScn(x, y)$, $tkOfStp(x, y)$, $located(x, y)$,...

Axioms

$\forall x, y.code_{Scene}(x, y) \rightarrow Scene(x) \wedge String(y)$
$\forall x, y.description(x, y) \rightarrow Scene(x) \wedge Text(y)$
$\forall x, y.code_{Setup}(x, y) \rightarrow Setup(x) \wedge String(y)$
$\forall x, y.photographicPars(x, y) \rightarrow Setup(x) \wedge Text(y)$
$\forall x, y.nbr(x, y) \rightarrow Take(x) \wedge Integer(y)$
$\forall x, y.filmedMeters(x, y) \rightarrow Take(x) \wedge Real(y)$
$\forall x, y.reel(x, y) \rightarrow Take(x) \wedge String(y)$
$\forall x, y.theater(x, y) \rightarrow Internal(x) \wedge String(y)$
$\forall x, y.nightScene(x, y) \rightarrow External(x) \wedge Boolean(y)$
$\forall x, y.name(x, y) \rightarrow Location(x) \wedge String(y)$
$\forall x, y.address(x, y) \rightarrow Location(x) \wedge String(y)$
$\forall x, y.description(x, y) \rightarrow Location(x) \wedge Text(y)$
$\forall x.Scene(x) \rightarrow (1 \leq \sharp\{y|code_{Scene}(x, y)\} \leq 1)$
$\forall x.Internal(x) \rightarrow Scene(x)$
$\forall x.External(x) \rightarrow Scene(x)$
$\forall x.Internal(x) \rightarrow \neg External(x)$
$\forall x.Scene(x) \rightarrow Internal(x) \vee External(x)$

$\forall x, y.stpForScn(x, y) \rightarrow$
$\forall Setup(x) \wedge Scene(y)$
$\forall x, y.tkOfStp(x, y) \rightarrow$
$\forall Take(x) \wedge Setup(y)$
$\forall x, y.located(x, y) \rightarrow$
$\forall External(x) \wedge Location(y)$
$\forall x.Setup(x) \rightarrow$
$\forall(1 \leq \sharp\{y|stpForScn(x, y)\} \leq 1)$
$\forall y.Scene(y) \rightarrow$
$\forall(1 \leq \sharp\{x|stpForScn(x, y))\}$
$\forall x.Take(x) \rightarrow$
$\forall(1 \leq \sharp\{y|tkOfStp(x, y)\} \leq 1)$
$\forall x.Setup(y) \rightarrow$
$\forall(1 \leq \sharp\{x|stpForScn(x, y)\})$
$\forall x.External(x) \rightarrow$
$\forall(1 \leq \sharp\{y|located(x, y)\} \leq 1)$
$\ldots$

Solution 2: Use logic - Discussion

Good points:

- Precise semantics.
- Formal verification.
- Allows for query answering.
- Machine comprehensible.
- Virtually unlimited expressiveness [a].

[a] Not necessarily a good point, in fact.

Bad points:

- Difficult to generate.
- Difficult to understand for humans.
- Too unstructured (making reasoning difficult), no well-established methodologies available.
- Automated reasoning may be impossible.

Solution 3: Use both!!!

Note these two approaches seem to be orthogonal, but in fact they can be used together cooperatively!!!

Basic idea

- Assign formal semantics to constructs of the conceptual design diagrams.
- Use conceptual design diagrams as usual, taking advantage of methodologies developed for them in Software Engineering.
- Read diagrams as logical theories when needed, i.e., for formal understanding, verification, automated reasoning, etc.

Added value

- Inherited from conceptual modeling diagrams: ease-to-use for humans
- inherit from logic: formal semantics and reasoning tasks, which are needed for formal verification and machine manipulation.

Solution 3: Use both!!! (cont'd)

Important

The logical theories that are obtained from conceptual modeling diagrams are of a specific form.

- Their expressiveness is limited (or better, well-disciplined).
- One can exploit the particular form of the logical theory to simplify reasoning.
- The aim is getting:
  - decidability, and
  - reasoning procedures that match the intrinsic computational complexity of reasoning over the conceptual modeling diagrams.

Conceptual models vs. logic

We illustrate now what we get from interpreting conceptual modeling diagrams in logic.
We will use:

- as conceptual modeling diagrams: UML Class Diagrams. Note: we could equivalently use Entity-Relationship Diagrams instead of UML.
- as logic: First-Order Logic to formally capture semantics and reasoning.

Outline

The Unified Modeling Language (UML)

The Unified Modeling Language (UML) was developed in 1994 by unifying and integrating the most prominent object-oriented modeling approaches:

- Booch
- Rumbaugh: Object Modeling Technique (OMT)
- Jacobson: Object-Oriented Software Engineering (OOSE)

History:

- 1995, version 0.8, Booch, Rumbaugh; 1996, version 0.9, Booch, Rumbaugh, Jacobson; version 1.0 BRJ + Digital, IBM, HP,...
- UML 1.4.2 is industrial standard ISO/IEC 19501.
- Current version: 2.3 (May 2010): `http://www.omg.org/spec/UML/`
- 1999-today: de facto standard object-oriented modeling language.

References:

- Grady Booch, James Rumbaugh, Ivar Jacobson, "The unified modeling language user guide", Addison Wesley, 1999 (2nd ed., 2005)
- `http://www.omg.org/` → UML
- `http://www.uml.org/`

In this course we deal only with one of the most prominent components of UML: UML Class Diagrams.

A UML Class Diagram is used to represent explicitly the information on a domain of interest (typically the application domain of software).

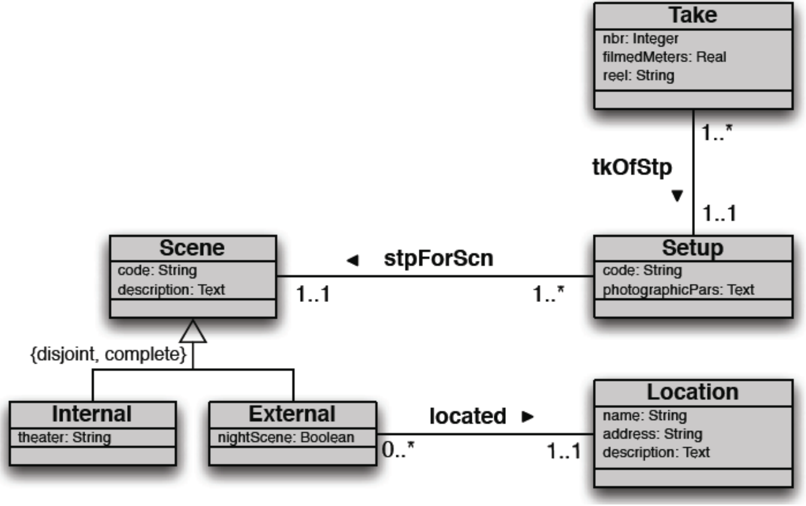
Note: This is exactly the goal of all conceptual modeling formalism, such as Entity-Relationship Diagrams (standard in Database design) or Ontologies.

UML Class Diagrams (cont'd)

The UML class diagram models the domain of interest in terms of:

- objects grouped into classes;
- associations, representing relationships between classes;
- attributes, representing simple properties of the instances of classes;
  Note: here we do not deal with "operations".
- sub-classing, i.e., ISA and generalization relationships.

Example of a UML Class Diagram



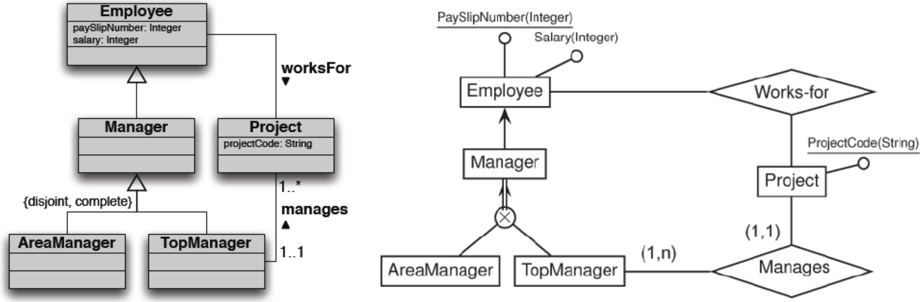
Use of UML Class Diagrams

UML Class Diagrams are used in various phases of a software design:

1. During the so-called analysis, where an abstract precise view of the domain of interest needs to be developed.
   $\rightsquigarrow$ the so-called "conceptual perspective".
2. During software development, to maintain an abstract view of the software to be developed.
   $\rightsquigarrow$ the so-called "implementation perspective".

In this course we focus on 1!

UML class diagrams (when used for the conceptual perspective) closely resemble Entity-Relationship (ER) Diagrams.
Example of UML vs. ER:



Classes in UML

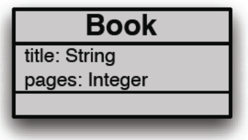
Definition (Class)

A class in UML models a set of objects (its "instances") that share certain common properties, such as attributes, operations, etc.

Each class is characterized by:

- a name (which must be unique in the whole class diagram),
- a set of (local) properties, namely attributes and operations (see later).

Example



- the name of the class is 'Book'
- the class has two properties (attributes)

Definition (Instances)

The objects that belong to a class are called instances of the class. They form a so-called instantiation (or extension) of the class.

Example

Here are some possible instantiations of our class Book:

$$\{book_a, book_b, book_c, book_d, book_e\}$$
$$\{book_\alpha, book_\beta\}$$
$$\{book_1, book_2, book_3, \dots, book_{500}, \dots\}$$

Which is the actual instantiation?
We will know it only at run-time!!! - We are now at design time!

A class represents a set of objects. ... But which set? We don't actually know.
So, how can we assign a semantics to such a class?

Definition (Class representation)

We represent a class as a FOL unary predicate!

Example

For our class Book, we introduce a predicate $Book(x)$.

Definition (Association)

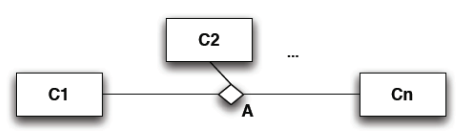
An association in UML models a relationship between two or more classes.

- At the instance level, an association is a relation between the instances of two or more classes.
- Associations model properties of classes that are non-local, in the sense that they involve other classes.
- An association between n classes is a property of each of these classes.

Example

Associations: formalization

Definition (Association representation)



We can represent an n-ary association A among classes $C_1, \ldots, C_n$ as an n-ary predicate A in FOL.
We assert that the components of the predicate must belong to the classes participating to the association:

$$\forall x_1, \ldots, x_n . A(x_1, \ldots, x_n) \rightarrow C_1(x_1) \land \cdots \land C_n(x_n)$$

Example

$$\forall x_1, x_2 . writtenBy(x_1, x_2) \rightarrow Book(x_1) \land Author(x_2)$$

Associations: multiplicity

Definition (Multiplicity Constraints)

On binary associations, we can place multiplicity constraints, i.e., a minimal and maximal number of tuples in which every object participates as first (second) component.

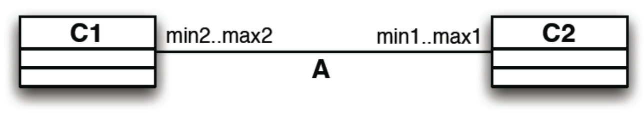
Example

Note: UML multiplicities for associations are look-across and are not easy to use in an intuitive way for n-ary associations. So typically they are not used at all.

In contrast, in ER Schemas, multiplicities are not look-across and are easy to use, and widely used.

Associations: formalization of multiplicities

Definition (Multiplicity constraint representation)



Multiplicities of binary associations are easily expressible in FOL:

$$\forall x_1 \cdot C_1(x_1) \rightarrow (min_1 \leq \sharp\{x_2 | A(x_1, x_2)\} \leq max_1)$$

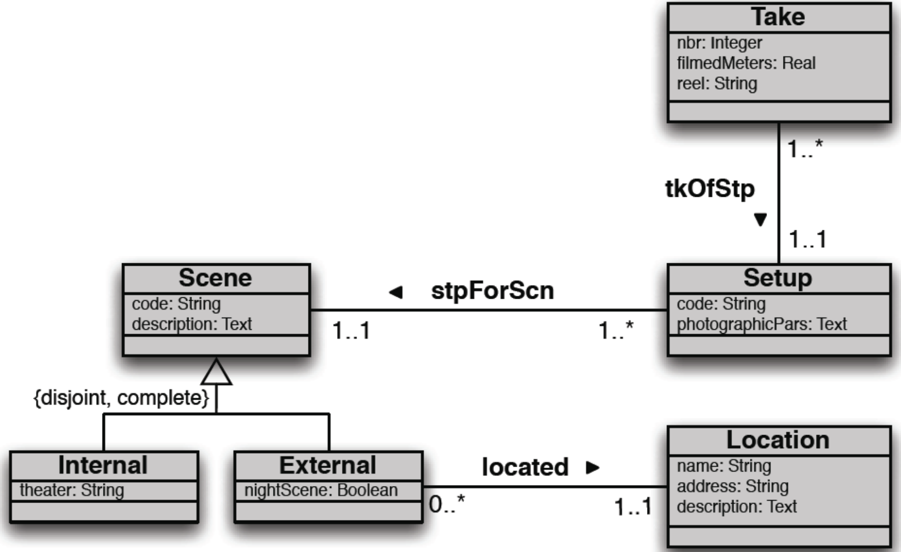
$$\forall x_2 \cdot C_2(x_2) \rightarrow (min_2 \leq \sharp\{x_1 | A(x_1, x_2)\} \leq max_2)$$

Example

$$\forall x \cdot Book(x) \rightarrow (1 \leq \sharp\{y | written_{by}(x, y)\})$$

Note: this is a shorthand for a FOL formula expressing the cardinality of the set of possible values for y.

In our example...



In our example...

Alphabet

$Scene(x)$, $Setup(x)$, $Take(x)$, $Internal(x)$, $External(x)$, $Location(x)$, $stpForScn(x, y)$, $tkOfStp(x, y)$, $located(x, y)$,...

Axioms

$\forall x, y.code_{Scene}(x, y) \rightarrow Scene(x) \wedge String(y)$
$\forall x, y.description(x, y) \rightarrow Scene(x) \wedge Text(y)$
$\forall x, y.code_{Setup}(x, y) \rightarrow Setup(x) \wedge String(y)$
$\forall x, y.photographicPars(x, y) \rightarrow Setup(x) \wedge Text(y)$
$\forall x, y.nbr(x, y) \rightarrow Take(x) \wedge Integer(y)$
$\forall x, y.filmedMeters(x, y) \rightarrow Take(x) \wedge Real(y)$
$\forall x, y.reel(x, y) \rightarrow Take(x) \wedge String(y)$
$\forall x, y.theater(x, y) \rightarrow Internal(x) \wedge String(y)$
$\forall x, y.nightScene(x, y) \rightarrow External(x) \wedge Boolean(y)$
$\forall x, y.name(x, y) \rightarrow Location(x) \wedge String(y)$
$\forall x, y.address(x, y) \rightarrow Location(x) \wedge String(y)$
$\forall x, y.description(x, y) \rightarrow Location(x) \wedge Text(y)$
$\forall x.Scene(x) \rightarrow (1 \leq \sharp\{y | code_{Scene}(x, y)\} \leq 1)$
$\forall x.Internal(x) \rightarrow Scene(x)$
$\forall x.External(x) \rightarrow Scene(x)$
$\forall x.Internal(x) \rightarrow \neg External(x)$
$\forall x.Scene(x) \rightarrow Internal(x) \vee External(x)$

$\forall x, y.stpForScn(x, y) \rightarrow$
$\forall Setup(x) \wedge Scene(y)$
$\forall x, y.tkOfStp(x, y) \rightarrow$
$\forall Take(x) \wedge Setup(y)$
$\forall x, y.located(x, y) \rightarrow$
$\forall External(x) \wedge Location(y)$
$\forall x.Setup(x) \rightarrow$
$\forall (1 \leq \sharp\{y | stpForScn(x, y)\} \leq 1)$
$\forall y.Scene(y) \rightarrow$
$\forall (1 \leq \sharp\{x | stpForScn(x, y))\}$
$\forall x.Take(x) \rightarrow$
$\forall (1 \leq \sharp\{y | tkOfStp(x, y)\} \leq 1)$
$\forall x.Setup(y) \rightarrow$
$\forall (1 \leq \sharp\{x | stpForScn(x, y)\})$
$\forall x.External(x) \rightarrow$
$\forall (1 \leq \sharp\{y | located(x, y)\} \leq 1)$
$\ldots$

Associations: most interesting multiplicities

The most interesting multiplicities are:

- $0..*$: unconstrained
- $1..*$: mandatory participation
- $0..1$: functional participation (the association is a partial function)
- $1..1$: mandatory and functional participation (the association is a total faction)

Definition (In FOL)

- $0..*$: no constraint
- $1..*$: $\forall x \cdot C_1(x) \rightarrow \exists y \cdot A(x.y)$
- $0..1$: $\forall x \cdot C_1(x) \rightarrow \forall y. \cdot y' \cdot A(x,y) \wedge A(x,y') \rightarrow y = y'$
  (or simply $\forall x, y, y'. \cdot A(x,y) \wedge A(x,y') \rightarrow y = y'$)
- $1..1$: $(\forall x \cdot C_1(x) \rightarrow \exists y. \cdot A(x,y)) \wedge (\forall x, y, y' \cdot A(x,y) \wedge A(x,y') \rightarrow y = y')$

Attributes

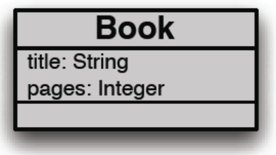
Definition (Attribute)

An attribute models a local property of a class.

It is characterized by:

- a name (which is unique only in the class it belongs to),
- a type (a collection of possible values),
- and possibly a multiplicity.

Example



-The name of one of the attributes is 'title'.
-Its type is 'String'.

Attributes as functions

Attributes (without explicit multiplicity) are:

- mandatory (must have at least a value), and
- single-valued (can have at most one value).

That is, they are total functions from the instances of the class to the values of the type they have.

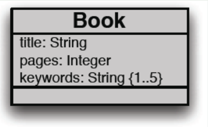
Example

$book_3$ has as value for the attribute 'title' the String: "The little digital video book".

Attributes with multiplicity

More generally attributes may have an explicit multiplicity (similar to that of associations).

Example



-The attribute 'title' has an implicit multiplicity of 1..1.
-The attribute 'keywords' has an explicit multiplicity of 1..5.

Note: When the multiplicity is not specified, then it is assumed to be 1..1

Attributes: formalization

Since attributes may have a multiplicity different from 1..1, they are better formalized as binary predicates, with suitable assertions representing types and multiplicity.

Definition (Attribute representation)

Given an attribute att of a class C with type T and multiplicity $i..j$, we capture it in FOL as a binary predicate $att_C(x, y)$ with the following assertions:

- An assertion for the attribute type:

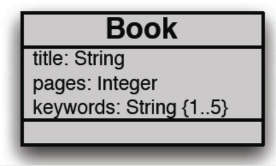
$$\forall x, y \cdot att_C(x, y) \rightarrow C(x) \wedge T(y)$$

- An assertion for the multiplicity:

$$\forall x \cdot C(x) \rightarrow (i \leq \sharp\{y | att_C(x, y)\} \leq j)$$

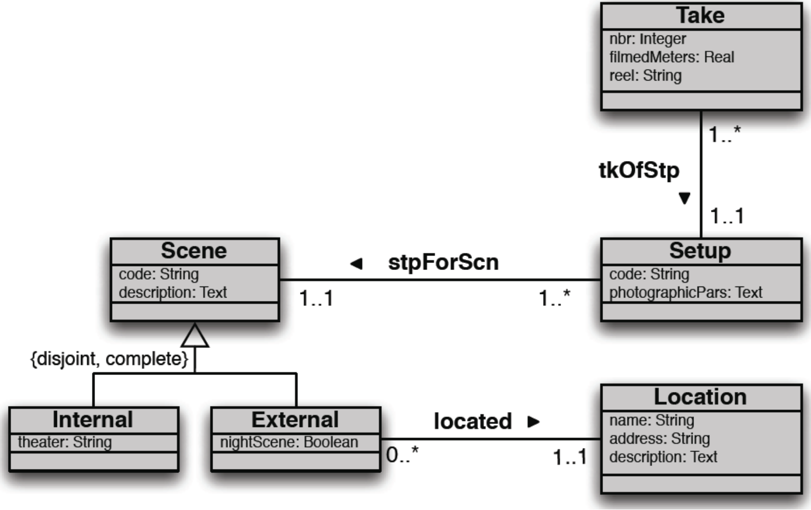
Attributes: example of formalization

Example



$$\forall x, y \cdot title_B(x, y) \rightarrow Book(x) \wedge String(y)$$
$$\forall Book(x) \rightarrow (1 \leq \sharp\{y | title_B(x, y)\} \leq 1)$$
$$\forall x, y \cdot pages_B(x, y) \rightarrow Book(x) \wedge Integer(y)$$
$$\forall Book(x) \rightarrow (1 \leq \sharp\{y | pages_B(x, y)\} \leq 1)$$
$$\forall x, y \cdot keywords_B(x, y) \rightarrow Book(x) \wedge String(y)$$
$$\forall Book(x) \rightarrow (1 \leq \sharp\{y | keywords_B(x, y)\} \leq 5)$$

In our example...



In our Example

Alphabet

$Scene(x)$, $Setup(x)$, $Take(x)$, $Internal(x)$, $External(x)$, $Location(x)$, $stpForScn(x,y)$, $tkOfStp(x,y)$, $located(x,y)$,...

Axioms

$\forall x,y.code_{Scene}(x,y) \to Scene(x) \wedge String(y)$
$\forall x,y.description(x,y) \to Scene(x) \wedge Text(y)$
$\forall x,y.code_{Setup}(x,y) \to Setup(x) \wedge String(y)$
$\forall x,y.photographicPars(x,y) \to Setup(x) \wedge Text(y)$
$\forall x,y.nbr(x,y) \to Take(x) \wedge Integer(y)$
$\forall x,y.filmedMeters(x,y) \to Take(x) \wedge Real(y)$
$\forall x,y.reel(x,y) \to Take(x) \wedge String(y)$
$\forall x,y.theater(x,y) \to Internal(x) \wedge String(y)$
$\forall x,y.nightScene(x,y) \to External(x) \wedge Boolean(y)$
$\forall x,y.name(x,y) \to Location(x) \wedge String(y)$
$\forall x,y.address(x,y) \to Location(x) \wedge String(y)$
$\forall x,y.description(x,y) \to Location(x) \wedge Text(y)$
$\forall x.Scene(x) \to (1 \leq \sharp\{y|code_{Scene}(x,y)\} \leq 1)$
$\forall x.Internal(x) \to Scene(x)$
$\forall x.External(x) \to Scene(x)$
$\forall x.Internal(x) \to \neg External(x)$
$\forall x.Scene(x) \to Internal(x) \vee External(x)$

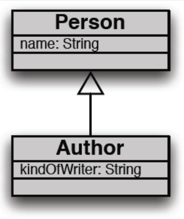
$\forall x,y.stpForScn(x,y) \to$
$\forall Setup(x) \wedge Scene(y)$
$\forall x,y.tkOfStp(x,y) \to$
$\forall Take(x) \wedge Setup(y)$
$\forall x,y.located(x,y) \to$
$\forall External(x) \wedge Location(y)$
$\forall x.Setup(x) \to$
$\forall (1 \leq \sharp\{y|stpForScn(x,y)\} \leq 1)$
$\forall y.Scene(y) \to$
$\forall (1 \leq \sharp\{x|stpForScn(x,y))\}$
$\forall x.Take(x) \to$
$\forall (1 \leq \sharp\{y|tkOfStp(x,y)\} \leq 1)$
$\forall x.Setup(y) \to$
$\forall (1 \leq \sharp\{x|stpForScn(x,y)\})$
$\forall x.External(x) \to$
$\forall (1 \leq \sharp\{y|located(x,y)\} \leq 1)$
$\ldots$

The ISA relationship is of particular importance in conceptual modeling: a class C ISA a class C' if every instance of C is also an instance of C'.

Generalization

In UML, the ISA relationship is modeled through the notion of generalization.

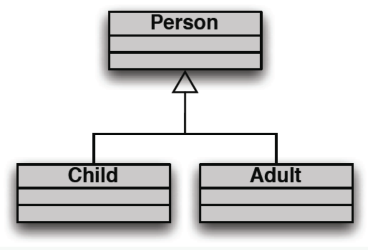
Example



The attibute 'name' is inherited by 'Author'.

A generalization involves a superclass (base class) and one or more subclasses: every instance of each subclass is also an instance of the superclass.

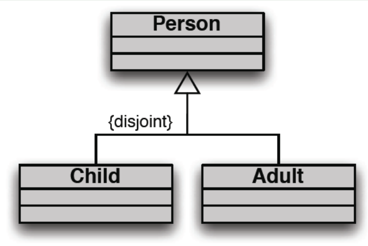
Example



Generalizations with constraints

The ability of having more subclasses in the same generalization, allows for placing suitable constraints on the classes involved in the generalization.

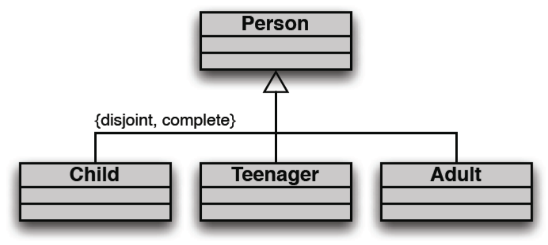
Example



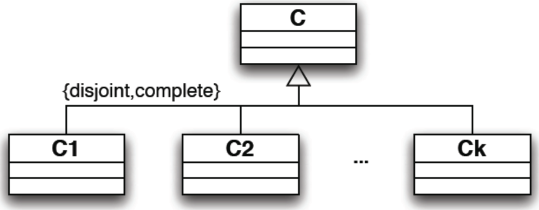
Most notable and used constraints:

- Disjointness, which asserts that different subclasses cannot have common instances (i.e., an object cannot be at the same time instance of two disjoint subclasses).
- Completeness (aka "covering"), which asserts that every instance of the superclass is also an instance of at least one of the subclasses.

Example



Generalizations: formalization

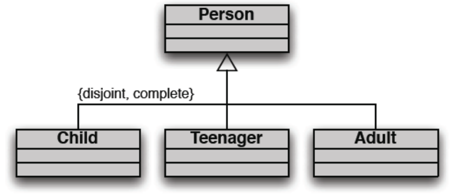


Definition (Generalization representation)

ISA:	$\forall x.C_i(x) \rightarrow C(x)$,	for $1 \leq i \leq k$
Disjointness:	$\forall x.C_i(x) \rightarrow \neg C_j(x)$,	for $1 \leq i < j \leq k$
Completeness:	$\forall x.C(x) \rightarrow \bigvee_{i=1}^{k} C_i(x)$	

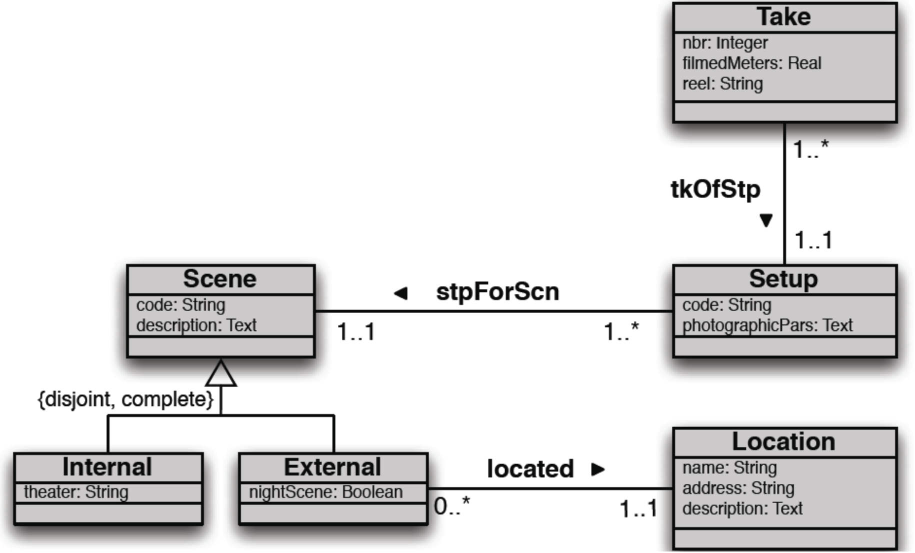
Generalizations: example of formalization

Example



$$\forall x.Child(x) \rightarrow Person(x)$$
$$\forall x.Teenager(x) \rightarrow Person(x)$$
$$\forall x.Adult(x) \rightarrow Person(x)$$
$$\forall x.Child(x) \rightarrow \neg Teenager(x)$$
$$\forall x.Child(x) \rightarrow \neg Adult(x)$$
$$\forall x.Teenager(x) \rightarrow \neg Adult(x)$$
$$\forall x.Person(x) \rightarrow (Child(x) \vee Teenager(x) \vee Adult(x))$$

In our example ...



In our Example

Alphabet

$Scene(x)$, $Setup(x)$, $Take(x)$, $Internal(x)$, $External(x)$, $Location(x)$, $stpForScn(x,y)$, $tkOfStp(x,y)$, $located(x,y)$,...

Axioms

$\forall x, y.code_{Scene}(x,y) \to Scene(x) \wedge String(y)$
$\forall x, y.description(x,y) \to Scene(x) \wedge Text(y)$
$\forall x, y.code_{Setup}(x,y) \to Setup(x) \wedge String(y)$
$\forall x, y.photographicPars(x,y) \to Setup(x) \wedge Text(y)$
$\forall x, y.nbr(x,y) \to Take(x) \wedge Integer(y)$
$\forall x, y.filmedMeters(x,y) \to Take(x) \wedge Real(y)$
$\forall x, y.reel(x,y) \to Take(x) \wedge String(y)$
$\forall x, y.theater(x,y) \to Internal(x) \wedge String(y)$
$\forall x, y.nightScene(x,y) \to External(x) \wedge Boolean(y)$
$\forall x, y.name(x,y) \to Location(x) \wedge String(y)$
$\forall x, y.address(x,y) \to Location(x) \wedge String(y)$
$\forall x, y.description(x,y) \to Location(x) \wedge Text(y)$
$\forall x.Scene(x) \to (1 \leq \sharp\{y | code_{Scene}(x,y)\} \leq 1)$
$\forall x.Internal(x) \to Scene(x)$
$\forall x.External(x) \to Scene(x)$
$\forall x.Internal(x) \to \neg External(x)$
$\forall x.Scene(x) \to Internal(x) \vee External(x)$

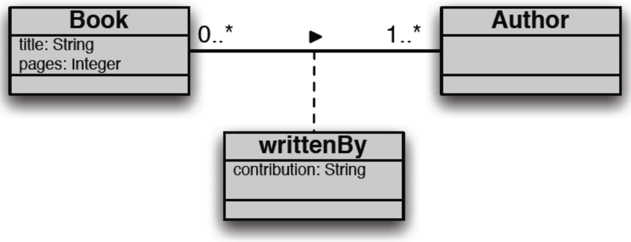
$\forall x, y.stpForScn(x,y) \to$
$\forall Setup(x) \wedge Scene(y)$
$\forall x, y.tkOfStp(x,y) \to$
$\forall Take(x) \wedge Setup(y)$
$\forall x, y.located(x,y) \to$
$\forall External(x) \wedge Location(y)$
$\forall x.Setup(x) \to$
$\forall (1 \leq \sharp\{y | stpForScn(x,y)\} \leq 1)$
$\forall y.Scene(y) \to$
$\forall (1 \leq \sharp\{x | stpForScn(x,y))\}$
$\forall x.Take(x) \to$
$\forall (1 \leq \sharp\{y | tkOfStp(x,y)\} \leq 1)$
$\forall x.Setup(y) \to$
$\forall (1 \leq \sharp\{x | stpForScn(x,y)\})$
$\forall x.External(x) \to$
$\forall (1 \leq \sharp\{y | located(x,y)\} \leq 1)$
...

Association classes

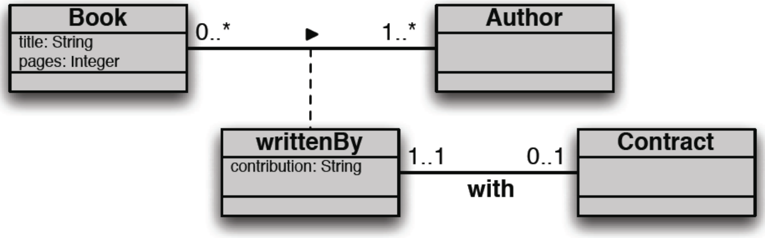
Sometimes we may want to assert properties of associations. In UML to do so we resort to association classes:

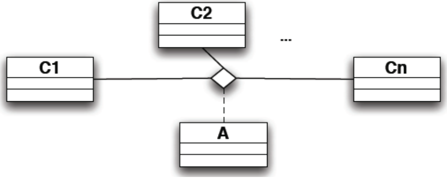
- That is, we associate to an association a class whose instances are in bijection with the tuples of the association.
- Then we use the association class exactly as a UML class (modeling local and non-local properties).

Association class - Example



Association class - Example (cont'd)

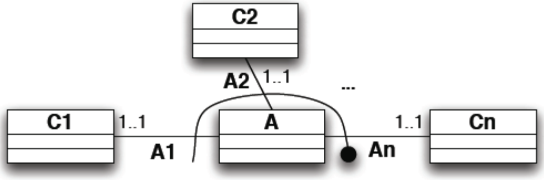




Definition (Reification)

The process of putting in correspondence objects of a class (the association class) with tuples in an association is formally described as reification.
That is:

- We introduce a unary predicate A for the association class A.
- We introduce n new binary predicates $A_1, \ldots, A_n$, one for each of the components of the association.
- We introduce suitable assertions so that objects in the extension of the unary-predicate A are in bijection with tuples in the n-ary association A.



Definition

Association Class Representation FOL Assertions are needed for stating a bijection between instances of the association class and instances of the association:

$$\forall x, y. A_i(x, y) \to A(x) \land C_i(y), \qquad for\ i \in \{1, \ldots, n\}$$

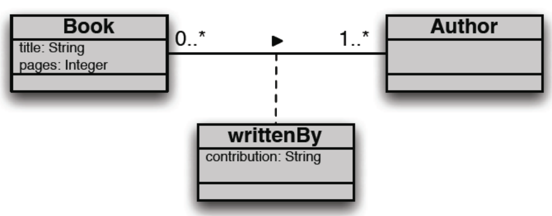
$$\forall x. A(x) \to \exists y. A_i(x, y), \qquad for\ i \in \{1, \ldots, n\}$$

$$\forall x, y, y'. A_i(x, y) \land A_i(x, y') \to y = y', \qquad for\ i \in \{1, \ldots, n\}$$

$$\forall x, x', y_1, \ldots, y_n. \bigwedge_{i=1}^{n} (A_i(x, y_i) \land A_i(x', y_i)) \to x = x'$$

Association classes: example of formalization

Example



$$\forall x, y.wb_1(x, y) \rightarrow writtenBy(x) \land Book(y)$$
$$\forall x, y.wb_2(x, y) \rightarrow writtenBy(x) \land Author(y)$$
$$\forall x.writtenBy(x) \rightarrow \exists y.wb_1(x, y)$$
$$\forall x.writtenBy(x) \rightarrow \exists y.wb_2(x, y)$$
$$\forall x, y, y'.wb_1(x, y) \land wb_1(x, y') \rightarrow y = y'$$
$$\forall x, y, y'.wb_2(x, y) \land wb_2(x, y') \rightarrow y = y'$$
$$\forall x, x', y_1, y_2.wb_1(x, y_1) \land wb_1(x', y_1) \land wb_2(x, y_2) \land wb_2(x', y_2) \rightarrow x = x'$$

Outline

Definition (Class Consistency)

A class is consistent, if the class diagram admits an instantiation in which the class has a non-empty set of instances.

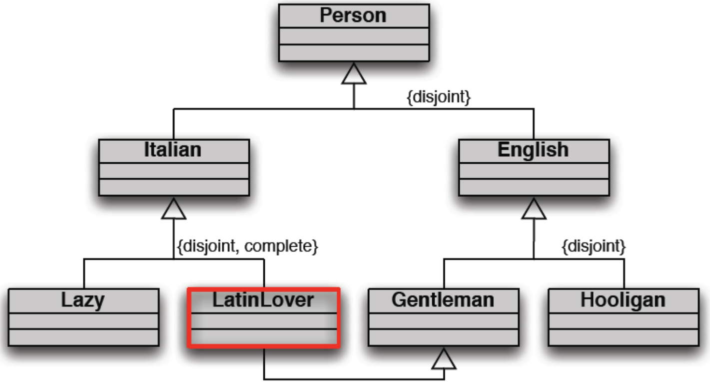
Theorem

Let Γ be the set of FOL assertions corresponding to the UML Class Diagram, and $C(x)$ the predicate corresponding to a class C of the diagram.
Then C is consistent iff

$$\Gamma \not\models \forall x.C(x) \rightarrow false$$

i.e., there exists a model of Γ in which the extension of $C(x)$ is not the empty set.

Note: Corresponding FOL reasoning task: satisfiability.



$$\Gamma \models \forall x.LatinLover(x) \rightarrow false$$

Definition (Class Diagram Consistency)

A class diagram is consistent, if it admits an instantiation, i.e., if its classes can be populated without violating any of the conditions imposed by the diagram.

Theorem

Let Γ be the set of FOL assertions corresponding to the UML Class Diagram. Then, the diagram is consistent iff

$$\Gamma \text{ is satisfiable}$$

i.e., Γ admits at least one model. (Remember that FOL models cannot be empty.)

Note: Corresponding FOL reasoning task: satisfiability.

Definition (Class Subsumption)

A class C_1 is subsumed by a class C_2 (or C_2 subsumes C_1), if the class diagram implies that C_2 is a generalization of C_1.

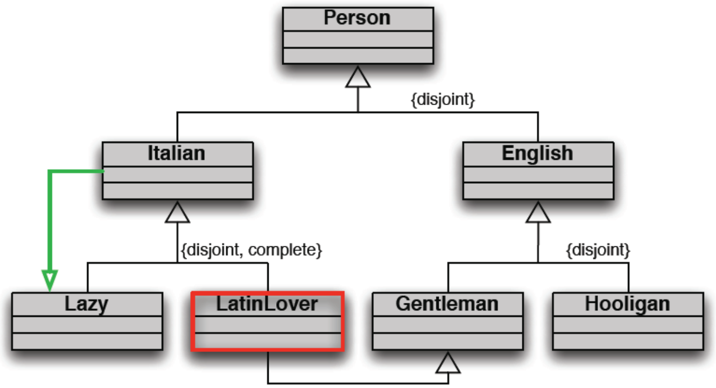
Theorem

Let Γ be the set of FOL assertions corresponding to the UML Class Diagram, and $C_1(x)$, $C_2(x)$ the predicates corresponding to the classes C_1, and C_2 of the diagram. Then C_1 is subsumed by C_2 iff

$$\Gamma \models \forall x.C_1(x) \rightarrow C_2(x)$$

Note: Corresponding FOL reasoning task: logical implication.

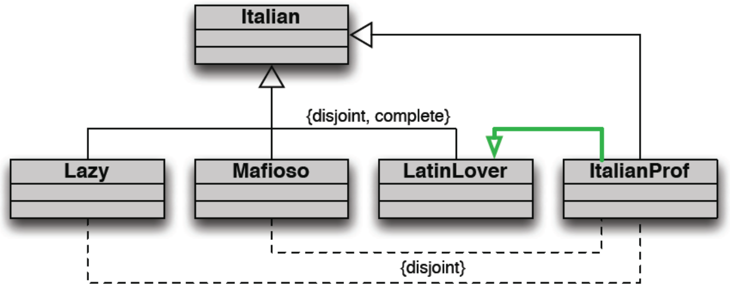
Class subsumption: example



$$\Gamma \models \forall x. LatinLover(x) \rightarrow false$$

$$\Gamma \models \forall x. Italian(x) \rightarrow Lazy(x)$$

Class subsumption: another example (by E. Franconi)



$$\Gamma \models \forall x.ItalianProf(x) \rightarrow LatinLover(x)$$

Note: this is an example of reasoning by cases.

Definition (Class Equivalence)

Two classes C_1 and C_2 are equivalent, if C_1 and C_2 denote the same set of instances in all instantiations of the class diagram.

Theorem

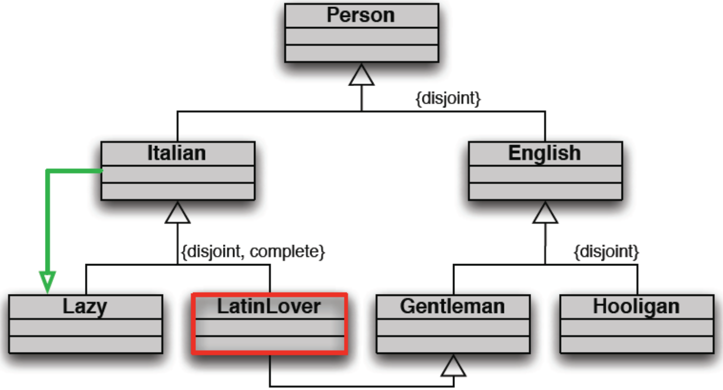
Let Γ be the set of FOL assertions corresponding to the UML Class Diagram, and $C_1(x), C_2(x)$ the predicates corresponding to the classes C_1, and C_2 of the diagram. Then C_1 and C_2 are equivalent iff

$$\Gamma \models \forall x. C_1(x) \leftrightarrow C_2(x)$$

Note:

- If two classes are equivalent then one of them is redundant.
- Determining equivalence of two classes allows for their merging, thus reducing the complexity of the diagram.

Class equivalence: example



$$\Gamma \models \forall x.ItalianLover(x) \rightarrow false$$

$$\Gamma \models \forall x.Italian(x) \rightarrow Lazy(x)$$

$$\Gamma \models \forall x.Lazy(x) \equiv Italian(x)$$

Forms of reasoning: implicit consequence

The properties of various classes and associations may interact to yield stricter multiplicities or typing than those explicitly specified in the diagram.
More generally...

Definition (Implicit Consequence)

A property $\mathcal{P}$ is an (implicit) consequence of a class diagram if $\mathcal{P}$ holds whenever all conditions imposed by the diagram are satisfied.

Theorem

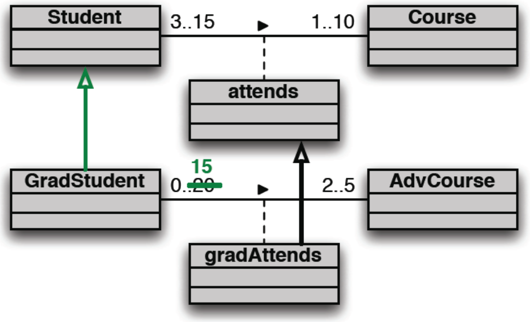
Let Γ be the set of FOL assertion corresponding to the UML Class Diagram, and $\mathcal{P}$ (the formalization in FOL of) the property of interest
Then $\mathcal{P}$ is an implicit consequence iff

$$\Gamma \models \mathcal{P}$$

i.e., the property $\mathcal{P}$ holds in every model of Γ.

Note: Corresponding FOL reasoning task: logical implication.

Implicit consequences: example

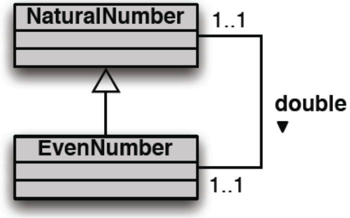


$$\Gamma \models \forall x.AdvCourse(x_2) \rightarrow \sharp\{x_1 | gradAttends(x_1, x_2)\} \leq 15$$

$$\Gamma \models \forall x.GradStudent(x) \rightarrow Student(x)$$

$$\Gamma \not\models \forall x.AdvCourse(x) \rightarrow Course(x)$$

Unrestricted vs. finite model reasoning



- Due to the multiplicities, the classes $NaturalNumber$ and $EvenNumber$ are in bijection.
  As a consequence, in every instantiation of the diagram, "the classes $NaturalNumber$ and $EvenNumber$ contain the same number of objects".
- Due to the ISA relationship, every instance of $EvenNumber$ is also an instance of $NaturalNumber$, i.e., we have that

$$\Gamma \models \forall x.EvenNumber(x) \rightarrow NaturalNumber(x)$$

Question: Does also the reverse implication hold, i.e.,

$$\Gamma \models \forall x.NaturalNumber(x) \rightarrow EvenNumber(x) \quad ?$$

- if the domain is infinite, the implication does not hold.
- If the domain is finite, the implication does hold.

Finite model reasoning: means reasoning only with respect to models with a finite domain.

- Finite model reasoning is interesting for standard databases.
- The previous example shows that in UML Class Diagrams, finite model reasoning is different form unrestricted model reasoning.

Questions

In the above examples reasoning could be easily carried out on intuitive grounds. However, two questions come up.

1. Can we develop sound, complete, and terminating procedures for reasoning on UML Class Diagrams?

- We cannot do so by directly relying on FOL!
- But we can use specialized logics with better computational properties. A form of such specialized logics are Description Logics.

2. How hard is it to reason on UML Class Diagrams in general?

- What is the worst-case situation?
- Can we single out interesting fragments on which to reason efficiently?

Note: all what we have said holds for Entity-Relationship Diagrams as well

Outline

Outline

[J.E. Hopcroft, 2007; Papadimitriou, 1994]
Computational complexity theory aims at understanding how difficult it is to solve specific problems.

- A problem is considered as an (in general infinite) set of instances of the problem, each encoded in some meaningful (i.e., compact) way.
- Standard complexity theory deals with decision problems: i.e., problems that admit a yes/no answer.
- Algorithm that solves a decision problem:
  - input: an instance of the problem
  - output: yes or no
- The difficulty (complexity) is measured in terms of the amount of resources (time, space) that the algorithm needs to solve the problem.
  $\rightsquigarrow$ complexity of the algorithm, or upper bound
- To measure the complexity of the problem, we consider the best possible algorithm that solves it.
  $\rightsquigarrow$ lower bound

- Worst-case complexity analysis: the complexity is measured in terms of a (complexity) function f:
  - argument: the size n of an instance of the problem (i.e., the length of its encoding)
  - result: the amount $f(n)$ of time/space needed in the worst-case to solve an instance of size n
- The asymptotic behaviour of the complexity function when n grows is considered.
- To abstract away from contingent issues (e.g., programming language, processor speed, etc.), we refer to an abstract computing model: Turing Machines (TMs).

To achieve robustness wrt encoding issues, usually one does not consider specific complexity functions f, but rather families $\mathcal{C}$ of complexity functions, giving rise to complexity classes.

Definition (Time/space complexity class $\mathcal{C}$)

A time/space complexity class $\mathcal{C}$ is the set of all problems P such that an instance of P of size n can be solved in time/space at most $C(n)$.

Note: Consider a (decision) problem P, and an encoding of the instances of P into strings over some alphabet Σ.
Once we fix such an encoding, the problem actually corresponds to a language L_P , namely the set of strings encoding those instances of the problem for which the answer is yes.
Hence, in the technical sense, a complexity class is actually a set of languages.

To establish lower bounds on the complexity of problems, we make use of the notion of reduction:

Definition (Reduction)

A reduction from a problem P_1 to a problem P_2 is a function R (the reduction) from instance of P_1 to instances of P_2 such that:

1. R is efficiently computable (i.e., in logarithmic space), and
2. An instance I of P_1 has answer yes iff $R(I)$ has answer yes.

P_1 reduces to P_2 if there is a reduction R from P_1 to P_2.

Intuition: If P_1 reduces to P_2, then P2 is at least as difficult as P_1, since we can solve an instance I of P_1 by reducing it to the instance $R(I)$ of P_2 and then solve $R(I)$.

Definition (Hardness)

A problem P is hard for a complexity class $\mathcal{C}$ if every problem in $\mathcal{C}$ can be reduced to P.

Definition (Completeness)

A problem P is complete for a complexity class $\mathcal{C}$ if

1. it is hard for $\mathcal{C}$, and
2. it belongs to $\mathcal{C}$

Intuitively, a problem that is complete for $\mathcal{C}$ is among the hardest problems in $\mathcal{C}$.

Outline

Tractability and intractability: PTime and NP

Definition (PTime)

Set of problems solvable in polynomial time by a deterministic TM.

- These problems are considered tractable, i.e., solvable for large inputs.
- Is a robust class (PTime computations compose).

Definition (NP)

Set of problems solvable in polynomial time by a non-deterministic TM.

- These problems are believed intractable, i.e., unsolvable for large inputs.
- The best known algorithms actually require exponential time.
- Corresponds to a large class of practical problems, for which the following type of algorithm can be used:
  1. Non-deterministically guess a possible solution of polynomial size.
  2. Check in polynomial time that the guessed solutions is good.

Definition (PSpace)

Set of problems solvable in polynomial space by a deterministic TM.

- Polynomial space is “not really good”, since these problems may require exponential time.
- These problems are considered to be more difficult than NP problems.
- Practical algorithms and heuristics work less well than for NP problems.

Definition (ExpTime)

Set of problems solvable in exponential time by a deterministic TM.

- This is the first provably intractable complexity class.
- These problems are considered to be very difficult.

Definition (NExpTime)

Set of problems solvable in exponential time by a non-deterministic TM.

Definition (LogSpace and NLogSpace)

Set of problems solvable in logarithmic space by a (non-)deterministic TM.

- Note: when measuring the space complexity, the size of the input does not count, and only the working memory (TM tape) is considered.
- Note 2: logarithmic space computations compose (this is not trivial).
- Correspond to reachability in undirected and directed graphs, respectively.

Definition (AC^0)

Set of problems solvable in constant time using a polynomial number of processors.

- These problems are solvable efficiently even for very large inputs.
- Corresponds to the complexity of model checking a fixed FO formula when the input is the model only.

Relationship between the complexity classes

The following relationships are known:

$$AC^0 \subsetneq LogSpace \subseteq NLogSpace \subseteq PTime \subseteq$$

$$\subseteq NP \subseteq PSpace \subseteq$$

$$\subseteq ExpTime \subseteq NExpTime$$

Moreover, we know that:

$$PTime \subsetneq ExpTime$$

Outline

Outline

What are Description Logics?

Description Logics (DLs) [Baader et al., 2003] are logics specifically designed to represent and reason on structured knowledge.

The domain of interest is composed of objects and is structured into:

- concepts, which correspond to classes, and denote sets of objects
- roles, which correspond to (binary) relationships, and denote binary relations on objects

The knowledge is asserted through so-called assertions, i.e., logical axioms.

Origins of Description Logics

Description Logics stem from early days knowledge representation formalisms (late '70s, early '80s):

- Semantic Networks: graph-based formalism, used to represent the meaning of sentences.
- Frame Systems: frames used to represent prototypical situations, antecedents of object-oriented formalisms.

Problems: no clear semantics, reasoning not well understood.
Description Logics (a.k.a. Concept Languages, Terminological Languages) developed starting in the mid '80s, with the aim of providing semantics and inference techniques to knowledge representation systems.

Abstractly, DLs allow one to predicate about labeled directed graphs:

- Vertexes represents real world objects.
- Vertexes's labels represents qualities of objects.
- Edges represents relations between (pairs of) objects.
- Edges' labels represents the types of relations between objects.

Every fragment of the world that can be abstractly represented in terms of a labeled directed graph is a good candidate for being represented by DLs.

What are Description Logics about? - Example 1

Exercise

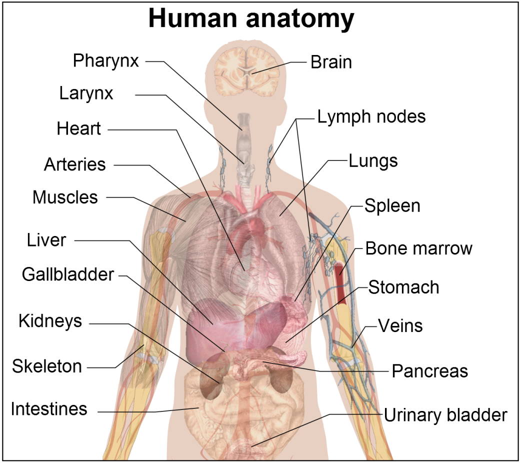
Represent Metro lines in Milan in a labelled directed graph.

What are Description Logics about? - Example 2

Exercise

Represent some aspects of Facebook as a labelled directed graph.

What are Description Logics about? - Example 3



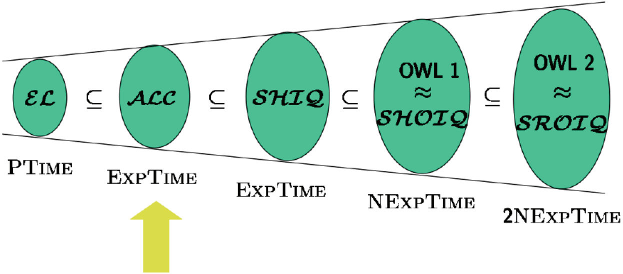
Exercise

Represent some aspects of human anatomy as a labelled directed graph.

Exercise

Represent some aspects of document classification as a labelled directed graph.

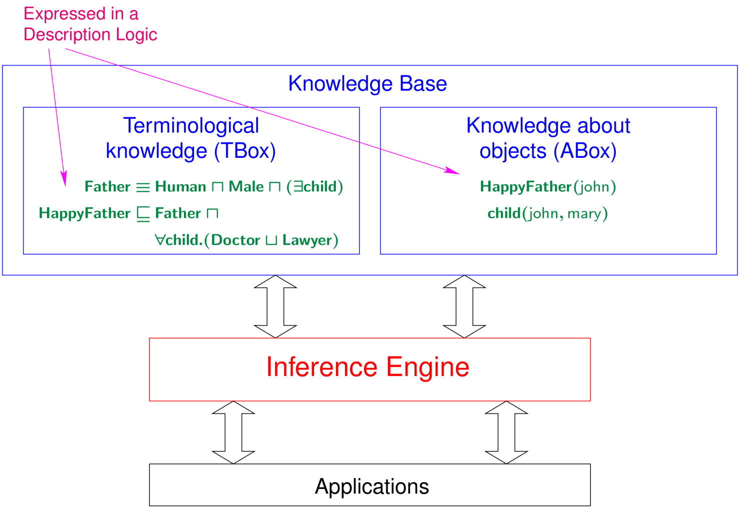
Many description logics



A DL is characterized by:

1. A description language: how to form concepts and roles
   $Human \sqcap Male \sqcap \exists hasChild \sqcap \forall hasChild.(Doctor \sqcup Lawyer)$
2. A mechanism to specify knowledge about concepts and roles (i.e., a TBox)
   $\mathcal{T} = \{Father \equiv Human \sqcap Male \sqcap \exists hasChild,$
   $HappyFather \sqsubseteq Father \sqcap \forall hasChild.(Doctor \sqcup Lawyer)\}$
3. A mechanism to specify properties of objects (i.e., an ABox)
   $\mathcal{A} = \{HappyFather(john), hasChild(john, mary)\}$
4. A set of inference services: how to reason on a given KB
   $\mathcal{T} \models HappyFather \sqsubseteq \exists hasChild.(Doctor \sqcup Lawyer)$
   $\mathcal{T} \cup \mathcal{A} \models (Doctro \sqcup Lawyer)(mary)$

Architecture of a Description Logic system



Description language

A description language provides the means for defining:

- concepts, corresponding to classes: interpreted as sets of objects;
- roles, corresponding to relationships: interpreted as binary relations on objects.

To define concepts and roles:

- We start from a (finite) alphabet of atomic concepts and atomic roles, i.e., simply names for concept and roles.
- Then, by applying specific constructors, we can build complex concepts and roles, starting from the atomic ones.

A description language is characterized by the set of constructs that are available for that.

The formal semantics of DLs is given in terms of interpretations.

Definition (Interpretation)

An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of:

- a nonempty set $\Delta^{\mathcal{I}}$, called the interpretation domain (of $\mathcal{I}$)
- an interpretation function $\cdot^{\mathcal{I}}$, which maps
  - each atomic concept A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$
  - each atomic role P to a subset $P^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$

The interpretation function is extended to complex concepts and roles according to their syntactic structure.

$\mathcal{AL}$ Concept constructors

Construct	Syntax	Example	Semantics
atomic concept	A	$Doctor$	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
atomic role	P	$hasChild$	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
atomic negation	$\neg A$	$\neg Doctor$	$\Delta^{\mathcal{I}} \backslash A^{\mathcal{I}}$
conjunction	$C \sqcap D$	$Hum \sqcap Male$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
(unqual.) exist. res.	$\exists R$	$\exists hasChild$	$\{o \mid \exists o'.(o.o') \in R^{\mathcal{I}}\}$
value restriction	$\forall R.C$	$\forall hasChild.Male$	$\{o \mid \forall o'.(o.o') \in R^{\mathcal{I}} \rightarrow o' \in C^{\mathcal{I}}\}$
bottom	$\bot$		$\emptyset$

(C, D denote arbitrary concepts and R an arbitrary role)
The above constructs form the basic language $\mathcal{AL}$ of the family of $\mathcal{AL}$ languages.

Additional concept and role constructors

Construct	$\mathcal{AL}$	Syntax	Semantics
disjunction	$\mathcal{U}$	$C \sqcup D$	$C^\mathcal{I} \cup D^\mathcal{I}$
top		$\top$	$\Delta^\mathcal{I}$
qual. exist. res.	$\mathcal{E}$	$\exists R.C$	$\{o \mid \exists o'.(o,o') \in R^\mathcal{I} \wedge o' \in C^\mathcal{I}\}$
(full) negation	$\mathcal{C}$	$\neg C$	$\Delta^\mathcal{I} \setminus C^\mathcal{I}$
number restriction	$\mathcal{N}$	$(\geq kR)$	$\{o \mid \sharp\{o' \mid (o,o') \in R^\mathcal{I}\} \geq k\}$
		$(\leq kR)$	$\{o \mid \sharp\{o' \mid (o,o') \in R^\mathcal{I}\} \leq k\}$
qual. number restriction	$\mathcal{Q}$	$(\geq kR.C)$	$\{o \mid \sharp\{o' \mid (o,o') \in R^\mathcal{I} \wedge o' \in C^\mathcal{I}\} \geq k\}$
		$(\leq kR.C)$	$\{o \mid \sharp\{o' \mid (o,o') \in R^\mathcal{I} \wedge o' \in C^\mathcal{I}\} \leq k\}$
inverse role	$\mathcal{I}$	R^-	$\{(o,o') \mid (o',o) \in R^\mathcal{I}\}$
role closure	reg	R^*	$(R^\mathcal{I})^*$

Note: Many different DL constructs and their combinations have been investigated.

Further examples of DL constructs

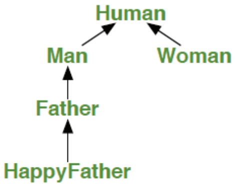
- Disjunction: $\forall hasChild.(Doctor \sqcup Lawyer)$
- Qualified existential restriction: $\exists hasChild.Doctor$
- Full negation: $\neg(Doctor \sqcup Lawyer)$
- Number restrictions: $(\geq 2\ hasChild) \sqcup (\leq 1\ sibling)$
- Qualified number restrictions: $(\geq 2\ hasChild.Doctor)$
- Inverse role: $\forall hasChild^{-}.Doctor$
- Reflexive-transitive role closure: $\exists hasChild^{*}.Doctor$

An interpretation $\mathcal{I}$ is a model of a concept C if $C^{\mathcal{I}} \neq \emptyset$.

Basic reasoning tasks

1. Concept satisfiability: does C admit a model?
2. Concept subsumption $C \sqsubseteq D$: does $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ hold for all interpretations $\mathcal{I}$

Subsumption is used to build the concept hierarchy:



Exercise

Show that if DL is propositionally closed then (1) and (2) are mutually reducible.

Complexity of concept satisfiability [Donini et al., 1997]

$\mathcal{AL}, \mathcal{ALN}$	PTime
$\mathcal{ALU}, \mathcal{ALUN}$	NP-complete
$\mathcal{ALE}$	coNP-complete
$\mathcal{ALC}, \mathcal{ALCN}, \mathcal{ALCI}, \mathcal{ALCQI}$	PSpace-complete

- Two sources of complexity:
  - union ($\mathcal{U}$) of type NP,
  - existential quantification ($\mathcal{E}$) of type coNP.
- When they are combined, the complexity jumps to PSpace.
- Number restrictions ($\mathcal{N}$) do not add to the complexity.

Outline

Structural properties vs. asserted properties

We have seen how to build complex concept and roles expressions, which allow one to denote classes with a complex structure.
However, in order to represent real world domains, one needs the ability to assert properties of classes and relationships between them (e.g., as done in UML class diagrams).
The assertion of properties is done in DLs by means of an ontology (or knowledge base).

Description Logics ontology

Definition (Description Logics ontology (or knowledge base))

A Description Logics ontology (or knowledge base) is a pair $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{T}$ is a TBox and $\mathcal{A}$ is an ABox.

Definition (Description Logics TBox)

A Description Logic TBox consists of a set of assertions on concepts and roles:

- Inclusion assertions on concepts: $C_1 \sqsubseteq C_2$
- Inclusion assertions on roles: $R_1 \sqsubseteq R_2$
- Property assertions on (atomic) roles:
  - $(\textbf{transitive} \quad P)$ $(\textbf{symmetric} \quad P)$ $(\textbf{domain} \quad P \quad C)$
  - $(\textbf{functional} \quad P)$ $(\textbf{reflexive} \quad P)$ $(\textbf{range} \quad P \quad C)$...

Definition (Description Logics ABox)

A Description Logics ABox consists of a set of assertions on individuals: (we use c_i to denote individuals)

- Membership assertions for concepts: $A(c)$
- Membership assertions for roles: $P(c_1, c_2)$
- Equality and distinctness assertions: $c_1 \approx c_2, \quad c_1 \not\approx c2$

Description Logics ontology - Example

Note: We use $C_1 \equiv C_2$ as an abbreviation for $C_1 \sqsubseteq C_2, \quad C_2 \sqsubseteq C_1$.

Example (TBox assertions)

- Inclusion assertions on concepts:

$$\begin{array}{rcl} Father & \equiv & Human \sqcap Male \sqcap \exists hasChild \\ HappyFather & \sqsubseteq & Father \sqcap \forall hasChild.(Doctor \sqcup Lawyer \sqcup HappyPerson) \\ HappyAnc & \sqsubseteq & \forall descendant.HappyFather \\ Teacher & \sqsubseteq & \neg Doctor \sqcap \neg Lawyer \end{array}$$

- Inclusion assertions on roles:
  $hasChild \sqsubseteq descendant \qquad hasFather \sqsubseteq hasChild^-$
- Property assertions on roles:
  (**transitive** descendant),(**reflexive** descendant),(**functional** hasFather)

Example (ABox membership assertions)

- $Teacher(mary), \quad hasFather(mary; john), \quad HappyAnc(john)$

Semantics of a Description Logics ontology

The semantics is given by specifying when an interpretation $\mathcal{I}$ satisfies an assertion α, denoted $\mathcal{I} \models \alpha$

Definition (Satisfiability of TBox Assertions)

- $\mathcal{I} \models C_1 \sqsubseteq C_2$ if $C_1^\mathcal{I} \subseteq C_2^\mathcal{I}$
- $\mathcal{I} \models R_1 \sqsubseteq R_2$ if $R_1^\mathcal{I} \subseteq R_2^\mathcal{I}$
- $\mathcal{I} \models (\mathbf{prop}\ P)$ if $P^\mathcal{I}$ is a relation that has the property **prop**.

(Note: domain and range assertions can be expressed by means of concept inclusion assertions.)

Definition (Satisfiability of ABox Assertions)

We need first to extend the interpretation function $\cdot^\mathcal{I}$, so that it maps each individual c to an element $c^\mathcal{I}$ of $\Delta^\mathcal{I}$.

- $\mathcal{I} \models A(c)$ if $c^\mathcal{I} \in A^\mathcal{I}$.
- $\mathcal{I} \models P(c_1, c_2)$ if $(c_1^\mathcal{I}, c_2^\mathcal{I}) \in P^\mathcal{I}$
- $\mathcal{I} \models c_1 \approx c_2$ if $c_1^\mathcal{I} = c_2^\mathcal{I}$
- $\mathcal{I} \models c_1 \not\approx c_2$ if $c_1^\mathcal{I} \neq c_2^\mathcal{I}$

Model of a Description Logics ontology

Definition (Model)

An interpretation $\mathcal{I}$ is a model of:

- an assertion α, if it satisfies α.
- a TBox $\mathcal{T}$, if it satisfies all assertions in $\mathcal{T}$.
- an ABox $\mathcal{A}$, if it satisfies all assertions in $\mathcal{A}$.
- an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ if it is a model of both $\mathcal{T}$ and $\mathcal{A}$.

Note: We use $\mathcal{I} \models \beta$ to denote that interpretation $\mathcal{I}$ is a model of β (where β stands for an assertion, TBox, ABox, or ontology).

We may make some assumptions on how individuals are interpreted.

Definition (Unique name assumption (UNA))

When c_1 and c_2 are two individuals such that $c_1 \neq c_2$, then $c_1^{\mathcal{I}} \neq c_2^{\mathcal{I}}$.

Note: When the UNA holds, equality and distinctness assertions are meaningless.

Definition (Standard name assumption (SNA))

The UNA holds, and moreover individuals are interpreted in the same way in all interpretations.
Hence, we may assume that $\Delta^{\mathcal{I}}$ contains the set of individuals, and that for each interpretation $\mathcal{I}$, we have that $c^{\mathcal{I}} = c$ (then, c is called a standard name).

Outline

1. Ontology Languages
   - Elements of an ontology language
   - Intensional and extensional level of an ontology language
   - Ontologies vs. other formalisms
2. UML class diagrams as FOL ontologies
   - Approaches to conceptual modelling
   - Formalising UML class diagram in FOL
   - Reasoning on UML class diagrams
3. Brief introduction to computational complexity
   - Basic Definitions
   - Hardness and Completeness
   - Most Important complexity classes
4. Introduction to Description Logics
   - Ingredients of Description Logics
   - Description Logics ontologies
   - Reasoning in Description Logics
   - Relationship between DLs and other representations formalisms

Logical implication

The fundamental reasoning service from which all other ones can be easily derived is ...

Definition (Logical implication)

An ontology $\mathcal{O}$ logically implies an assertion α , written $\mathcal{O} \models \alpha$ if α is satisfied by all models of $\mathcal{O}$.

We can provide an analogous definition for a TBox $\mathcal{T}$ instead of an ontology $\mathcal{O}$.

- TBox Satisfiability: $\mathcal{T}$ is satisfiable, if it admits at least one model.
- Concept Satisfiability: C is satisfiable wrt $\mathcal{T}$, if there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}}$ is not empty, i.e., $\mathcal{T} \not\models C \equiv \perp$
- Subsumption: C_1 is subsumed by C_2 wrt $\mathcal{T}$, if for every model $\mathcal{I}$ of $\mathcal{T}$ we have $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$, i.e., $\mathcal{T} \models C_1 \sqsubseteq C_2$
- Equivalence: C_1 and C_2 are equivalent wrt $\mathcal{T}$ if for every model $\mathcal{I}$ of $\mathcal{T}$ we have $C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$, i.e., $\mathcal{T} \models C_1 \equiv C_2$
- Disjointness: C_1 and C_2 are disjoint wrt $\mathcal{T}$ if for every model $\mathcal{I}$ of $\mathcal{T}$ we have $C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} = \emptyset$, i.e., $\mathcal{T} \models C_1 \sqcap C_2 \equiv \perp$
- Functionality implication: A functionality assertion $(\mathbf{funct}\ R)$ is logically implied by $\mathcal{T}$ if for every model $\mathcal{I}$ of $\mathcal{T}$, we have that $(o, o_1) \in R^{\mathcal{I}}$ and $(o, o_2) \in R^{\mathcal{I}}$ implies $o_1 = o_2$, i.e., $\mathcal{T} \models (\mathbf{funct}\ R)$

Note: Analogous definitions hold for role satisfiability, subsumption, equivalence, and disjointness.

- Ontology Satisfiability: Verify whether an ontology $\mathcal{O}$ is satisfiable, i.e., whether $\mathcal{O}$ admits at least one model.
- Concept Instance Checking: Verify whether an individual c is an instance of a concept C in every model of $\mathcal{O}$, i.e., whether $\mathcal{O} \models C(c)$.
- Role Instance Checking: Verify whether a pair (c_1, c_2) of individuals is an instance of a role R in every model of $\mathcal{O}$, i.e., whether $\mathcal{O} \models R(c_1, c_2)$.

Reasoning in Description Logics - Example

Example (TBox)

- Inclusion assertions on concepts:

$$\begin{array}{rcl} Father & \equiv & Human \sqcap Male \sqcap \exists hasChild \\ HappyFather & \sqsubseteq & Father \sqcap \forall hasChild.(Doctor \sqcup Lawyer \sqcup HappyPerson) \\ HappyAnc & \sqsubseteq & \forall descendant.HappyFather \\ Teacher & \sqsubseteq & \neg Doctor \sqcap \neg Lawyer \end{array}$$

- Inclusion assertions on roles:
  $hasChild \sqsubseteq descendant$ $\qquad$ $hasFather \sqsubseteq hasChild^{-}$
- Property assertions on roles:
  (**transitive** descendant), (**reflexive** descendant), (**functional** hasFather)

The above TBox logically implies: $HappyAncestor \sqsubseteq Father.$

Example (ABox)

- Membership assertions:
  $Teacher(mary)$, $\qquad$ $hasFather(mary; john)$, $\qquad$ $HappyAnc(john)$

The above TBox and ABox logically imply: $HappyPerson(mary)$

Relationship among TBox reasoning tasks

The TBox reasoning tasks are mutually reducible to each other, provided the description language is propositionally closed:

Theorem (TBox satisfiability to concept satisfiability to concept non-subsumption)

$\mathcal{T}$ *satisfiable* *iff* $\mathcal{T} \not\models \top \equiv \bot$ *iff* *not* $\mathcal{T} \models \top \sqsubseteq \bot$
(i.e., $\top$ *satisfiable w.r.t.* $\mathcal{T}$ *)*

Theorem (Concept subsumption to concept unsatisfiability)

$\mathcal{T} \models C_1 \sqsubseteq C_2$ *iff* $\mathcal{T} \models C_1 \sqcap \neg C_2 \equiv \bot$
(i.e., $\models C_1 \sqcap \neg C_2$ *unsatisfiable w.r.t.* $\mathcal{T}$ *)*

Theorem (Concept satisfiability to TBox satisfiability)

$\mathcal{T} \not\models C \equiv \bot$ *iff* $\mathcal{T} \cup \{\top \sqsubseteq \exists P_{new} \sqcap \forall P_{new}.C\}$ *satisfiable*
(where P_{new} *is a new atomic role)*

TBox reasoning can be reduced to reasoning over an ontology:

Theorem (Concept satisfiability to ontology satisfiability)

C satisfiable wrt $\mathcal{T}$ iff $\langle \mathcal{T} \cup \{A_{new} \sqsubseteq C\}, \{A(c_{new})\} \rangle$ is satisfiable (where A_{new} is a new atomic concept and c_{new} is a new individual)

Exercise

Show mutual reductions between the remaining (TBox and ontology) reasoning tasks.

Internalization of the TBox

- In some (very expressive) DLs, it is possible to reduce reasoning wrt a TBox to reasoning over concept expressions only, i.e., the whole TBox can be internalized into a single concept.
- Whether this is possible depends on the available role and concept constructors, and the details differ for each DL.

Complexity of reasoning over DL ontologies

Reasoning over DL ontologies is much more complex than reasoning over concept expressions:

Bad news:

- without restrictions on the form of TBox assertions, reasoning over DL ontologies is already ExpTime-hard, even for very simple DLs (see, e.g.,[Donini, 2003]).

Good news:

- We can add a lot of expressivity (i.e., essentially all DL constructs seen so far), while still staying within the ExpTime upper bound.
- There are DL reasoners that perform reasonably well in practice for such DLs (e.g, Racer, Pellet, Fact++, . . .) [Möoller and Haarslev, 2003].

Outline

Most DLs are well-behaved fragments of First Order Logic.

Definition (From $\mathcal{ALC}$ TBox to FOL)

To translate an $\mathcal{ALC}$ TBox to FOL:

1. Introduce: a unary predicate A(x) for each atomic concept A
   a binary predicate $P(x, y)$ for each atomic role P
2. Translate complex concepts using translation functions t_x, one for each variable x:

$$t_x(A) = A(x) \qquad t_x(C \sqcap D) = t_x(C) \wedge t_x(D)$$

$$t_x(\neg C) = \neg t_x(C) \qquad t_x(C \sqcup D) = t_x(C) \vee t_x(D)$$

$$t_x(\exists P.C) = \exists y.P(x, y) \wedge t_y(C)$$

$$t_x(\forall P.C) = \forall y.P(x, y) \rightarrow t_y(C) \qquad (\textit{with } y \textit{ a new variable})$$

3. Translate a TBox $\mathcal{T} = \bigcup_i \{C_i \sqsubseteq D_i\}$ as the FOL theory:
$$\Gamma_{\mathcal{T}} = \bigcup_i \{\forall x.t_x(C_i) \rightarrow t_x(D_i)\}$$
4. Translate an ABox $\mathcal{A} = \bigcup_i \{A_i(c_i)\} \cup \bigcup_j \{P_j(c'_j, c''_j)\}$ as the FOL theory:
$$\Gamma_{\mathcal{A}} = \bigcup_i \{A_i(c_i)\} \cup \bigcup_j \{P_j(c'_j, c''_j)\}$$

There is a direct correspondence between DL reasoning services and FOL reasoning services:

Theorem

C *is satisfiable*	*iff*	*its translation* $t_x(C)$ *is satisfiable*
$C \sqsubseteq D$	*iff*	$t_x(C) \rightarrow t_x(D)$ *is valid*
C *is satisfiable w.r.t.* $\mathcal{T}$	*iff*	$\Gamma_{\mathcal{T}} \cup \{\exists x.t_x(C)\}$ *is satisfiable*
$\mathcal{T} \models C \sqsubseteq D$	*iff*	$\Gamma_{\mathcal{T}} \models \forall x.(t_x(C) \rightarrow t_x(D))$

Exercise

Translate the following $\mathcal{ALC}$ concepts into FOL formulas:

1. $Father \sqcap \forall child.(Doctor \sqcup Manager)$
2. $\exists manages.(Company \sqcap \exists employs.Doctor)$
3. $Father \sqcap \forall child.(Doctor \sqcup \exists manages.(Company \sqcap \exists employs.Doctor))$

Solution

1. $Father(x) \land \forall y.(child(x,y) \rightarrow (Doctor(y) \lor Manager(y)))$
2. $\exists y.(manages(x,y) \land (Company(y) \land \exists w.(employs(y,w) \land Doctor(w))))$
3. $Father(x) \land \forall y.(child(x,y) \rightarrow (Doctor(y) \lor \exists w.(manages(y,w) \land (Company(w) \land \exists z.(employs(w,z) \land Doctor(z))))))$

The previous translation shows us that DLs are a fragment of First Order Logic.
In particular, we can translate complex concepts using just two translation functions t_x and t_y (thus reusing the same variables):

$$
\begin{aligned}
t_x(A) &= A(x) & t_y(A) &= A(y) \\
t_x(\neg C) &= \neg C(x) & t_y(\neg C) &= \neg C(y) \\
t_x(C \sqcap D) &= t_x(C) \wedge t_x(D) & t_y(C \sqcap D) &= t_y(C) \wedge t_y(D) \\
t_x(C \sqcup D) &= t_x(C) \vee t_x(D) & t_y(C \sqcup D) &= t_y(C) \vee t_y(D) \\
t_x(\exists P.C) &= \exists y.P(x,y) \wedge t_y(C) & t_y(\exists P.C) &= \exists x.P(y,x) \wedge t_x(C) \\
t_x(\forall P.C) &= \forall y.P(x,y) \rightarrow t_y(C) & t_y(\forall P.C) &= \forall x.P(y,x) \rightarrow t_x(C)
\end{aligned}
$$

$\rightsquigarrow$ $\mathcal{ALC}$ is a fragment of L2, i.e., FOL with 2 variables, known to be decidable (NExpTime-complete).
Note: FOL with 2 variables is more expressive than ALC (tradeoff expressive power vs. complexity of reasoning).

Exercise

Translate the following $\mathcal{ALC}$ concepts into FOL formulas: (i.e., into FOL formulas that use only variables x and y):

1. $Father \sqcap \forall child.(Doctor \sqcup Manager)$
2. $\exists manages.(Company \sqcap \exists employs.Doctor)$
3. $Father \sqcap \forall child.(Doctor \sqcup \exists manages.(Company \sqcap \exists employs.Doctor))$

Solution

1. $Father(x) \land \forall y.(child(x,y) \to (Doctor(y) \lor Manager(y)))$
2. $\exists y.(manages(x,y) \land (Company(y) \land \exists x.(employs(y,x) \land Doctor(x))))$
3. $Father(x) \land \forall y.(child(x,y) \to (Doctor(y) \lor \exists x.(manages(y,x) \land (Company(x) \land \exists y.(employs(x,y) \land Doctor(y))))))$

DLs as fragments of First Order Logic (Cont'd)

The previous translations can be extended to other constructs:

- For inverse roles, swap the variables in the role predicate, i.e.,
  $t_x(\exists P^-.C) = \exists y.P(y,x) \wedge t_y(C)$ with y a new variable
  $t_x(\forall P^-.C) = \forall y.P(y,x) \rightarrow t_y(C)$ with y a new variable
  $\rightsquigarrow$ $\mathcal{ALCI}$ is still a fragment of $L2$
- For number restrictions, two variables do not suffice
  $\rightsquigarrow$ $\mathcal{ALCQI}$ is a fragment of $C2$ (i.e, $L2$+counting quantifiers)

Relationship between DLs and ontology formalisms

- DLs are nowadays advocated to provide the foundations for ontology languages.
- Different versions of the W3C standard Web Ontology Language (OWL) have been defined as syntactic variants of certain DLs.
- DLs are also ideally suited to capture the fundamental features of conceptual modeling formalism used in information systems design:
  - Entity-Relationship diagrams, used in database conceptual modeling
  - UML Class Diagrams, used in the design phase of software applications
- We briefly overview the correspondence with OWL, highlighting essential DL constructs.
- We will come back a bit later to the correspondence between UML Class Diagrams and DLs.

The Web Ontology Language (OWL) comes in different variants:

- OWL1 Lite is a variant of the DL $\mathcal{SHIF}(\mathcal{D})$, where:
  - $\mathcal{S}$ stands for $\mathcal{ALC}$ extended with transitive roles,
  - $\mathcal{H}$ stands for role hierarchies (i.e., role inclusion assertions),
  - $\mathcal{I}$ stands for inverse roles,
  - $\mathcal{F}$ stands for functionality of roles,
  - $(\mathcal{D})$ stand for data types, which are necessary in any practical knowledge representation language.
- OWL1 DL is a variant of $\mathcal{SHOIN}(\mathcal{D})$, where:
  - $\mathcal{O}$ stands for nominals, which means the possibility of using individuals in the TBox (i.e., the intensional part of the ontology),
  - $\mathcal{N}$ stands for (unqualified) number restrictions.

DLs vs. OWL2

A new version of OWL, OWL2, is currently being standardized by the W3C:

- OWL2 DL is a variant of $\mathcal{SROIQ}(\mathcal{D})$, which adds to OWL1 DL several constructs, while still preserving decidability of reasoning.
  - $\mathcal{Q}$ stands for qualified number restrictions.
  - $\mathcal{R}$ stands for regular role hierarchies, where role chaining might be used inthe left-hand side of role inclusion assertions, with suitable acyclicity conditions.
- OWL2 defines also three profiles: OWL2 QL, OWL2 EL, OWL2 RL.
  - Each profile corresponds to a syntactic fragment (i.e., a sub-language) of OWL2 DL that is targeted towards a specific use.
  - The restrictions in each profile guarantee better computational properties than those of OWL2 DL.
  - The OWL2 QL profile is derived from the DLs of the DL-Lite family (see later).

DL constructs vs. OWL constructs

OWL contructor	DL constructor	Example
`ObjectIntersectionOf`	$C_1 \sqcap \cdots \sqcap c_n$	$Human \sqcap Male$
`ObjectUnionOf`	$C_1 \sqcup \cdots \sqcup C_n$	$Doctor \sqcup Lawyer$
`ObjectComplementOf`	$\neg C$	$\neg Male$
`ObjectOneOf`	$\{a_1\} \sqcup \ldots \{a_n\}$	$\{john\} \sqcup \{mary\}$
`ObjectAllValuesFrom`	$\forall P.C$	$\forall hasChild.Doctor$
`ObjectSomeValuesFrom`	$\exists P.C$	$\exists hasChild.Lawyer$
`ObjectMaxCardinality`	$(\leq nP)$	$(\leq 1 hasChild)$
`ObjectMinCardinality`	$(\geq nP)$	$(\geq 2 hasChild)$
...		

Note: all constructs come also in the Data... instead of Object... variant.

DL axioms vs. OWL axioms

OWL axiom	DL syntax	Example
`SubClassOf`	$C_1 \sqsubseteq C_2$	$Human \sqsubseteq Animal \sqcap Biped$
`EquivalentClasses`	$C_1 \equiv C_2$	$Man \equiv Human \sqcap Male$
`DisjointClasseses`	$C_1 \sqsubseteq \neg C_2$	$Man \sqsubseteq \neg Female$
`SameIndividual`	$\{a_1\} \equiv \{a_2\}$	$\{presBush\} \equiv \{G.W.Bush\}$
`DifferentIndividuals`	$\{a_1\} \sqsubseteq \neg\{a_2\}$	$\{john\} \sqsubseteq \neg\{peter\}$
`SubObjectPropertyOf`	$P_1 \sqsubseteq P_2$	$hasDaughter \sqsubseteq hasChild$
`EquivalentObjectProperties`	$P_1 \equiv P_2$	$hasCost \equiv hasPrice$
`InverseObjectProperties`	$P_1 \equiv P_2^-$	$hasChild \equiv hasParent^-$
`TransitiveObjectProperty`	$P^+ \sqsubseteq P$	$ancestor^+ \sqsubseteq ancestor$
`FunctionalObjectProperty`	$\top \sqsubseteq (\leq 1 P)$	$\top \sqsubseteq (\leq 1 hasFather)$
...		