First-Order Logic

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2021/2022

Roadmap

- **Classical First-Order Logic**
  - syntax; semantics; decision problems SAT and VAL;
  - normal forms; Herbrandization; Skolemization;
  - FOL with equality; many-sorted FOL.
- **Modeling with FOL**
  - Formalization of domain models in FOL.

(Classical) First-Order Logic

Introduction

First-order logic (FOL) is a richer language than propositional logic. Its lexicon contains not only the symbols $\wedge$, $\vee$, $\neg$, and $\rightarrow$ (and parentheses) from propositional logic, but also the symbols $\exists$ and $\forall$ for "there exists" and "for all", along with various symbols to represent variables, constants, functions, and relations.

There are two sorts of things involved in a first-order logic formula:

- *terms*, which denote the objects that we are talking about;
- *formulas*, which denote truth values.

Examples:

"Not all birds can fly."
"Every mother is older than her children."
"John and Peter have the same maternal grandmother."

Syntax

The alphabet of a first-order language is organised into the following categories.

- *Variables:* $x, y, z, \ldots \in \mathcal{X}$ (arbitrary elements of an underlying domain)
- *Constants:* $a, b, c, \ldots \in \mathcal{C}$ (specific elements of an underlying domain)
- *Functions:* $f, g, h, \ldots \in \mathcal{F}$ (every function f has a fixed arity, $\mathsf{ar}(f)$)
- *Predicates:* $P, Q, R, \ldots \in \mathcal{P}$ (every predicate P has a fixed arity, $\mathsf{ar}(P)$)
- *Logical connectives:* $\top$, $\bot$, $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\forall$ (*for all*), $\exists$ (*there exists*)
- *Auxiliary symbols*: ".", "(" and ")".

We assume that all these sets are disjoint. $\mathcal{C}$, $\mathcal{F}$ and $\mathcal{P}$ are the non-logical symbols of the language. These three sets constitute the *vocabulary* $\mathcal{V} = \mathcal{C} \cup \mathcal{F} \cup \mathcal{P}$.

Syntax

Terms

The set of *terms* of a first-order language over a vocabulary $\mathcal{V}$ is given by the following abstract syntax

$$\mathbf{Term}_{\mathcal{V}} \ni t \ ::= \ x \mid c \mid f(t_1, \ldots, t_{\mathsf{ar}(f)})$$

Formulas

The set $\mathbf{Form}_{\mathcal{V}}$, of *formulas* of FOL, is given by the abstract syntax

$$\mathbf{Form}_{\mathcal{V}} \ni \phi, \psi \ ::= \ P(t_1, \ldots, t_{\mathsf{ar}(P)}) \mid \bot \mid \top \mid (\neg\phi) \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid (\phi \rightarrow \psi) \mid (\forall x.\, \phi) \mid (\exists x.\, \phi)$$

An *atomic formula* has the form $\bot$, $\top$, or $P(t_1, \ldots, t_{\mathsf{ar}(P)})$. A *ground term* is a term without variables. *Ground formulas* are formulas without variables, i.e., quantifier-free formulas ϕ such that all terms occurring in ϕ are ground terms.

Syntax

Convention

We adopt some syntactical conventions to lighten the presentation of formulas:

- Outermost parenthesis are usually dropped.
- In absence of parentheses, we adopt the following convention about precedence. Ranging from the highest precedence to the lowest, we have respectively: $\neg$, $\wedge$, $\vee$ and $\rightarrow$. Finally we have that $\rightarrow$ binds more tightly than $\forall$ and $\exists$.
- All binary connectives are right-associative.
- Nested quantifications such as $\forall x. \forall y. \phi$ are abbreviated to $\forall x, y.\, \phi$.
- $\forall \overline{x}. \phi$ denotes the nested quantification $\forall x_1, \ldots, x_n.\, \phi$.

Modeling with FOL

"Not all birds can fly."

We can code this sentence assuming the two unary predicates B and F expressing

$B(x)$ – x is a bird
$F(x)$ – x can fly

The declarative sentence "Not all birds can fly" can now be coded as

$$\neg(\forall x.\, B(x) \rightarrow F(x))$$

or, alternatively, as

$$\exists x. B(x) \wedge \neg F(x)$$

Modeling with FOL

"Every mother is older than her children."
"John and Peter have the same maternal grandmother."

Using constants symbols j and p for John and Peter, and predicates $=$, $mother$ and $older$ expressing that

$mother(x,y)$ – x is mother of y
$older(x,y)$ – x is older than y

these sentences could be expressed by

$$\forall x.\forall y.\, mother(x,y) \to older(x,y)$$

$$\forall x,y,u,v.\, mother(x,y) \land mother(y,j) \land mother(u,v) \land mother(v,p) \to x = u$$

A different and more elegant encoding is to represent y'mother in a more direct way, by using a function instead of a relation. We write $m(y)$ to mean y'mother. This way the two sentences above have simpler encondings.

$$\forall x.\, older(m(x),x) \quad \text{and} \quad m(m(j)) = m(m(p))$$

Modeling with FOL

Assume further the following predicates and constant symbols

$flower(x)$ – x is a flower	$likes(x,y)$ – x likes y
$sport(x)$ – x is a sport	$brother(x,y)$ – x is brother of y
a – Anne	

- "Anne likes John's brother." $\exists x.\, brother(x,j) \land likes(a,x)$
- "John likes all sports." $\forall x.\, sports(x) \to likes(j,x)$
- "John's mother likes flowers." $\forall x.\, flower(x) \to likes(m(j),x)$
- "John's mother does not like some sports." $\exists y.\, sport(y) \land \neg likes(m(j),y)$
- "Peter only likes sports." $\forall x.\, likes(p,x) \to sports(x)$
- "Anne has two children."

$$\exists x_1,x_2.\, mother(a,x_1) \land mother(a,x_2) \land x_1 \neq x_2 \land$$
$$\forall z.\, mother(a,z) \to z = x_1 \lor z = x_2$$

Free and bound variables

- The *free variables* of a formula ϕ are those variables occurring in ϕ that are not quantified. FV(ϕ) denotes the set of free variables occurring in ϕ.
- The *bound variables* of a formula ϕ are those variables occurring in ϕ that do have quantifiers. BV(ϕ) denote the set of bound variables occurring in ϕ.

Note that variables can have both free and bound occurrences within the same formula. Let ϕ be $\exists x.\, R(x,y) \land \forall y.\, P(y,x)$, then

$$\text{FV}(\phi) = \{y\} \quad \text{and} \quad \text{BV}(\phi) = \{x,y\}.$$

- A formula ϕ is *closed* (or *a sentence*) if it does not contain any free variables.
- If FV$(\phi) = \{x_1,\ldots,x_n\}$, then
  - its *universal closure* is $\forall x_1.\ldots.\forall x_n.\,\phi$
  - its *existential closure* is $\exists x_1.\ldots.\exists x_n.\,\phi$

Substitution

Substitution

- We define $u[t/x]$ to be the term obtained by replacing each occurrence of variable x in u with t.
- We define $\phi[t/x]$ to be the formula obtained by replacing each free occurrence of variable x in ϕ with t.

Care must be taken, because substitutions can give rise to undesired effects.

Given a term t, a variable x and a formula ϕ, we say that *t is free for x in ϕ* if no free x in ϕ occurs in the scope of $\forall z$ or $\exists z$ for any variable z occurring in t.

From now on we will assume that all substitutions satisfy this condition. That is when performing the $\phi[t/x]$ we are always assuming that t is free for x in ϕ.

Substitution

Convention

We write $\phi(x_1,\ldots,x_n)$ to denote a formula having free variables $x_1,\ldots,x_n$. We write $\phi(t_1,\ldots,t_n)$ to denote the formula obtained by replacing each free occurrence of x_i in ϕ with the term t_i. When using this notation, it should always be assumed that each t_i is free for x_i in ϕ. Also note that when writhing $\phi(x_1,...,x_n)$ we do not mean that $x_1,...,x_n$ are the only free variables of ϕ.

Semantics

$\mathcal{V}$-structure

Let $\mathcal{V}$ be a vocabulary. A *$\mathcal{V}$-structure* $\mathcal{M}$ is a pair $\mathcal{M} = (D, I)$ where D is a nonempty set called the *interpretation domain*, and I is an *interpretation function* that assigns constants, functions and predicates over D to the symbols of $\mathcal{V}$ as follows:

- for each constant symbol $c \in \mathcal{C}$, the interpretation of c is a constant $I(c) \in D$;
- for each $f \in \mathcal{F}$, the interpretation of f is a function $I(f) : D^{\mathsf{ar}(f)} \to D$;
- for each $P \in \mathcal{P}$, the interpretation of P is a function $I(P) : D^{\mathsf{ar}(P)} \to \{0,1\}$. In particular, 0-ary predicate symbols are interpreted as truth values.

$\mathcal{V}$-structures are also called *models* for $\mathcal{V}$.

Semantics

Assignment

An *assignment* for a domain D is a function $\alpha : \mathcal{X} \to D$.

We denote by $\alpha[x \mapsto a]$ the assignment which maps x to a and any other variable y to $\alpha(y)$.

Given a $\mathcal{V}$-structure $\mathcal{M} = (D, I)$ and given an assignment $\alpha : \mathcal{X} \to D$, we define an *interpretation function for terms*, $\alpha_{\mathcal{M}} : \mathbf{Term}_{\mathcal{V}} \to D$, as follows:

$$\begin{array}{lcl}
\alpha_{\mathcal{M}}(x) & = & \alpha(x) \\
\alpha_{\mathcal{M}}(c) & = & I(c) \\
\alpha_{\mathcal{M}}(f(t_1,\ldots,t_n)) & = & I(f)(\alpha_{\mathcal{M}}(t_1),\ldots,\alpha_{\mathcal{M}}(t_n))
\end{array}$$

Semantics

Satisfaction relation

Given a $\mathcal{V}$-structure $\mathcal{M} = (D, I)$ and given an assignment $\alpha : \mathcal{X} \to D$, we define the *satisfaction relation* $\mathcal{M}, \alpha \models \phi$ for each $\phi \in \mathbf{Form}_{\mathcal{V}}$ as follows:

$$\begin{array}{lll}
\mathcal{M},\alpha \models \top & & \\
\mathcal{M},\alpha \not\models \bot & & \\
\mathcal{M},\alpha \models P(t_1,\ldots,t_n) & \text{iff} & I(P)(\alpha_{\mathcal{M}}(t_1),\ldots,\alpha_{\mathcal{M}}(t_n)) = 1 \\
\mathcal{M},\alpha \models \neg\phi & \text{iff} & \mathcal{M},\alpha \not\models \phi \\
\mathcal{M},\alpha \models \phi \land \psi & \text{iff} & \mathcal{M},\alpha \models \phi \text{ and } \mathcal{M},\alpha \models \psi \\
\mathcal{M},\alpha \models \phi \lor \psi & \text{iff} & \mathcal{M},\alpha \models \phi \text{ or } \mathcal{M},\alpha \models \psi \\
\mathcal{M},\alpha \models \phi \to \psi & \text{iff} & \mathcal{M},\alpha \not\models \phi \text{ or } \mathcal{M},\alpha \models \psi \\
\mathcal{M},\alpha \models \forall x.\,\phi & \text{iff} & \mathcal{M},\alpha[x \mapsto a] \models \phi \text{ for all } a \in D \\
\mathcal{M},\alpha \models \exists x.\,\phi & \text{iff} & \mathcal{M},\alpha[x \mapsto a] \models \phi \text{ for some } a \in D
\end{array}$$

Validity and satisfiability

When $\mathcal{M}, \alpha \models \phi$, we say that $\mathcal{M}$ *satisfies* ϕ *with* α.

We write $\mathcal{M} \models \phi$ iff $\mathcal{M}, \alpha \models \phi$ holds for every assignment α.

A formula ϕ is

valid	iff	$\mathcal{M}, \alpha \models \phi$ holds for all structure $\mathcal{M}$ and assignments α. A valid formula is called a *tautology*. We write $\models \phi$.
satisfiable	iff	there is some structure $\mathcal{M}$ and some assigment α such that $\mathcal{M}, \alpha \models \phi$ holds.
unsatisfiable	iff	it is not satisfiable. An unsatisfiable formula is called a *contradiction*.
refutable	iff	it is not valid.

Consequence and equivalence

Given a set of formulas Γ, a model $\mathcal{M}$ and an assignment α, $\mathcal{M}$ is said to *satisfy* Γ *with* α, denoted by $\mathcal{M}, \alpha \models \Gamma$, if $\mathcal{M}, \alpha \models \phi$ for every $\phi \in \Gamma$.

Γ *entails* ϕ (or that ϕ is a *logical consequence* of Γ), denoted by $\Gamma \models \phi$, iff for all structures $\mathcal{M}$ and assignments α, whenever $\mathcal{M}, \alpha \models \Gamma$ holds, then $\mathcal{M}, \alpha \models \phi$ holds as well.

ϕ is *logically equivalent* to ψ, denoted by $\phi \equiv \psi$, iff $\{\phi\} \models \psi$ and $\{\psi\} \models \phi$.

Deduction theorem

$\Gamma, \phi \models \psi \quad \text{iff} \quad \Gamma \models \phi \to \psi$

Consistency

The set Γ is *consistent* or *satisfiable* iff there is a model $\mathcal{M}$ and an assignment α such that $\mathcal{M}, \alpha \models \phi$ holds for all $\phi \in \Gamma$.

We say that Γ is *inconsistent* iff it is not consistent and denote this by $\Gamma \models \bot$.

Proposition

- $\{\phi, \neg\phi\} \models \bot$
- If $\Gamma \models \bot$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \models \bot$.
- $\Gamma \models \phi \quad \text{iff} \quad \Gamma, \neg\phi \models \bot$

Substitution

- Formula ψ is a *subformula* of formula ϕ if it occurs syntactically within ϕ.
- Formula ψ is a *strict subformula* of ϕ if ψ is a subformula of ϕ and $\psi \neq \phi$

Substitution theorem

Suppose $\phi \equiv \psi$. Let θ be a formula that contains ϕ as a subformula. Let θ' be the formula obtained by safe replacing (i.e., avoiding the capture of free variables of ϕ) some occurrence of ϕ in θ with ψ. Then $\theta \equiv \theta'$.

Adquate sets of connectives for FOL

Renaming of bound variables

If y is free for x in ϕ and $y \notin \mathsf{FV}(\phi)$, then the following equivalences hold.

- $\forall x.\phi \equiv \forall y.\phi[y/x]$
- $\exists x.\phi \equiv \exists y.\phi[y/x]$

Lemma

The following equivalences hold in first-order logic.

$$\begin{array}{rcl} \forall x.\phi \wedge \psi \equiv (\forall x.\phi) \wedge (\forall x.\psi) & & \exists x.\phi \vee \psi \equiv (\exists x.\phi) \vee (\exists x.\psi) \\ \forall x.\phi \equiv (\forall x.\phi) \wedge \phi[t/x] & & \exists x.\phi \equiv (\exists x.\phi) \vee \phi[t/x] \\ \neg\forall x.\phi \equiv \exists x.\neg\phi & & \neg\exists x.\phi \equiv \forall x.\neg\phi \end{array}$$

As in propositional logic, there is some redundancy among the connectives and quantifiers since $\forall x.\,\phi \equiv \neg\exists x.\,\neg\phi$ and $\exists x.\,\phi \equiv \neg\forall x.\,\neg\phi$.

Decidability

Given formulas ϕ and ψ as input, we may ask:

Decision problems

Validity problem:	"Is ϕ valid ?"
Satisfiability problem:	"Is ϕ satisfiable ?"
Consequence problem:	"Is ψ a consequence of ϕ ?"
Equivalence problem:	"Are ϕ and ψ equivalent ?"

These are, in some sense, variations of the same problem.

ϕ is valid	iff	$\neg\phi$ is unsatisfiable
$\phi \models \psi$	iff	$\neg(\phi \to \psi)$ is unsatisfiable
$\phi \equiv \psi$	iff	$\phi \models \psi$ and $\psi \models \phi$
ϕ is satisfiable	iff	$\neg\phi$ is not valid

Decidability

A *solution* to a decision problem is a program that takes problem instances as input and always terminates, producing a correct "yes" or "no" output.

- A decision problem is *decidable* if it has a solution.
- A decision problem is *undecidable* if it is not decidable.

Theorem (Church & Turing)

- The decision problem of validity in first-order logic is undecidable: no program exists which, given any ϕ, decides whether $\models \phi$.
- The decision problem of satisfiability in first-order logic is undecidable: no program exists which, given any ϕ, decides whether ϕ is satisfiable.

Semi-decidability

However, there is a procedure that halts and says "yes" if ϕ is valid.

A decision problem is *semi-decidable* if exists a procedure that, given an input,

- halts and answers "yes" iff "yes" is the correct answer,
- halts and answers "no" if "no" is the correct answer, or
- does not halt if "no" is the correct answer

Unlike a decidable problem, the procedure is only guaranteed to halt if the correct answer is "yes".

The decision problem of validity in first-order logic is semi-decidable.

Normal forms

A first-order formula is in *negation normal form (NNF)* if the implication connective is not used in it, and negation is only applied to atomic formulas.

If x does not occur free in ψ, then the following equivalences hold.

$$\begin{array}{ll}
(\forall x.\phi) \land \psi \equiv \forall x.\phi \land \psi & \psi \land (\forall x.\phi) \equiv \forall x.\psi \land \phi \\
(\forall x.\phi) \lor \psi \equiv \forall x.\phi \lor \psi & \psi \lor (\forall x.\phi) \equiv \forall x.\psi \lor \phi \\
(\exists x.\phi) \land \psi \equiv \exists x.\phi \land \psi & \psi \land (\exists x.\phi) \equiv \exists x.\psi \land \phi \\
(\exists x.\phi) \lor \psi \equiv \exists x.\phi \lor \psi & \psi \lor (\exists x.\phi) \equiv \exists x.\psi \lor \phi
\end{array}$$

The applicability of these equivalences can always be assured by appropriate renaming of bound variables.

Normal forms

A formula is in *prenex form* if it is of the form $Q_1 x_1.Q_2 x_2.\ldots Q_n x_n.\psi$ where each Q_i is a quantifier (either $\forall$ or $\exists$) and ψ is a quantifier-free formula.

Prenex form of $\forall x.(\forall y.P(x,y) \lor Q(x)) \to \exists z.P(x,z)$

First we compute the NNF and then we go for the prenex form.

$$\begin{array}{lll}
 & \forall x.(\forall y.P(x,y) \lor Q(x)) \to \exists z.P(x,z) & \equiv \\
 & \forall x.\neg(\forall y.P(x,y) \lor Q(x)) \lor \exists z.P(x,z) & \equiv \\
\text{(NNF)} & \forall x.\exists y.(\neg P(x,y) \land \neg Q(x)) \lor \exists z.P(x,z) & \equiv \\
\text{(prenex)} & \forall x.\exists y.\exists z.(\neg P(x,y) \land \neg Q(x)) \lor P(x,z) &
\end{array}$$

Herbrand/Skolem normal forms

Let ϕ be a first-order formula in prenex normal form.

- The *Herbrandization* of ϕ (written ϕ^H) is an existential formula obtained from ϕ by repeatedly and exhaustively applying the following transformation:

$$\exists x_1,\ldots,x_n.\forall y.\psi \;\rightsquigarrow\; \exists x_1,\ldots,x_n.\psi[f(x_1,\ldots,x_n)/y]$$

  with f a fresh function symbol with arity n (i.e. f does not occur in ψ).
- The *Skolemization* of ϕ (written ϕ^S) is a universal formula obtained from ϕ by repeatedly applying the transformation:

$$\forall x_1,\ldots,x_n \exists y.\psi \;\rightsquigarrow\; \forall x_1,\ldots,x_n.\psi[f(x_1,\ldots,x_n)/y]$$

  with f a fresh function symbol with arity n.
- *Herbrand normal form* (resp. *Skolem normal form*) formulas are those obtained by the process of Herbrandization (resp. Skolemization).

Herbrandization/Skolemization

A formula ϕ and its Herbrandization/Skolemization are not logically equivalent.

Proposition

Let ϕ be a first-order formula in prenex normal form.

- ϕ is valid iff its Herbrandization ϕ^H is valid.
- ϕ is satisfiable iff its Skolemization ϕ^S is satisfiable.

Herbrandization/Skolemization change the underlying vocabulary. These additional symbols are called *Herbrand/Skolem functions*.

FOL with equality

There are different conventions for dealing with equality in first-order logic.

- We have follow the approach of considering equality predicate $(=)$ as a non-logical symbol, treated in the same way as any other predicate. We are working with what are usually known as *"first-order languages without equality"*.
- An alternative approach, usually called *"first-order logic with equality"*, considers equality as a logical symbol with a fixed interpretation.

  In this approach the equality symbol $(=)$ is interpreted as the equality relation in the domain of interpretation. So we have, for a structure $\mathcal{M} = (D, I)$ and an assignment $\alpha : \mathcal{X} \to D$, that

  $$\mathcal{M}, \alpha \models t_1 = t_2 \quad \text{iff} \quad \alpha_{\mathcal{M}}(t_1) \text{ and } \alpha_{\mathcal{M}}(t_2) \text{ are the same element of } D$$

FOL with equality

To understand the significant difference between having equality with the status of any other predicate, or with a fixed interpretation as in first-order logic with equality, consider the formulas

- $\exists x_1, x_2. \forall y.\ y = x_1 \lor y = x_2$

  With a fixed interpretation of equality, the validity of this formula implies that the cardinality of the interpretation domain is at most two – the quantifiers can actually be used to fix the cardinality of the domain, which is not otherwise possible in first-order logic.

- $\exists x_1, x_2. \neg(x_1 = x_2)$

  The validity of this formula implies that there exist at least two distinct elements in the domain, thus its cardinality must be at least two.

Many-sorted FOL

- A natural variant of first-order logic that can be considered is the one that results from allowing different domains of elements to coexist in the framework. This allows distinct "sorts" or types of objects to be distinguished at the syntactical level, constraining how operations and predicates interact with these different sorts.
- Having full support for different sorts of objects in the language allows for cleaner and more natural encodings of whatever we are interested in modeling and reasoning about.
- By adding to the formalism of FOL the notion of sort, we can obtain a flexible and convenient logic called *many-sorted first-order logic*, which has the same properties as FOL.

Many-sorted FOL

- A many-sorted vocabulary (signature) is composed of a set of sorts, a set of function symbols, and a set of predicate symbols.
  - Each function symbol f has associated with a type of the form $S_1 \times \ldots \times S_{\mathsf{ar}(f)} \to S$ where $S_1, \ldots, S_{\mathsf{ar}(f)}, S$ are sorts.
  - Each predicate symbol P has associated with it a type of the form $S_1 \times \ldots \times S_{\mathsf{ar}(P)}$.
  - Each variable is associated with a sort.
- The formation of terms and formulas is done only accordingly to the typing policy, i.e., respecting the "sorts".
- The domain of discourse of any structure of a many-sorted vocabulary is fragmented into different subsets, one for every sort.
- The notions of assignment and structure for a many-sorted vocabulary, and the interpretation of terms and formulas are defined in the expected way.

Modeling with FOL

Modeling with FOL

Being able to express an idea in FOL is an essential skill for Formal Methods in Software Engineering.

Use the predicates

admires(x, y) :	x admires y	Professor(x) :	x is a professor
attended(x, y) :	x attended y	Student(x) :	x is a student
		Lecture(x) :	x is a lecture

and the constant Mary to translate the following into predicate logic:

- Mary admires every professor.

$$\forall x.\, \mathsf{Professor}(x) \to \mathsf{admires}(\mathsf{Mary}, x)$$

- Some professor admires Mary.

$$\exists x.\, \mathsf{Professor}(x) \land \mathsf{admires}(x, \mathsf{Mary})$$

- Mary admires herself.

$$\mathsf{admires}(\mathsf{Mary}, \mathsf{Mary})$$

Modeling with FOL

Use the predicates

admires(x, y) :	x admires y	Professor(x) :	x is a professor
attended(x, y) :	x attended y	Student(x) :	x is a student
		Lecture(x) :	x is a lecture

and the constant Mary to translate the following into predicate logic:

- No student attended every lecture.

$$\neg(\exists x.\, \mathsf{Student}(x) \land (\forall y.\, \mathsf{Lecture}(y) \to \mathsf{attended}(x, y)))$$

- No lecture was attended by every student.

$$\neg(\exists x.\, \mathsf{Lecture}(x) \land (\forall y.\, \mathsf{Student}(y) \to \mathsf{attended}(y, x)))$$

- No lecture was attended by any student.

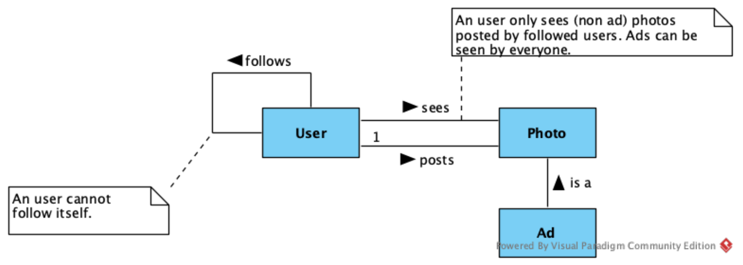
$$\neg(\exists l.\, \mathsf{Lecture}(l) \land \forall s.\, \mathsf{Student}(s) \to \neg\mathsf{attended}(s, l))$$

or equivalently

$$\forall l.\, \mathsf{Lecture}(l) \to \exists s.\, \mathsf{Student}(s) \land \mathsf{attended}(s, l)$$

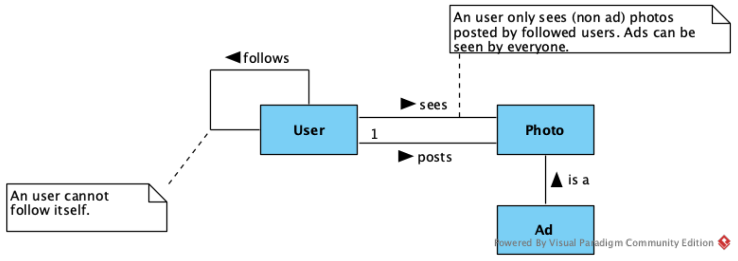
Domain models

- A **domain model** is a conceptual model of a system that describes the various entities involved in that system and their relationships.


An user only sees (non ad) photos posted by followed users. Ads can be seen by everyone.
◀ follows
▶ sees
User
1
Photo
▶ posts
▲ is a
An user cannot follow itself.
Ad
Powered By Visual Paradigm Community Edition

- It is used in software development as a first step to realize a new domain and resolve ambiguities in requirements.
- A domain model is a selective and structured representation of domain knowledge relevant to a given software development project.

Domain models


An user only sees (non ad) photos posted by followed users. Ads can be seen by everyone.
follows
sees
User
1
Photo
posts
is a
An user cannot follow itself.
Ad
Powered By Visual Paradigm Community Edition

A domain model is a diagram where

- entities are represented by boxes and
- relationships between entities by arcs annotated with
  - the name of the relationship/association,
  - the reading direction (indicated by an arrow) and
  - its multiplicity (annotated at the tip of the arcs).
- annotations with restrictions that are informally indicated in boxes may also appear in the diagram.

Domain models

- There is no standard notation for domain models.
- We are going to use the notation used in the *Desenvolvimento de Sistemas de Software* course. So, we assume that:
  - all entities present in a diagram are disjoint, unless between two entities there is a relation/association *"is a"* which denotes a specialization/extension;
  - if there are multiple specializations for the same entity, those specializations are disjoint;
  - the relation/association *"is a"* may be a subset or a membership relation. If it is a membership relation, the entity that "belongs to" another will be modeled as a constant.

Formalizing a domain model

We start by establish the logical language that we are going to use, i.e., the vocabulary of the language.

- a unary predicate for each entity (these predicates will act as the "type" of the entity, in a domain that is untyped);
- a predicate for each association, except specializations (which will be codified by formulas that establish the kind of specialization);
- a constant for each entity belonging to an "enumerated type".

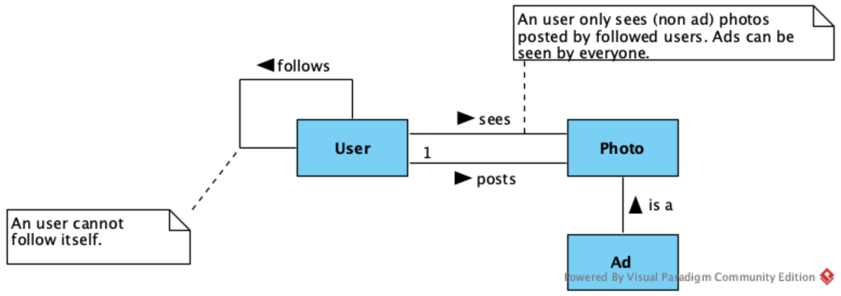
Formalizing a domain model

Next we write the set of formulas that describe the system. These formulas are of a different nature:

- codification of specialization relationships (which may be subset or membership relations, depending on the case);
- partitioning the universe of discourse with the types of the entities;
- disjunction of specialization entities;
- typing associations;
- multiplicity restrictions on associations;
- restrictions annotated informally.

Example: Instagram (simplified)

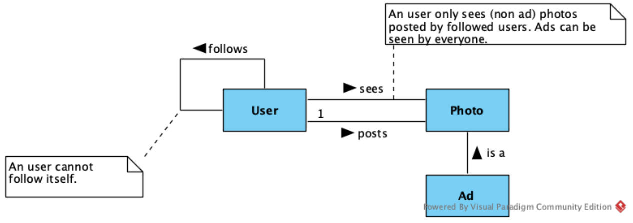
A simplified domain model for *Instagram*:


An user only sees (non ad) photos posted by followed users. Ads can be seen by everyone.
follows
sees
User
1
Photo
posts
is a
Ad
An user cannot follow itself.
Powered By Visual Paradigm Community Edition

Predicates:

User(-) Photo(-) Ad(-)
sees(-,-) posts(-,-) follows(-,-)

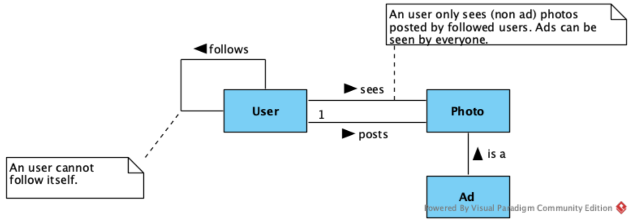
Example: Instagram (simplified)


An user only sees (non ad) photos posted by followed users. Ads can be seen by everyone.
follows
sees
User
1
Photo
posts
is a
Ad
An user cannot follow itself.
Powered By Visual Paradigm Community Edition

- Codification of specialization relationships (in this case, a subset relation).
  $\forall x.\, \mathsf{Ad}(x) \rightarrow \mathsf{Photo}(x)$
- Partitioning the universe of discourse with the types of the entities.
  $\forall x.\, \mathsf{User}(x) \leftrightarrow \neg\mathsf{Photo}(x)$

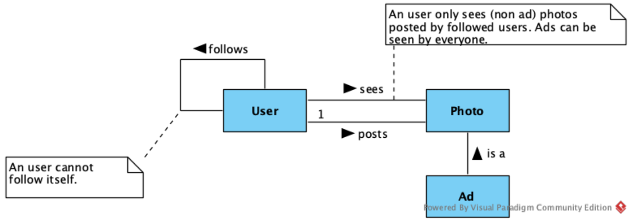
  Alternative: any element of the universe
  - at most is of one of these "types" $\forall x.\, \mathsf{User}(x) \rightarrow \neg\mathsf{Photo}(x)$
  - must be of one of these "types" $\forall x.\, \mathsf{User}(x) \vee \mathsf{Photo}(x)$

Example: Instagram (simplified)


An user only sees (non ad) photos posted by followed users. Ads can be seen by everyone.
follows
sees
User
1
Photo
posts
is a
Ad
An user cannot follow itself.
Powered By Visual Paradigm Community Edition

- Disjunction of specialization entities.
  (there is nothing to add)
- Typing associations.
  $\forall x, y.\, \mathsf{follows}(x,y) \rightarrow \mathsf{User}(x) \wedge \mathsf{User}(y)$
  $\forall x, y.\, \mathsf{sees}(x,y) \rightarrow \mathsf{User}(x) \wedge \mathsf{Photo}(y)$
  $\forall x, y.\, \mathsf{posts}(x,y) \rightarrow \mathsf{User}(x) \wedge \mathsf{Photo}(y)$

Example: Instagram (simplified)


An user only sees (non ad) photos posted by followed users. Ads can be seen by everyone.
follows
sees
User
1
Photo
posts
is a
Ad
An user cannot follow itself.
Powered By Visual Paradigm Community Edition

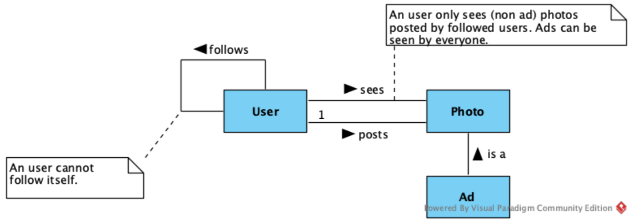
- Multiplicity restrictions on associations.

$(\forall y.\, \mathsf{Photo}(y) \rightarrow \exists x.\, \mathsf{posts}(x,y)) \wedge (\forall x,y,z.\, \mathsf{posts}(x,z) \wedge \mathsf{posts}(y,z) \rightarrow x = y)$

That is:
- every photo has to be posted by some user
  $\forall y.\, \mathsf{Photo}(y) \rightarrow \exists x.\, \mathsf{posts}(x,y)$
- every photo is posted by no more than one user
  $\forall x,y,z.\, \mathsf{posts}(x,z) \wedge \mathsf{posts}(y,z) \rightarrow x = y$

Therefore, every photo é posted by a single user.

Example: Instagram (simplified)


An user only sees (non ad) photos posted by followed users. Ads can be seen by everyone.
follows
User
sees
1
Photo
posts
is a
Ad
An user cannot follow itself.
Powered By Visual Paradigm Community Edition

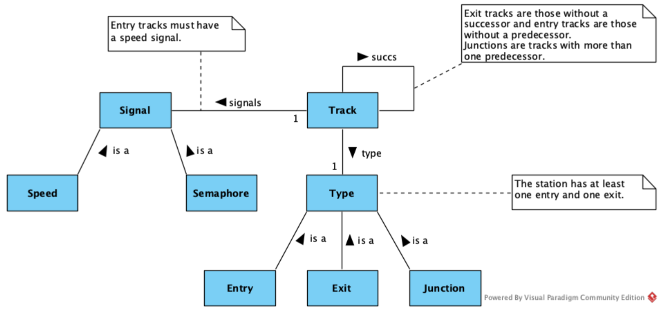
- Restrictions annotated informally.
  - An user cannot follow itself.

    $$\forall x.\, \neg \mathsf{follows}(x, x)$$
  - An user only sees (non ad) photos posted by followed users. Ads can be seen by everyone.

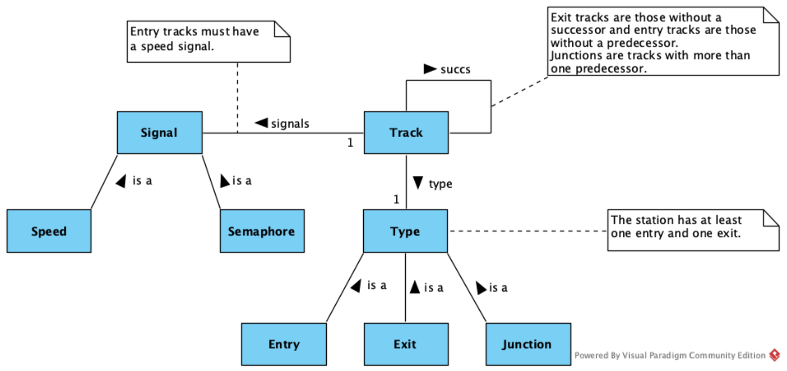
    $$\forall x, y.\, \mathsf{sees}(x, y) \to (\mathsf{Ad}(y) \lor (\forall z. \mathsf{posts}(z, y) \to \mathsf{follows}(x, z)))$$

Example: Train station

In simplistic terms, the layout of a train station is composed of tracks that are connected to each other. A track can have more than one successor, namely if it ends in a railway switch. A switch can also be used to form a junction between two or more tracks. Trains arrive at the station through entry tracks and leave the station through exit tracks. The interlocking system is responsible for ensuring the safe flow of the trains through a station. A key part of the interlocking are signals. Possible signals are semaphores and speed limits.


Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
1
Signal
Track
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

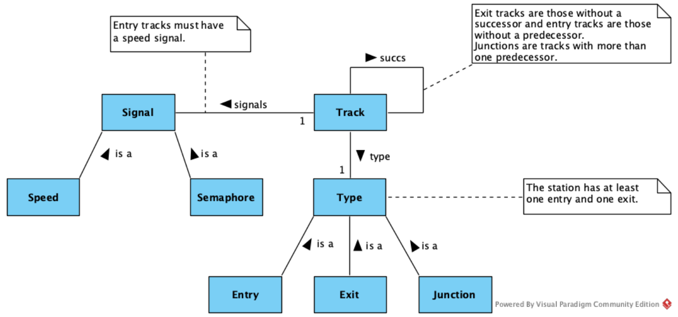
Example: Train station


Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
1
Signal
Track
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

Predicates: Signal(-) Speed(-) Semaphore(-) Track(-) Type(-)
signals(-,-) type(-,-) succs(-,-)

Constants: Entry Exit Junction

Example: Train station

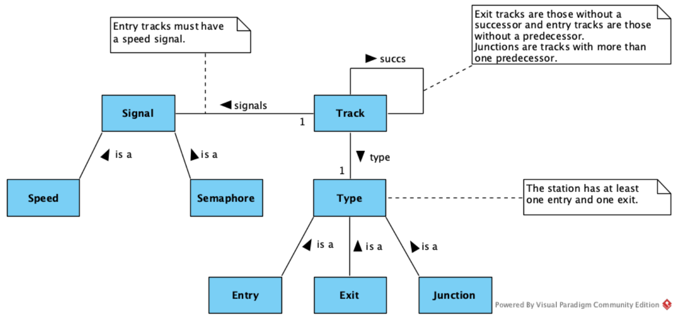

Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
1
Signal
Track
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

- Codification of specialization relationships.
  - The Signal entity has two subsets.

    $$\forall x.\, \mathsf{Speed}(x) \to \mathsf{Signal}(x)$$
    $$\forall x.\, \mathsf{Semaphore}(x) \to \mathsf{Signal}(x)$$
  - The Type entity is an “enumerated type” with at least 3 elements.

    $$\mathsf{Type}(\mathsf{Entry}),\quad \mathsf{Type}(\mathsf{Exit}),\quad \mathsf{Type}(\mathsf{Junction})$$

Example: Train station

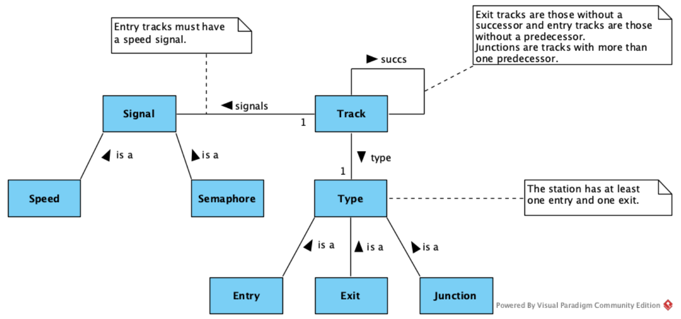


Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
1
Signal
Track
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

- Partitioning the universe of discourse with the types of the entities.

$$\forall x.\, \text{Signal}(x) \leftrightarrow \neg\text{Track}(x) \wedge \neg\text{Type}(x)$$
$$\forall x.\, \text{Track}(x) \leftrightarrow \neg\text{Signal}(x) \wedge \neg\text{Type}(x)$$
$$\forall x.\, \text{Type}(x) \leftrightarrow \neg\text{Track}(x) \wedge \neg\text{Signal}(x)$$

Example: Train station



Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
1
Signal
Track
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

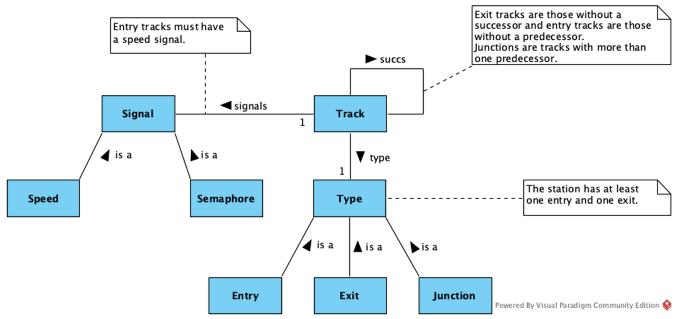
- Disjunction of specialization entities

$$\forall x.\, \text{Speed}(x) \rightarrow \neg\text{Semaphore}(x)$$
$$\text{Entry} \neq \text{Exit} \wedge \text{Entry} \neq \text{Junction} \wedge \text{Exit} \neq \text{Junction}$$

- Typing associations.

$$\forall x, y.\, \text{signals}(x, y) \rightarrow \text{Track}(x) \wedge \text{Signal}(y)$$
$$\forall x, y.\, \text{succs}(x, y) \rightarrow \text{Track}(x) \wedge \text{Track}(y)$$
$$\forall x, y.\, \text{type}(x, y) \rightarrow \text{Track}(x) \wedge \text{Type}(y)$$

Example: Train station



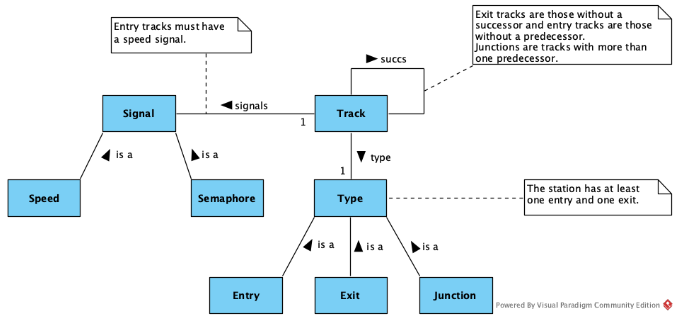
Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
1
Signal
Track
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

- Multiplicity restrictions on associations.

$$(\forall x.\, \text{Signal}(x) \rightarrow \exists y.\, \text{signals}(y, x)) \;\wedge$$
$$(\forall x, y, z.\, \text{signals}(x, z) \wedge \text{signals}(y, z) \rightarrow x = y)$$

$$(\forall x.\, \text{Track}(x) \rightarrow \exists y.\, \text{type}(x, y)) \;\wedge$$
$$(\forall x, y, z.\, \text{type}(x, z) \wedge \text{type}(x, y) \rightarrow z = y)$$

Example: Train station

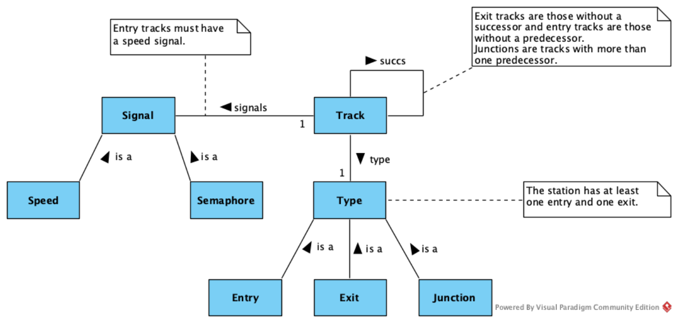


Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
1
Signal
Track
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

- Restrictions annotated informally.
  - Entry tracks must have a speed signal.

$$\forall x.\, \text{type}(x, \text{Entry}) \rightarrow \exists y.\, \text{signals}(x, y) \wedge \text{Speed}(y)$$

Example: Train station

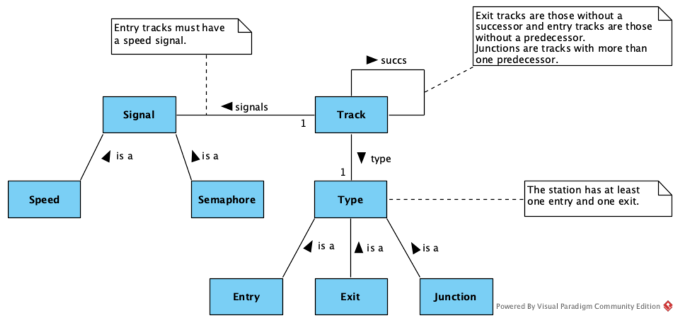


Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
Signal
Track
1
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

- Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.

$$\forall x.\, \mathsf{type}(x, \mathsf{Exit}) \leftrightarrow \forall y.\, \neg\mathsf{succs}(x, y)$$
$$\forall x.\, \mathsf{type}(x, \mathsf{Entry}) \leftrightarrow \neg\exists y.\, \mathsf{succs}(y, x)$$
$$\forall x.\, \mathsf{type}(x, \mathsf{Junction}) \leftrightarrow \exists y, z.\, \mathsf{succs}(y, x) \wedge \mathsf{succs}(z, x) \wedge y \neq z$$

Example: Train station



Entry tracks must have a speed signal.
Exit tracks are those without a successor and entry tracks are those without a predecessor. Junctions are tracks with more than one predecessor.
succs
signals
Signal
Track
1
is a
is a
type
1
Speed
Semaphore
Type
The station has at least one entry and one exit.
is a
is a
is a
Entry
Exit
Junction
Powered By Visual Paradigm Community Edition

- The station has at least one entry and one exit.

$$\exists x.\, \mathsf{type}(x, \mathsf{Entry})$$
$$\exists x.\, \mathsf{type}(x, \mathsf{Exit})$$