Fundamentals of Artificial Intelligence and Knowledge Representation (Module 2)

Last update: 24 December 2023

Academic Year 2023 – 2024
Alma Mater Studiorum · University of Bologna

Contents

1 Propositional logic

1.1 Syntax

Syntax Rules and symbols to define well-formed sentences. Syntax

The symbols of propositional logic are:

Proposition symbols p_0, p_1, ...

Connectives $\wedge \vee \rightarrow \leftrightarrow \neg \bot$ ( )

Well-formed formula The definition of a well-formed formula is recursive: Well-formed formula

- An atomic proposition is a well-formed formula.
- If S is well-formed, $\neg S$ is well-formed.
- If S_1 and S_2 are well-formed, $S_1 \wedge S_2$ is well-formed.
- If S_1 and S_2 are well-formed, $S_1 \vee S_2$ is well-formed.

Note that the implication $S_1 \rightarrow S_2$ can be written as $\neg S_1 \vee S_2$.

The BNF definition of a formula is:

$$F := \texttt{atomic\_proposition} \,|\, F \wedge F \,|\, F \vee F \,|\, F \rightarrow F \,|\, F \leftrightarrow F \,|\, \neg F \,|\, (F)$$

1.2 Semantics

Semantics Rules to associate a meaning to well-formed sentences. Semantics

Model theory What is true.

Proof theory What is provable.

Interpretation Given a propositional formula F of n atoms $\{A_1, \ldots, A_n\}$, an interpretation $\mathcal{I}$ of F is is a pair (D, I) where: Interpretation

- D is the domain. Truth values in the case of propositional logic.
- I is the interpretation mapping that assigns to the atoms $\{A_1, \ldots, A_n\}$ an element of D.

Note: given a formula F of n distinct atoms, there are 2^n district interpretations.

Model If F is true under the interpretation $\mathcal{I}$, we say that $\mathcal{I}$ is a model of F ($\mathcal{I} \models F$). Model

Valid formula A formula F is valid (tautology) iff it is true in all the possible interpretations. It is denoted as $\models F$. Valid formula

Invalid formula A formula F is invalid iff it is not valid (:o). Invalid formula

In other words, there is at least an interpretation where F is false.

Inconsistent formula A formula F is inconsistent (unsatisfiable) iff it is false in all the possible interpretations. Inconsistent formula

Consistent formula A formula F is consistent (satisfiable) iff it is not inconsistent. Consistent formula

In other words, there is at least an interpretation where F is true.

Decidability A logic is decidable if there is a terminating method to decide if a formula is valid. Decidability

Propositional logic is decidable.

Truth table Useful to define the semantics of connectives. Truth table

- $\neg S$ is true iff S is false.
- $S_1 \wedge S_2$ is true iff S_1 is true and S_2 is true.
- $S_1 \vee S_2$ is true iff S_1 is true or S_2 is true.
- $S_1 \rightarrow S_2$ is true iff S_1 is false or S_2 is true.
- $S_1 \leftrightarrow S_2$ is true iff $S_1 \rightarrow S_2$ is true and $S_1 \leftarrow S_2$ is true.

Evaluation The connectives of a propositional formula are evaluated in the order: Evaluation order

$$\leftrightarrow, \rightarrow, \vee, \wedge, \neg$$

Formulas in parenthesis have higher priority.

Logical consequence Let $\Gamma = \{F_1, \ldots, F_n\}$ be a set of formulas (premises) and G a formula (conclusion). G is a logical consequence of Γ ($\Gamma \models G$) if in all the possible interpretations $\mathcal{I}$, if $F_1 \wedge \cdots \wedge F_n$ is true, G is true. Logical consequence

Logical equivalence Two formulas F and G are logically equivalent ($F \equiv G$) iff the truth values of F and G are the same under the same interpretation. In other words, $F \equiv G \iff F \models G \wedge G \models F$. Logical equivalence

Common equivalences are:

Commutativity : $(P \wedge Q) \equiv (Q \wedge P)$ and $(P \vee Q) \equiv (Q \vee P)$

Associativity : $((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$ and $((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$

Double negation elimination : $\neg(\neg P) \equiv P$

Contraposition : $(P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P)$

Implication elimination : $(P \rightarrow Q) \equiv (\neg P \vee Q)$

Biconditional elimination : $(P \leftrightarrow Q) \equiv ((P \rightarrow Q) \wedge (Q \rightarrow P))$

De Morgan : $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$ and $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$

Distributivity of $\wedge$ over $\vee$: $(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$

Distributivity of $\vee$ over $\wedge$: $(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$

1.2.1 Normal forms

Negation normal form (NNF) A formula is in negation normal form iff negations appear only in front of atoms (i.e. not parenthesis). Negation normal form

Conjunctive normal form (CNF) A formula F is in conjunctive normal form iff: Conjunctive normal form

- it is in negation normal form;

- it has the form $F := F_1 \wedge F_2 \cdots \wedge F_n$, where each F_i (clause) is a disjunction of literals.

Example.
$(\neg P \vee Q) \wedge (\neg P \vee R)$ is in CNF.
$\neg(P \vee Q) \wedge (\neg P \vee R)$ is not in CNF (not in NNF).

Disjunctive normal form (DNF) A formula F is in disjunctive normal form iff: Disjunctive normal form

- it is in negation normal form;
- it has the form $F := F_1 \vee F_2 \cdots \vee F_n$, where each F_i is a conjunction of literals.

1.3 Reasoning

Reasoning method Systems to work with symbols. Reasoning method

Given a set of formulas Γ, a formula F and a reasoning method E, we denote with $\Gamma \vdash^E F$ the fact that F can be deduced from Γ using the reasoning method E.

Sound A reasoning method E is sound iff: Soundness

$$(\Gamma \vdash^E F) \rightarrow (\Gamma \models F)$$

Complete A reasoning method E is complete iff: Completeness

$$(\Gamma \models F) \rightarrow (\Gamma \vdash^E F)$$

Deduction theorem Given a set of formulas $\{F_1, \ldots, F_n\}$ and a formula G: Deduction theorem

$$(F_1 \wedge \cdots \wedge F_n) \models G \iff \models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$$

Proof.

$\rightarrow$) By hypothesis $(F_1 \wedge \cdots \wedge F_n) \models G$.

So, for each interpretation $\mathcal{I}$ in which $(F_1 \wedge \cdots \wedge F_n)$ is true, G is also true. Therefore, $\mathcal{I} \models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$.

Moreover, for each interpretation $\mathcal{I}'$ in which $(F_1 \wedge \cdots \wedge F_n)$ is false, $(F_1 \wedge \cdots \wedge F_n) \rightarrow G$ is true. Therefore, $\mathcal{I}' \models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$.

In conclusion, $\models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$.

$\leftarrow$) By hypothesis $\models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$. Therefore, for each interpretation where $(F_1 \wedge \cdots \wedge F_n)$ is true, G is also true.

In conclusion, $(F_1 \wedge \cdots \wedge F_n) \models G$.

□

Refutation theorem Given a set of formulas $\{F_1, \ldots, F_n\}$ and a formula G: Refutation theorem

$$(F_1 \wedge \cdots \wedge F_n) \models G \iff F_1 \wedge \cdots \wedge F_n \wedge \neg G \text{ is inconsistent}$$

Note: this theorem is not accepted in intuitionistic logic.

Proof. By definition, $(F_1 \wedge \cdots \wedge F_n) \models G$ iff for every interpretation where $(F_1 \wedge \cdots \wedge F_n)$ is true, G is also true. This requires that there are no interpretations where $(F_1 \wedge \cdots \wedge F_n)$ is true and G false. In other words, it requires that $(F_1 \wedge \cdots \wedge F_n \wedge \neg G)$ is inconsistent. □

1.3.1 Natural deduction

Proof theory Set of rules that allows to derive conclusions from premises by exploiting syntactic manipulations.

Proof theory

Natural deduction Set of rules to introduce or eliminate connectives. We consider a subset $\{\wedge, \rightarrow, \bot\}$ of functionally complete connectives.

Natural deduction for propositional logic

Natural deduction can be represented using a tree like structure:

$$\begin{array}{c} [\text{hypothesis}] \\ \vdots \\ \dfrac{\text{premise}}{\text{conclusion}}\ \text{rule name} \end{array}$$

The conclusion is true when the hypothesis are able to prove the premise. Another tree can be built on top of premises to prove them.

Introduction Usually used to prove the conclusion by splitting it.

Introduction rules

$$\frac{\psi \qquad \varphi}{\varphi \wedge \psi}\ \wedge\text{I} \qquad\qquad \begin{array}{c} [\varphi] \\ \vdots \\ \dfrac{\psi}{\varphi \rightarrow \psi} \end{array}\ \rightarrow\text{I}$$

Elimination Usually used to exploit hypothesis and derive a conclusion.

Elimination rules

$$\frac{\varphi \wedge \psi}{\varphi}\ \wedge\text{E} \qquad \frac{\varphi \wedge \psi}{\psi}\ \wedge\text{E} \qquad \frac{\varphi \qquad \varphi \rightarrow \psi}{\psi}\ \rightarrow\text{E}$$

Ex falso sequitur quodlibet From contradiction, anything follows. This can be used when we have two contradicting hypothesis.

Ex falso sequitur quodlibet

$$\frac{\bot}{\varphi}\ \bot$$

Reductio ad absurdum Assume the opposite and prove a contradiction (not accepted in intuitionistic logic).

Reductio ad absurdum

$$\begin{array}{c} [\neg\varphi] \\ \vdots \\ \dfrac{\bot}{\varphi}\ \text{RAA} \end{array}$$

2 First order logic

2.1 Syntax

Syntax

The symbols of propositional logic are:

Constants Known elements of the domain. Do not represent truth values.

Variables Unknown elements of the domain. Do not represent truth values.

Function symbols Function $f^{(n)}$ applied on n constants to obtain another constant.

Predicate symbols Function $P^{(n)}$ applied on n constants to obtain a truth value.

Connectives $\forall\ \exists\ \wedge\ \vee\ \rightarrow\ \neg\ \leftrightarrow\ \top\ \bot\ (\ )$

Using the basic syntax, the following constructs can be defined:

Term Denotes elements of the domain.

$$t := \texttt{constant} \mid \texttt{variable} \mid f^{(n)}(t_1, \ldots, t_n)$$

Proposition Denotes truth values.

$$P := \top \mid \bot \mid P \wedge P \mid P \vee P \mid P \rightarrow P \mid P \leftrightarrow P \mid \neg P \mid \forall x.P \mid \exists x.P \mid (P) \mid P^{(n)}(t_1, \ldots, t_n)$$

Well-formed formula

Well-formed formula The definition of well-formed formula in first order logic extends the one of propositional logic by adding the following conditions:

- If S is well-formed, $\exists X.S$ is well-formed. Where X is a variable.
- If S is well-formed, $\forall X.S$ is well-formed. Where X is a variable.

Free variables

Free variables The universal and existential quantifiers bind their variable within the scope of the formula. Let $F_v(F)$ be the set of free variables in a formula F, F_v is defined as follows:

- $F_v(p(t)) = \bigcup \texttt{vars}(t)$
- $F_v(\top) = F_v(\bot) = \varnothing$
- $F_v(\neg F) = F_v(F)$
- $F_v(F_1 \wedge F_2) = F_v(F_1 \vee F_2) = F_v(F_1 \rightarrow F_2) = F_v(F_1) \cup F_v(F_2)$
- $F_v(\forall X.F) = F_v(\exists X.F) = F_v(F) \setminus \{X\}$

Sentence

Closed formula/Sentence Proposition without free variables.

Theory

Theory Set of sentences.

Formula

Ground term/Formula Proposition without variables.

2.2 Semantics

Interpretation An interpretation in first order logic $\mathcal{I}$ is a pair (D, I): Interpretation

- D is the domain of the terms.
- I is the interpretation function such that:
  - $I(f) : D^n \to D$ for every n-ary function symbol.
  - $I(p) \subseteq D^n$ for every n-ary predicate symbol.

Variable evaluation Given an interpretation $\mathcal{I} = (D, I)$ and a set of variables $\mathcal{V}$, a variable is evaluated through $\eta : \mathcal{V} \to D$. Variable evaluation

Model Given an interpretation $\mathcal{I}$ and a formula F, $\mathcal{I}$ models F ($\mathcal{I} \models F$) when $\mathcal{I}, \eta \models F$ for every variable evaluation η. Model

A sentence S is:

Valid S is satisfied by every interpretation ($\forall \mathcal{I} : \mathcal{I} \models S$).

Satisfiable S is satisfied by some interpretations ($\exists \mathcal{I} : \mathcal{I} \models S$).

Falsifiable S is not satisfied by some interpretations ($\exists \mathcal{I} : \mathcal{I} \not\models S$).

Unsatisfiable S is not satisfied by any interpretation ($\forall \mathcal{I} : \mathcal{I} \not\models S$).

Logical consequence A sentence T_1 is a logical consequence of T_2 ($T_2 \models T_1$) if every model of T_2 is also model of T_1: Logical consequence

$$\mathcal{I} \models T_2 \to \mathcal{I} \models T_1$$

Theorem 2.2.1. It is undecidable to determine if a first order logic formula is a tautology.

Equivalence A sentence T_1 is equivalent to T_2 if $T_1 \models T_2$ and $T_2 \models T_1$. Equivalence

Theorem 2.2.2. The following statements are equivalent:

1. $F_1, \ldots, F_n \models G$.
2. $(\bigwedge_{i=1}^{n} F_i) \to G$ is valid.
3. $(\bigwedge_{i=1}^{n} F_i) \wedge \neg G$ is unsatisfiable.

2.3 Substitution

Substitution A substitution $\sigma : \mathcal{V} \to \mathcal{T}$ is a mapping from variables to terms. It is written as $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$. Substitution

The application of a substitution is the following:

- $p(t_1, \ldots, t_n)\sigma = p(t_1\sigma, \ldots, t_n\sigma)$
- $f(t_1, \ldots, t_n)\sigma = fp(t_1\sigma, \ldots, t_n\sigma)$
- $\bot\sigma = \bot$ and $\top\sigma = \top$
- $(\neg F)\sigma = (\neg F\sigma)$
- $(F_1 \star F_2)\sigma = (F_1\sigma \star F_2\sigma)$ for $\star \in \{\wedge, \vee, \to\}$

- $(\forall X.F)\sigma = \forall X'(F\sigma[X \mapsto X'])$ where X' is a fresh variable (i.e. does not appear in F).
- $(\exists X.F)\sigma = \exists X'(F\sigma[X \mapsto X'])$ where X' is a fresh variable.

Unifier A substitution σ is a unifier for $e_1, \ldots, e_n$ if $e_1\sigma = \cdots = e_n\sigma$. Unifier

Most general unifier A unifier σ is the most general unifier (MGU) for $\bar{e} = e_1, \ldots, e_n$ if every unifier τ for $\bar{e}$ is an instance of σ ($\tau = \sigma\rho$ for some substitution ρ). In other words, σ is the smallest substitution to unify $\bar{e}$. Most general unifier

3 Prolog

It may be useful to first have a look at the "Logic programming" section of `Languages and Algorithms for AI (module 2)`.

3.1 Syntax

Term Following the first-order logic definition, a term can be a: Term

- Constant (`lowerCase`).
- Variable (`UpperCase`).
- Function symbol (`f(t1, ..., tn)` with `t1`, ..., `tn` terms).

Atomic formula An atomic formula has form: Atomic formula

`p(t1, ..., tn)`

where `p` is a predicate symbol and `t1`, ..., `tn` are terms.

Note: there are no syntactic distinctions between constants, functions and predicates.

Clause A Prolog program is a set of horn clauses: Horn clause

Fact `A.`

Rule `A :- B1, ..., Bn.` (`A` is the head and `B1, ..., Bn` the body)

Goal `:- B1, ..., Bn.`

where:

- `A`, `B1`, ..., `Bn` are atomic formulas.
- `,` represents the conjunction ($\wedge$).
- `:-` represents the logical implication ($\Leftarrow$).

Quantification Quantification

Facts Variables appearing in a fact are quantified universally.

$$\texttt{A(X).} \equiv \forall \texttt{X} : \texttt{A(X)}$$

Rules Variables appearing the the body only are quantified existentially. Variables appearing in both the head and the body are quantified universally.

$$\texttt{A(X) :- B(X, Y).} \equiv \forall \texttt{X}, \exists \texttt{Y} : \texttt{A(X)} \Leftarrow \texttt{B(X, Y)}$$

Goals Variables are quantified existentially.

$$\texttt{:- B(Y).} \equiv \exists \texttt{Y} : \texttt{B(Y)}$$

3.2 Semantics

Execution of a program A computation in Prolog attempts to prove the goal. Given a program P and a goal `:- p(t1, ..., tn)`, the objective is to find a substitution σ such that:

$$P \models [\texttt{p(t1, ..., tn)}]\sigma$$

In practice, it uses two stacks:

Execution stack Contains the predicates the interpreter is trying to prove.

Backtracking stack Contains the choice points (clauses) the interpreter can try.

SLD resolution Prolog uses SLD resolution with the following choices: SLD

Left-most Always proves the left-most literal first.

Depth-first Applies the predicates following the order of definition.

Note that the depth-first approach can be efficiently implemented (tail recursion) but the termination of a Prolog program on a provable goal is not guaranteed as it may loop depending on the ordering of the clauses.

Disjunction operator The operator `;` can be seen as a disjunction and makes the Prolog interpreter explore the remaining SLD tree looking for alternative solutions.

3.3 Arithmetic operators

In Prolog: Arithmetic operators

- Integers and floating points are built-in atoms.
- Math operators are built-in function symbols.

Therefore, mathematical expressions are terms.

`is` predicate The predicate `is` is used to evaluate and unify expressions:

```
T is Expr
```

where `T` is a numerical atom or a variable and `Expr` is an expression without free variables. After evaluation, the result of `Expr` is unified with `T`.

Example.

```
?- X is 2+3.
    yes X=5
```

Note: a term representing an expression is evaluated only with the predicate `is` (otherwise it remains as is).

Relational operators Relational operators (`>`, `<`, `>=`, `=<`, `==`, `=/=`) are built-in.

3.4 Lists

Lists

A list is defined recursively as:

Empty list `[]`

List constructor `.(T, L)` where `T` is a term and `L` is a list.

Note that a list always ends with an empty list.
As the formal definition is impractical, some syntactic sugar has been defined:

List definition `[t1, ..., tn]` can be used to define a list.

Head and tail `[H | T]` where `H` is the head (term) and `T` the tail (list) can be useful for recursive calls.

3.5 Cut

Cut

The cut operator (`!`) allows to control the exploration of the SLD tree.
A cut in a clause:

```
p :- q1, ..., qi, !, qj, ..., qn.
```

makes the interpreter consider only the first choice points for `q1, ..., qi`, dropping all the other possibilities. Therefore, if `qj, ..., qn` fails, there won't be backtracking and `p` fails.

Example.

```
p(X) :- q(X), r(X).
q(1).
q(2).
r(2).

?- p(X).
    yes X=2
```

```
p(X) :- q(X), !, r(X).
q(1).
q(2).
r(2).

?- p(X).
    no
```

In the second case, the cut drops the choice point `q(2)` and only considers `q(1)`.

Mutual exclusion A cut can be useful to achieve mutual exclusion. In other words, to represent a conditional branching:

```
if a(X) then b else c
```

a cut can be used as follows:

```
p(X) :- a(X), !, b.
p(X) :- c.
```

If `a(X)` succeeds, other choice points for `p` will be dropped and only `b` will be evaluated. If `a(X)` fails, the second clause will be considered, therefore evaluating `c`.

3.6 Negation

Closed-world assumption Only what is stated in a program P is true, everything else is false: Closed-world assumption

$$\texttt{CWA}(P) = P \cup \{\neg A \mid A \text{ is a ground atomic formula and } P \not\models A\}$$

Non-monotonic inference rule Adding new axioms to the program may change the set of valid theorems.

As first-order logic in undecidable, closed-world assumption cannot be directly applied in practice.

Negation as failure A negated atom $\neg A$ is considered true iff A fails in finite time: Negation as failure

$$\texttt{NF}(P) = P \cup \{\neg A \mid A \in \texttt{FF}(P)\}$$

where $\texttt{FF}(P) = \{B \mid P \not\models B \text{ in finite time}\}$ is the set of atoms for which the proof fails in finite time. Note that not all atoms B such that $P \not\models B$ are in $\texttt{FF}(P)$.

SLDNF SLD resolution with NF to solve negative atoms. SLDNF

Given a goal of literals `:-` $\texttt{L}_1$`,` `...,` $\texttt{L}_m$, SLDNF does the following:

1. Select a positive or ground negative literal $\texttt{L}_i$:
   - If $\texttt{L}_i$ is positive, apply the normal SLD resolution.
   - If $\texttt{L}_i = \neg A$, prove that A fails in finite time. If it succeeds, $\texttt{L}_i$ fails.
2. Solve the goal `:-` $\texttt{L}_1$`,` `...,` $\texttt{L}_{i-1}$`,` $\texttt{L}_{i+1}$`,` `...`$\texttt{L}_m$.

Theorem 3.6.1. If only positive or ground negative literal are selected during resolution, SLDNF is correct and complete.

Prolog SLDNF Prolog uses an incorrect implementation of SLDNF where the selection rule always chooses the left-most literal. This potentially causes incorrect deductions.

Proof. When proving `:- \+capital(X).`, the intended meaning is:

$$\exists \texttt{X} : \neg \texttt{capital(X)}$$

In SLDNF, to prove `:- \+capital(X).`, the algorithm proves `:- capital(X).`, which results in:

$$\exists \texttt{X} : \texttt{capital(X)}$$

and then negates the result, which corresponds to:

$$\neg(\exists \texttt{X} : \texttt{capital(X)}) \iff \forall \texttt{X} : (\neg \texttt{capital(X)})$$

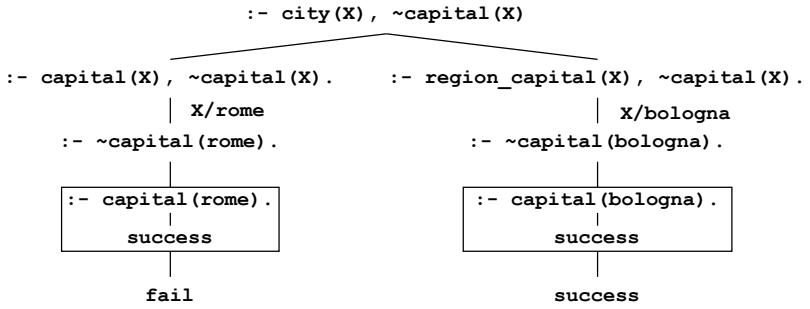
□

Example (Correct SLDNF resolution)**.** Given the program:

```
capital(rome).
region_capital(bologna).
city(X) :- capital(X).
city(X) :- region_capital(X).

?- city(X), \+capital(X).
```

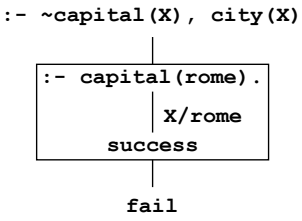
its resolution succeeds with `X=bologna` as `\+capital(X)` is ground by the unification of `city(X)`.



Example (Incorrect SLDNF resolution)**.** Given the program:

```
capital(rome).
region_capital(bologna).
city(X) :- capital(X).
city(X) :- region_capital(X).

?- \+capital(X), city(X).
```



its resolution fails as `\+capital(X)` is a free variable and the proof of `capital(X)` is ground with `X=rome` and succeeds, therefore failing `\+capital(X)`. Note that `bologna` is not tried as it does not appear in the axioms of `capital`.

3.7 Meta predicates

`call/1` Given a term `T`, `call(T)` considers `T` as a predicate and evaluates it. At the time of evaluation, `T` must be a non-numeric term. call/1

Example.

```
p(X) :- call(X).
q(a).

?- p(q(Y)).
    yes Y=a
```

`fail/0` The evaluation of `fail` always fails, forcing the interpreter to backtrack. fail/0

Example (Implementation of negation as failure)**.**

```
not(P) :- call(P), !, fail.
not(P).
```

Note that the cut followed by `fail` (`!, fail`) is useful to force a global failure.

`bagof/3` and `setof/3`

`bagof/3` The predicate `bagof(X, P, L)` unifies `L` with a list of the instances of `X` that satisfy `P`. Fails if none exists. bagof/3

`sefof/3` The predicate `setof(X, P, S)` unifies `S` with a set of the instances of `X` that satisfy `P`. Fails if none exists. sefof/3

In practice, for computational reasons, a list (with repetitions) might be computed.

Example.

```
p(1).
p(2).
p(1).

?- setof(X, p(X), S).
    yes S=[1, 2] X=X

?- bagof(X, p(X), S).
    yes S=[1, 2, 1] X=X
```

Quantification When solving a goal, the interpreter unifies free variables with a value. This may cause unwanted behaviors when using `bagof` or `setof`. The `X^` tells the interpreter to not (permanently) bind the variable `X`.

Example.

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- setof(X, father(X, Y), S).
    yes X=X Y=giuseppe S=[giovanni];
        X=X Y=mario    S=[giovanni];
        X=X Y=paola    S=[mario]
```

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- setof(X, Y^father(X, Y), S).
    yes S=[giovanni, mario] X=X Y=Y
```

`findall/3` The predicate `findall(X, P, S)` unifies `S` with a list of the instances of `X` that satisfy `P`. If none exists, `S` is unified with an empty list. Variables in `P` that do not appear in `X` are not bound (same as the `Y^` operator). findall/3

Example.

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- findall(X, father(X, Y), S).
    yes S=[giovanni, mario] X=X Y=Y
```

`var/1` The predicate `var(T)` is true if `T` is a variable. var/1

`nonvar/1` The predicate `nonvar(T)` is true if `T` is not a free variable. nonvar/1

`number/1` The predicate `number(T)` is true if `T` is a number. number/1

`ground/1` The predicate `ground(T)` is true if `T` does not have free variables. ground/1

`=../2` The operator `T =.. L` unifies `L` with a list where its head is the head of `T` and the tail contains the remaining arguments of `T` (i.e. puts all the components of a predicate into a list). Only one between `T` and `L` may be a variable. =../2

Example.

```
?- foo(hello, X) =.. List.
    List = [foo, hello, X]
```

```
?- Term =.. [baz, foo(1)].
    Term = baz(foo(1))
```

`clause/2` The predicate `clause(Head, Body)` is true if it can unify `Head` and `Body` with an existing clause. `Head` must be initialized to a non-numeric term. `Body` can be a variable or a term. clause/2

Example.

```
p(1).
q(X, a) :- p(X), r(a).
q(2, Y) :- d(Y).

?- clause(p(1), B).
    yes B=true

?- clause(p(X), true).
    yes X=1

?- clause(q(X, Y), B).
    yes X=_1 Y=a B=p(_1), r(a);
        X=2 Y=_2 B=d(_2)
```

`assert/1` The predicate `assert(T)` adds `T` in an unspecified position of the clauses database of Prolog. In other words, it allows to dynamically add clauses. assert/1

`asserta/1` As `assert(T)`, with insertion at the beginning of the database. asserta/1

`assertz/1` As `assert(T)`, with insertion at the end of the database. assertz/1

Note that `:- assert((p(X)))` quantifies `X` existentially as it is a query. If it is not ground and added to the database as is, is becomes a clause and therefore quantified universally: $\forall$`X` : `p(X)`.

Example (Lemma generation)**.**

```
fib(0, 0) :- !.
fib(1, 1) :- !.
fib(N, F) :- N1 is N-1, fib(N1, F1),
             N2 is N-2, fib(N2, F2),
             F is F1+F2,
             generate_lemma(fib(N, F)).

generate_lemma(T) :- clause(T, true), !.
generate_lemma(T) :- assert(T).
```

`generate_lemma/1` allows to add to the clauses database all the intermediate steps to compute the Fibonacci sequence (similar concept to dynamic programming).

`retract/1` The predicate `retract(T)` removes from the database the first clause that unifies with `T`. retract/1

`abolish/2` The predicate `abolish(T, n)` removes from the database all the occurrences of `T` with arity `n`. abolish/2

3.8 Meta-interpreters

Meta-interpreter Interpreter for a language L_1 written in another language L_2. Meta-interpreter

Prolog vanilla meta-interpreter The Prolog vanilla meta-interpreter is defined as follows: Vanilla meta-interpreter

```
solve(true) :- !.
solve( (A, B) ) :- !, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

In other words, the clauses state the following:

1. A tautology is a success.
2. To prove a conjunction, we have to prove both atoms.
3. To prove an atom `A`, we look for a clause `A :- B` that has `A` as conclusion and prove its premise `B`.

4 Ontologies

Ontology Formal (non-ambiguous) and explicit (obtainable through a finite sound procedure) description of a domain. — Ontology

Category Can be organized hierarchically on different levels of generality. — Category

Object Belongs to one or more categories. — Object

Upper/general ontology Ontology focused on the most general domain. — Upper/general ontology

Properties:

- Should be applicable to almost any special domain.
- Combining general concepts should not incur in inconsistences.

Approaches to create ontologies:

- Created by philosophers/logicians/researchers.
- Automatic knowledge extraction from well-structured databases.
- Created from text documents (e.g. web).
- Crowd-sharing information.

4.1 Categories

Category Used in human reasoning when the goal is category-driven (in contrast to specific-instance-driven). — Category

In first order logic, categories can be represented through:

Predicate A predicate to tell if an object belongs to a category (e.g. `Car(c1)` indicates that `c1` is a car). — Predicate categories

Reification Represent categories as objects as well (e.g. $\texttt{c1} \in \texttt{Car}$). — Reification

4.1.1 Reification properties and operations

Membership Indicates if an object belongs to a category. (e.g. $\texttt{c1} \in \texttt{Car}$). — Membership

Subclass Indicates if a category is a subcategory of another one. (e.g. $\texttt{Car} \subset \texttt{Vehicle}$). — Subclass

Necessity Members of a category enjoy some properties (e.g. $(\texttt{x} \in \texttt{Car}) \Rightarrow \texttt{hasWheels(x)}$). — Necessity

Sufficiency Sufficient conditions to be part of a category (e.g. $\texttt{hasPlate(x)} \wedge \texttt{hasWheels(x)} \Rightarrow \texttt{x} \in \texttt{Car}$). — Sufficiency

Category-level properties Category themselves can enjoy properties (e.g. $\texttt{Car} \in \texttt{VehicleType}$) — Category-level properties

Disjointness Given a set of categories S, the categories in S are disjoint iff they all have different objects: Disjointness

$$\texttt{disjoint}(S) \iff (\forall c_1, c_2 \in S, c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset)$$

Exhaustive decomposition Given a category c and a set of categories S, S is an exhaustive decomposition of c iff any element in c belongs to at least a category in S: Exhaustive decomposition

$$\texttt{exhaustiveDecomposition}(S,\ c) \iff (\forall o \in c \iff \exists c_2 \in S : o \in c_2)$$

Partition Given a category c and a set of categories S, S is a partition of c when: Partition

$$\texttt{partition}(S,\ c) \iff \texttt{disjoint}(S) \wedge \texttt{exhaustiveDecomposition}(S,\ c)$$

4.1.2 Physical composition

Objects (meronyms) are part of a whole (holonym).

Part-of If the objects have a structural relation (e.g. `partOf(cylinder1, engine1)`). Part-of

Properties:

Transitivity $\texttt{partOf(x, y)} \wedge \texttt{partOf(y, z)} \Rightarrow \texttt{partOf(x, z)}$

Reflexivity `partOf(x, x)`

Bunch-of If the objects do not have a structural relation. Useful to define a composition of countable objects (e.g. `bunchOf(nail1, nail3, nail4)`). Bunch-of

4.1.3 Measures

A property of objects.

Quantitative measure Something that can be measured using a unit (e.g. $\texttt{length(table1)} = \texttt{cm(80)}$). Quantitative measure

Qualitative measures propagate when using `partOf` or `bunchOf` (e.g. the weight of a car is the sum of its parts).

Qualitative measure Something that can be measured using terms with a partial or total order relation (e.g. $\{\texttt{good}, \texttt{neutral}, \texttt{bad}\}$). Qualitative measure

Qualitative measures do not propagate when using `partOf` or `bunchOf`.

Fuzzy logic Provides a semantics to qualitative measures (i.e. convert qualitative to quantitative). Fuzzy logic

4.1.4 Things vs stuff

Intrinsic property Related to the substance of the object. It is retained when the object is divided (e.g. water boils at 100°C). Intrinsic property

Extrinsic property Related to the structure of the object. It is not retained when the object is divided (e.g. the weight of an object changes when split). Extrinsic property

Substance Category of objects with only intrinsic properties. Substance

Stuff The most general substance category. Stuff

Count noun Category of objects with only extrinsic properties. Count noun

Things The most general object category. Things

4.2 Semantic networks

Graphical representation of objects and categories connected through labelled links. Semantic networks

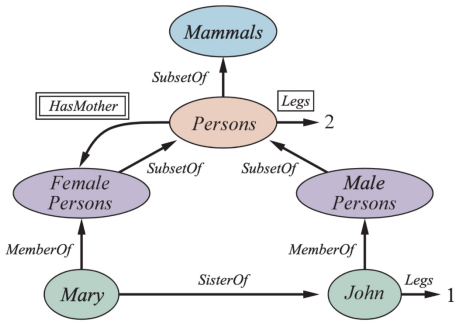


Figure 4.1: Example of semantic network

Objects and categories Represented using the same symbol.

Links Four different types of links:

- Relation between objects (e.g. `SisterOf`).
- Property of a category (e.g. 2 `Legs`).
- Is-a relation (e.g. `SubsetOf`).
- Property of the members of a category (e.g. `HasMother`).

Single inheritance reasoning Starting from an object, check if it has the queried property. If not, iteratively move up to the category it belongs to and check for the property. Single inheritance reasoning

Multiple inheritance reasoning Reasoning is not possible as it is not clear which parent to choose. Multiple inheritance reasoning

Limitations Compared to first order logic, semantic networks do not have:

- Negations.
- Universally and existentially quantified properties.
- Disjunctions.
- Nested function symbols.

Many semantic network systems allow to attach special procedures to handle special cases that the standard inference algorithm cannot handle. This approach is powerful but does not have a corresponding logical meaning.

Advantages With semantic networks it is easy to attach default properties to categories and override them on the objects (i.e. `Legs` of `John`).

4.3 Frames

Knowledge that describes an object in terms of its properties. Each frame has: Frames

- An unique name
- Properties represented as pairs `<slot - filler>`

Example.

```
(
    toronto
        <:Instance-Of City>
        <:Province ontario>
        <:Population 4.5M>
)
```

Prototype Members of a category used as comparison metric to determine if another object belongs to the same class (i.e. an object belongs to a category if it is similar enough to the prototypes of that category). Prototype

Defeasible value Value that is allowed to be different when comparing an object to a prototype. Defeasible value

Facets Additional information contained in a slot for its filler (e.g. default value, type, domain). Facets

Procedural information Fillers can be a procedure that can be activated by specific facets:

`if-needed` Looks for the value of the slot.

`if-added` Adds a value.

`if-removed` Removes a value.

Example.

```
(
    toronto
        <:Instance-Of City>
        <:Province ontario>
        <:Population [if-needed QueryDB]>
)
```

5 Description logic

5.1 Syntax

Logical symbols Symbols with fixed meaning. Logical symbols

Punctuation ( ) [ ]

Positive integers

Concept-forming operators `ALL`, `EXISTS`, `FILLS`, `AND`

Connectives $\sqsubseteq$, $\doteq$, $\rightarrow$

Non-logical symbols Domain-dependant symbols. Non-logical symbols

Atomic concepts Categories (CamelCase, e.g. `Person`).

Roles Used to describe objects (:CamelCase, e.g. `:Height`).

Constants (camelCase, e.g. `johnDoe`).

Complex concept Concept-forming operators can be used to combine atomic concepts and form complex concepts. A well-formed concept follows the conditions: Complex concept

- An atomic concept is a concept.
- If `r` is a role and `d` is a concept, then `[ALL r d]` is a concept.
- If `r` is a role and n is a positive integer, then `[EXISTS` n `r]` is a concept.
- If `r` is a role and `c` is a constant, then `[FILLS r c]` is a concept.
- If $\texttt{d}_1 \ldots \texttt{d}_n$ are concepts, then `[AND` $\texttt{d}_1 \ldots \texttt{d}_n$`]` is a concept.

Sentence Connectives can be used to combine concepts and form sentences. A well-formed sentence follows the conditions: Sentence

- If $\texttt{d}_1$ and $\texttt{d}_2$ are concepts, then $(\texttt{d}_1 \sqsubseteq \texttt{d}_2)$ is a sentence.
- If $\texttt{d}_1$ and $\texttt{d}_2$ are concepts, then $(\texttt{d}_1 \doteq \texttt{d}_2)$ is a sentence.
- If `c` is a constant and `d` is a concept, then $(\texttt{c} \rightarrow \texttt{d})$ is a sentence.

Knowledge base Collection of sentences. Knowledge base

Constants are individuals of the domain.

Concepts are categories of individuals.

Roles are binary relations between individuals.

Assetion box (A-box) List of facts about individuals. Assetion box (A-box)

Terminological box (T-box) List of sentences (axioms) about concepts. Terminological box (T-box)

5.2 Semantics

5.2.1 Concept-forming operators

Concept-forming operators

Let `r` be a role, `d` be a concept, `c` be a constant and n a positive integer. The semantics of concept-forming operators are:

`[ALL r d]` Individuals `r`-related to the individuals of the category `d`.

Example. `[ALL :HasChild Male]` individuals that have zero children or only male children.

`[EXISTS` n `r]` Individuals `r`-related to at least n other individuals.

Example. `[EXISTS 1 :Child]` individuals with at least one child.

`[FILLS r c]` Individuals `r`-related to the individual `c`.

Example. `[FILLS :Child john]` individuals with child `john`.

`[AND` $\mathtt{d}_1 \dots \mathtt{d}_n$`]` Individuals belonging to all the categories $\mathtt{d}_1 \dots \mathtt{d}_n$.

5.2.2 Sentences

Sentences

Sentences are expressions with truth values in the domain. Let `d` be a concept and `c` be a constant. The semantics of sentences are:

$\mathtt{d}_1 \sqsubseteq \mathtt{d}_2$ Concept $\mathtt{d}_1$ is subsumed by $\mathtt{d}_2$.

Example. `PhDStudent` $\sqsubseteq$ `Student` as every PhD is also a student.

$\mathtt{d}_1 \doteq \mathtt{d}_2$ Concept $\mathtt{d}_1$ is equivalent to $\mathtt{d}_2$.

Example. `PhDStudent` $\doteq$ `[AND Student :Graduated :HasFunding]`

$\mathtt{c} \to \mathtt{d}$ The individual `c` satisfies the description of the concept `d`.

Example. `federico` $\to$ `Professor`

5.2.3 Interpretation

Interpretation

Interpretation An interpretation $\mathfrak{I}$ in description logic is a pair $(\mathcal{D}, \mathcal{I})$ where:

- $\mathcal{D}$ is the domain.
- $\mathcal{I}$ is the interpretation mapping.

  Constant Let `c` be a constant, $\mathcal{I}[\mathtt{c}] \in \mathcal{D}$.

  Atomic concept Let `a` be an atomic concept, $\mathcal{I}[\mathtt{a}] \subseteq \mathcal{D}$.

  Role Let `r` be a role, $\mathcal{I}[\mathtt{r}] \subseteq \mathcal{D} \times \mathcal{D}$.

  `Thing` The concept `Thing` corresponds to the domain: $\mathcal{I}[\mathtt{Thing}] = \mathcal{D}$.

  `[ALL r d]`

  $$\mathcal{I}[\mathtt{[ALL\ r\ d]}] = \{\mathtt{x} \in \mathcal{D} \mid \forall \mathtt{y} : \langle \mathtt{x}, \mathtt{y} \rangle \in \mathcal{I}[r] \text{ then } \mathtt{y} \in \mathcal{I}[d]\}$$

  `[EXISTS` n `r]`

  $$\mathcal{I}[\mathtt{[EXISTS}\ n\ \mathtt{r]}] = \{\mathtt{x} \in \mathcal{D} \mid \text{ exists at least } n \text{ distinct } \mathtt{y} : \langle \mathtt{x}, \mathtt{y} \rangle \in \mathcal{I}[r]\}$$

`[FILLS r c]`

$$\mathcal{I}[\texttt{[FILLS r c]}] = \{\texttt{x} \in \mathcal{D} \mid \langle \texttt{x}, \mathcal{I}[\texttt{c}] \rangle \in \mathcal{I}[\texttt{r}]\}$$

`[AND` $\texttt{d}_1 \ldots \texttt{d}_n$`]`

$$\mathcal{I}[\texttt{[AND } \texttt{d}_1 \ldots \texttt{d}_n\texttt{]}] = \mathcal{I}[\texttt{d}_1] \cap \cdots \cap \mathcal{I}[\texttt{d}_n]$$

Model Given an interpretation $\mathfrak{I} = (\mathcal{D}, \mathcal{I})$, a sentence is true under $\mathfrak{I}$ ($\mathfrak{I} \models$ sentence) if: Model

- $\mathfrak{I} \models (\texttt{c} \rightarrow \texttt{d})$ iff $\mathcal{I}[\texttt{c}] \in \mathcal{I}[\texttt{d}]$.
- $\mathfrak{I} \models (\texttt{d}_1 \sqsubseteq \texttt{d}_2)$ iff $\mathcal{I}[\texttt{d}_1] \subseteq \mathcal{I}[\texttt{d}_2]$.
- $\mathfrak{I} \models (\texttt{d}_1 \doteq \texttt{d}_2)$ iff $\mathcal{I}[\texttt{d}_1] = \mathcal{I}[\texttt{d}_2]$.

Given a set of sentences S, $\mathfrak{I}$ models S if $\mathfrak{I} \models S$.

Entailment A set of sentences S logically entails a sentence α if: Entailment

$$\forall \mathfrak{I} : (\mathfrak{I} \models S) \rightarrow (\mathfrak{I} \models \alpha)$$

5.3 Reasoning

5.3.1 T-box reasoning

Given a knowledge base of a set of sentences S, we would like to be able to determine the following:

Satisfiability A concept $\texttt{d}$ is satisfiable w.r.t. S if: Satisfiability

$$\exists \mathfrak{I}, (\mathfrak{I} \models S) : \mathfrak{I}[\texttt{d}] \neq \varnothing$$

Subsumption A concept $\texttt{d}_1$ is subsumed by $\texttt{d}_2$ w.r.t. S if: Subsumption

$$\forall \mathfrak{I}, (\mathfrak{I} \models S) : \mathfrak{I}[\texttt{d}_1] \subseteq \mathfrak{I}[\texttt{d}_2]$$

Equivalence A concept $\texttt{d}_1$ is equivalent to $\texttt{d}_2$ w.r.t. S if: Equivalence

$$\forall \mathfrak{I}, (\mathfrak{I} \models S) : \mathfrak{I}[\texttt{d}_1] = \mathfrak{I}[\texttt{d}_2]$$

Disjointness A concept $\texttt{d}_1$ is disjoint to $\texttt{d}_2$ w.r.t. S if: Disjointness

$$\forall \mathfrak{I}, (\mathfrak{I} \models S) : \mathfrak{I}[\texttt{d}_1] \neq \mathfrak{I}[\texttt{d}_2]$$

Theorem 5.3.1 (Reduction to subsumption)**.** Given the concepts $\texttt{d}_1$ and $\texttt{d}_2$, it holds that: Reduction to subsumption

- $\texttt{d}_1$ is unsatisfiable $\iff \texttt{d}_1 \sqsubseteq \bot$.
- $\texttt{d}_1 \doteq \texttt{d}_2 \iff \texttt{d}_1 \sqsubseteq \texttt{d}_2 \wedge \texttt{d}_2 \sqsubseteq \texttt{d}_1$.
- $\texttt{d}_1$ and $\texttt{d}_2$ are disjoint $\iff (\texttt{d}_1 \cap \texttt{d}_2) \sqsubseteq \bot$.

5.3.2 A-box reasoning

Given a constant $\texttt{c}$, a concept $\texttt{d}$ and a set of sentences S, we can determine the following:

Satisfiability A constant $\texttt{c}$ satisfies the concept $\texttt{d}$ if: Satisfiability

$$S \models (\texttt{c} \rightarrow \texttt{d})$$

Note that it can be reduced to subsumption.

5.3.3 Computing subsumptions

Given a knowledge base KB and two concepts `d` and `e`, we want to prove:

$$KB \models (\texttt{d} \sqsubseteq \texttt{e})$$

The following algorithms can be employed:

Structural matching Structural matching

1. Normalize `d` and `e` into a conjunctive form:

$$\texttt{d} = \texttt{[AND d}_1 \ \ldots \texttt{d}_n\texttt{]} \qquad \texttt{e} = \texttt{[AND e}_1 \ \ldots \texttt{e}_m\texttt{]}$$

2. Check if each part of `e` is accounted by at least a component of `d`.

Tableaux-based algorithms Exploit the following theorem: Tableaux-based algorithms

$$(KB \models (C \sqsubseteq D)) \iff (KB \cup (x : C \sqcap \neg D)) \text{ is inconsistent}$$

Note: similar to refutation.

5.3.4 Open world assumption

Open world assumption If a sentence cannot be inferred, its truth values is unknown. Open world assumption

Description logics are based on the open world assumption. To reason in open world assumption, all the possible models are split upon encountering an unknown facts depending on the possible cases (Oedipus example).

5.4 Expanding description logic

It is possible to expand a description logic by:

Adding concept-forming operators Let `r` be a role, `d` be a concept, `c` be a constant and n a positive integer. We can extend our description logic with: Adding concept-forming operators

`[AT-MOST` n `r]` Individuals `r`-related to at most n other individuals.

Example. `[AT-MOST 1 :Child]` individuals with only a child.

`[ONE-OF` $\texttt{c}_1 \ \ldots \ \texttt{c}_n$`]` Concept only satisfied by $\texttt{c}_1 \ \ldots \ \texttt{c}_n$.

Example. `Beatles` $\doteq$ `[ALL :BandMember [ONE-OF john paul george ringo]]`

`[EXISTS` n `r d]` Individuals `r`-related to at least n individuals in the category `d`.

Example. `[EXISTS 2 :Child Male]` individuals with at least two male children.

Note: this increases the computational complexity of entailment.

Relating roles Relating roles

`[SAME-AS` $\texttt{r}_1$ $\texttt{r}_2$`]` Equates fillers of the roles r_1 and r_2

Example. `[SAME-AS :CEO :Owner]`

Note: this increases the computational complexity of entailment. Role chaining also leads to undecidability.

Adding rules Rules are useful to add conditions (e.g. `if` $\texttt{d}_1$ `then [FILLS r c]`). Adding rules

5.5 Description logics family

Depending on the number of operators, a description logic can be:

- More expressive.
- Computationally more expensive.
- Undecidable.

Attributive language ($\mathcal{AL}$) Minimal description logic with:

- Atomic concepts.
- Universal concept (`Thing` or $\top$).
- Bottom concept (`Nothing` or $\bot$).
- Atomic negation (only for atomic concepts).
- `AND` operator ($\sqcap$).
- `ALL` operator ($\forall$).
- `[EXISTS 1 r]` operator ($\exists$).

Attributive language complement ($\mathcal{ALC}$) $\mathcal{AL}$ with negation for concepts.

$\mathcal{F}$	Functional properties
$\mathcal{E}$	Full existential quantification
$\mathcal{U}$	Concept union
$\mathcal{C}$	Complex concept negation
$\mathcal{S}$	$\mathcal{ALC}$ with transitive roles
$\mathcal{H}$	Role hierarchy
$\mathcal{R}$	Limited complex roles axioms Reflexivity and irreflexivity Roles disjointness
$\mathcal{O}$	Nominals
$\mathcal{I}$	Inverse properties
$\mathcal{N}$	Cardinality restrictions
$\mathcal{Q}$	Qualified cardinality restrictions
$(\mathcal{D})$	Datatype properties, data values and data types

Table 5.1: Name and expressivity of logics

6 Web reasoning

6.1 Semantic web

Semantic web Method to represent and reason on the data available on the web. Semantic web aims to preserve the characteristics of the web, this includes: Semantic web

- Globality.
- Information distribution.
- Information inconsistency of contents and links (as everyone can publish).
- Information incompleteness of contents and links.

Information is structured using ontologies and logic is used as inference mechanism. New knowledge can be derived through proofs.

Uniform resource identifier Naming system to uniquely identify concepts. Each URI correspond to one and only one concept, but multiple URIs can refer to the same concept. URI

XML Markup language to represent hierarchically structured data. An XML can contain in its preamble the description of the grammar used within the document. XML

Resource description framework (RDF) XML-based language to represent knowledge. Based on triplets: Resource description framework (RDF)

```
<subject, predicate, object>
<resource, attribute, value>
```

RDF supports:

Types Using the attribute `type` which can assume an URI as value.

Collections Subjects and objects can be bags, sequences or alternatives.

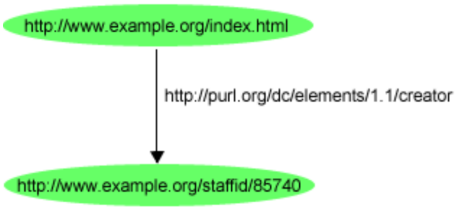
Meta-sentences Reification of the sentences (e.g. "X says that Y...").

RDF schema RDF can be used to describe classes and relations with other classes (e.g. `type`, `subClassOf`, `subPropertyOf`, ...) RDF schema

Representation

Graph A graph where nodes are subjects or objects and edges are predicates.

Example.



The graph stands for: `http://www.example.org/index.html` has a `creator` with staff id `85740`.

XML

Example.

```
<rdf:RDF
xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
xmlns:contact=http://www.w3.org/2000/10/swap/pim/contact#>
    <contact:Person rdf:about="http://www.w3.org/People/EM/
    contact#me">
        <contact:fullName>Eric Miller</contact:fullName>
        <contact:mailbox rdf:resource="mailto:em@w3.org"/>
        <contact:personalTitle>Dr.</contact:personalTitle>
    </contact:Person>
</rdf:RDF>
```

Database similarities RDF aims to integrate different databases:

- A DB record is a RDF node.
- The name of a column can be seen as a property type.
- The value of a field corresponds to the value of a property.

RDFa Specification to integrate XHTML and RDF. RDFa

SPARQL Language to query different data sources that support RDF (natively or through a middleware). SPARQL

Ontology web language (OWL) Ontology based on RDF and description logic fragments. Three level of expressivity are available: Ontology web language (OWL)

- OWL lite.
- OWL DL.
- OWL full.

An OWL has:

Classes Categories.

Properties Roles and relations.

Instances Individuals.

6.2 Knowledge graphs

Knowledge graph Knowledge graphs overcome the computational complexity of T-box reasoning with semantic web and description logics. Knowledge graph

- Use a simple vocabulary with a simple but robust corpus of types and properties adopted as a standard.
- Represent a graph with terms as nodes and edges connecting them. Knowledge is therefore represented as triplets `(h, r, t)` where `h` and `t` are entities and `r` is a relation.
- Logic formulas are removed. T-box and A-box can be seen as the same concept. There is no reasoning but only facts.

- Data does not have a conceptual schema and can come from different sources with different semantics.
- Graph algorithms to traverse the graph and solve queries.

KG quality Quality

Coverage If the graph has all the required information.

Correctness If the information is correct (can be objective or subjective).

Freshness If the content is up-to-date.

Graph embedding Project entities and relations into a vectorial space for ML applications. Graph embedding

Entity prediction Given two entities `h` and `t`, determine the relation `r` between them.

Link prediction Given an entity `h` and a relation `t`, determine an entity `t` related to `h`.

7 Time reasoning

7.1 Propositional logic

State The current state of the world can be represented as a set of propositions that are true according the observation of an agent. State

The union of a countable sequence of states represents the evolution of the world. Each proposition is distinguished by its time step.

Example. A child has a bow and an arrow, then shoots the arrow.

$$\text{KB}^0 = \{\texttt{hasBow}^0, \texttt{hasArrow}^0\}$$
$$\text{KB}^1 = \{\texttt{hasBow}^0, \texttt{hasArrow}^0, \texttt{hasBow}^1, \neg\texttt{hasArrow}^1\}$$

Action An action indicates how a state evolves into the next one. It is described using effect axioms in the form: Action

$$\texttt{action}^t \Rightarrow (\texttt{preconditions}^t \iff \texttt{effects}^{t+1})$$

Frame problem The effect axioms of an action do not tell what remains unchanged in the next state. Frame problem

Frame axioms The frame axioms of an action describe the unaffected propositions of an action. Frame axioms

Example. The action of shooting an arrow can be described as:

$$\texttt{SHOOT}^t \Rightarrow \{\texttt{hasArrow}^t \iff \neg\texttt{hasArrow}^{t+1}\}$$
$$\texttt{SHOOT}^t \Rightarrow \{\texttt{hasBow}^t \iff \texttt{hasBow}^{t+1}\}$$

Note that with m actions and n propositions, the number of frame axioms will be of order $O(mn)$. Inference for t time steps will have complexity $O(nt)$.

7.2 Situation calculus (Green's formulation)

Situation calculus uses first order logic instead of propositional logic.

Situation The initial state is a situation. Applying an action in a situation is a situation: Situation

$$s \text{ is a situation and } \texttt{a} \text{ is an action} \iff \texttt{result}(\texttt{a}, s) \text{ is situation}$$

(Note: in FAIRK module 1, `result` is denoted as `do`).

Fluent Function that varies depending on the situation (i.e. tells if a property holds in a given situation). Fluent

Example. $\texttt{hasBow}(s)$ where s is a situation.

Action Actions are described using:

Action

Possibility axioms Indicates the preconditions $\phi_{\mathtt{a}}$ of an action $\mathtt{a}$ in a given situation s:

Possibility axioms

$$\phi_{\mathtt{a}}(s) \Rightarrow \mathtt{poss}(\mathtt{a}, s)$$

Successor state axiom The evolution of a fluent F follows the axiom:

Successor state axiom

$$F^{t+1} \iff (\mathtt{ActionCauses}(F) \vee (F^t \wedge \neg \mathtt{ActionCauses}(\neg F)))$$

In other words, a fluent is true if an action makes it true or does not change if the action does not involve it.

Adding the notion of possibility, an action can be described as:

$$\begin{aligned} \mathtt{poss}(\mathtt{a}, s) \Rightarrow \Big(& F(\mathtt{result}(\mathtt{a}, s)) \iff \\ & (\mathtt{a} = \mathtt{ActionCauses}(F)) \vee \\ & (F(s) \wedge \mathtt{a} \neq \neg \mathtt{ActionCauses}(\neg F)) \Big) \end{aligned}$$

Unique action axiom Only a single action can be executed in a situation to avoid non-determinism.

Unique action axiom

7.3 Event calculus (Kowalski's formulation)

Event calculus reifies fluents and events (actions) as terms (instead of predicates).

Event calculus ontology A fixed set of predicates:

Event calculus ontology

$\mathtt{holdsAt}(F, T)$ The fluent F holds at time T.

$\mathtt{happens}(\mathtt{E}, T)$ The event $\mathtt{E}$ (i.e. execution of an action) happened at time T.

$\mathtt{initiates}(\mathtt{E}, F, T)$ The event $\mathtt{E}$ causes the fluent F to start holding at time T.

$\mathtt{terminates}(\mathtt{E}, F, T)$ The event $\mathtt{E}$ causes the fluent F to cease holding at time T.

$\mathtt{clipped}(T_i, F, T_j)$ The fluent F has been made false between the times T_i and T_j ($T_i < T_j$).

$\mathtt{initially}(F)$ The fluent F holds at time 0.

Domain-independent axioms A fixed set of axioms:

Domain-independent axioms

Truthness of a fluent

1. A fluent holds if an event initiated it in the past and has not been clipped.

$$\begin{aligned} \mathtt{holdsAt}(F, T_j) \Leftarrow & \mathtt{happens}(\mathtt{E}, T_i) \wedge (T_i < T_j) \wedge \\ & \mathtt{initiates}(\mathtt{E}, F, T_i) \wedge \neg \mathtt{clipped}(T_i, F, T_j) \end{aligned}$$

2. A fluent holds if it was initially true and has not been clipped.

$$\mathtt{holdsAt}(F, T) \Leftarrow \mathtt{initially}(F) \wedge \neg \mathtt{clipped}(0, F, T)$$

Note: the negations make the definition of these axioms in Prolog unsafe.

Clipping of a fluent

$$\texttt{clipped}(T_i, F, T_j) \Leftarrow \texttt{happens}(\texttt{E}, T) \wedge (T_i < T < T_j) \wedge \texttt{terminates}(\texttt{E}, F, T)$$

Domain-dependent axioms Domain-specific axioms defined using the predicates `initially`, `initiates` and `terminates`.

Domain-dependent axioms

Deductive reasoning Event calculus only allows deductive reasoning: it takes as input the domain-dependant axioms and a set of events, and computes a set of true fluents. If a new event is observed, the query need to be recomputed again.

Example. A room with a light and a button can be described as:

Fluents `lightOn` · `lightOff`

Events `PUSH_BUTTON`

Domain-dependent axioms are:

Initial state $\texttt{initially}(\texttt{lightOff})$

Effects of `PUSH_BUTTON` **on** `lightOn`

- $\texttt{initiates}(\texttt{PUSH\_BUTTON}, \texttt{lightOn}, T) \Leftarrow \texttt{holdsAt}(\texttt{lightOff}, T)$
- $\texttt{terminates}(\texttt{PUSH\_BUTTON}, \texttt{lightOn}, T) \Leftarrow \texttt{holdsAt}(\texttt{lightOn}, T)$

Effects of `PUSH_BUTTON` **on** `lightOff`

- $\texttt{initiates}(\texttt{PUSH\_BUTTON}, \texttt{lightOff}, T) \Leftarrow \texttt{holdsAt}(\texttt{lightOn}, T)$
- $\texttt{terminates}(\texttt{PUSH\_BUTTON}, \texttt{lightOff}, T) \Leftarrow \texttt{holdsAt}(\texttt{lightOff}, T)$

A set of events could be:

$$\texttt{happens}(\texttt{PUSH\_BUTTON}, 3) \cdot \texttt{happens}(\texttt{PUSH\_BUTTON}, 5) \cdot \texttt{happens}(\texttt{PUSH\_BUTTON}, 6)$$

7.3.1 Reactive event calculus

Allows to add events dynamically without the need to recompute the result.

Reactive event calculus

7.4 Allen's logic of intervals

Event calculus only captures instantaneous events that happen in given points in time.

Allen's logic of intervals Reasoning on time intervals.

Allen's logic of intervals

Interval An interval i starts at a time $\texttt{begin}(i)$ and ends at a time $\texttt{end}(i)$.

Interval

Temporal operators

Temporal operators

- $\texttt{meet}(i, j) \iff \texttt{end}(i) = \texttt{begin}(j)$
- $\texttt{before}(i, j) \iff \texttt{end}(i) < \texttt{begin}(j)$
- $\texttt{after}(i, j) \iff \texttt{before}(j, i)$
- $\texttt{during}(i, j) \iff \texttt{begin}(j) < \texttt{begin}(i) < \texttt{end}(i) < \texttt{end}(j)$
- $\texttt{overlap}(i, j) \iff \texttt{begin}(i) < \texttt{begin}(j) < \texttt{end}(i) < \texttt{end}(j)$

- $\texttt{starts}(i, j) \iff \texttt{begin}(i) = \texttt{begin}(j)$
- $\texttt{finishes}(i, j) \iff \texttt{end}(i) = \texttt{end}(j)$
- $\texttt{equals}(i, j) \iff \texttt{starts}(i, j) \wedge \texttt{ends}(i, j)$

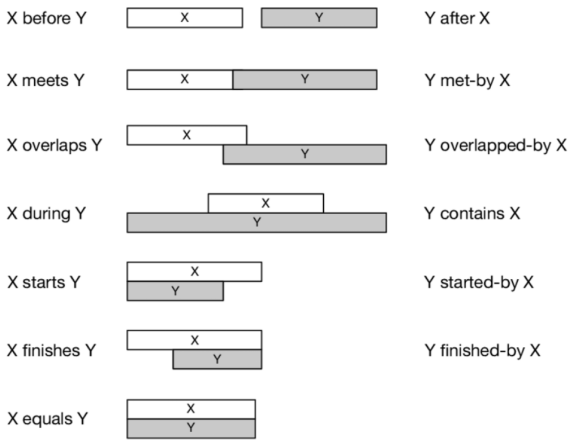


Figure 7.1: Visual representation of temporal operators

7.5 Modal logics

Logic based on interacting agents with their own knowledge base.

Propositional attitudes Operators to represent knowledge and beliefs of an agent towards the environment and other agents. Propositional attitudes

First-order logic is not suited to represent these operators.

Modal logics Modal logics have the same syntax of first-order logic with the addition of modal operators. Modal logics

Modal operator A modal operator takes as input the name of an agent and a sentence (instead of a term as in FOL).

Knowledge operator Operator to indicate that an agent $\texttt{a}$ knows P: Knowledge operator

$$\mathbf{K}_{\texttt{a}}(P)$$

Belief operator

Everyone knows operator

Common knowledge operator

Distribute knowledge operator

Depending on the operators, different modal logics can be defined.

Semantics An agent has a current perception of the world and considers the unknown as other possible worlds. Moreover, if P is true in any accessible world from the current one, the agent has knowledge of P.

Formally, semantics is defined on a set of primitive propositions ϕ using a Kripke structure $M = (S, \pi, K_1, \ldots, K_{\texttt{n}})$ where:

- S is a set of states of the world.
- $\pi : \phi \to 2^S$ specifies in which states each primitive proposition holds.
- $K_{\mathtt{i}} \subseteq S \times S$ is a binary relation where $(s,t) \in K_{\mathtt{i}}$ if an agent $\mathtt{i}$ considers the world t possible (accessible) from s. In other words, when the agent is in the world s, it considers t to be a possibly valid world. Obviously, $(s,s) \in K_{\mathtt{i}}$ for all states.

Example. Alice is in a room an tosses a coin. Bob is in another room an will enter Alice's room when the coin lands to observe the result.

We define a model $M = (S, \pi, K_{\mathtt{a}}, K_{\mathtt{b}})$ on ϕ where:

- $\phi = \{\mathtt{tossed}, \mathtt{heads}, \mathtt{tails}\}$.
- $S = \{s_0, h_1, t_1, h_2, t_2\}$ where the possible states are divided in three stages: the initial state (s_0), the result of the coin flip (h_1, t_1) and the observation of Bob (h_2, t_2).
- $\pi(\mathtt{tossed}) = \{h_1, t_1, h_2, t_2\}$
  $\pi(\mathtt{heads}) = \{h_1, h_2\}$
  $\pi(\mathtt{tails}) = \{t_1, t_2\}$
- $K_{\mathtt{a}} = \{(s,s) \mid s \in S\}$ as Alice observes everything in each world and does not have doubts.
  $K_{\mathtt{b}} = \{(s,s) \mid s \in S\} \cup \{(h_1,t_1),(t_1,h_1)\}$ as Bob is unsure of what happens in the second stage.

With this model, we can determine the truthness of sentences like:

$$(M, s_0) \models K_{\mathtt{a}}(\neg\mathtt{tossed}) \wedge K_{\mathtt{b}}\Big(K_{\mathtt{a}}\big(K_{\mathtt{b}}(\neg\mathtt{heads} \wedge \neg\mathtt{tails})\big)\Big)$$

$$(M, t_1) \models (\mathtt{heads} \vee \mathtt{tails}) \wedge \neg\mathtt{K_b}(\mathtt{heads}) \wedge \neg\mathtt{K_b}(\mathtt{tails}) \wedge \mathtt{K_b}(\mathtt{K_a}(\mathtt{heads}) \vee \mathtt{K_a}(\mathtt{tails}))$$

Axioms

Tautology All propositional tautologies are valid.

Modus ponens If φ and $\varphi \Rightarrow \psi$ are valid, then ψ is valid.

Distribution axiom Knowledge is closed under implication:

$$(K_{\mathtt{i}}(\varphi) \wedge K_{\mathtt{i}}(\varphi \Rightarrow \psi)) \Rightarrow K_{\mathtt{i}}(\psi)$$

Knowledge generalization rule An agent knows all the tautologies:

$$\forall \text{ structures } M : (M \models \varphi) \Rightarrow (M \models K_{\mathtt{i}}(\varphi))$$

Knowledge axiom If an agent knows φ, then φ is true:

$$K_{\mathtt{i}}(\varphi) \Rightarrow \varphi$$

In belief logic, this axiom is substituted with $\neg K_{\mathtt{i}}(\mathtt{false})$.

Introspection axioms An agent is sure of its knowledge:

Positive $K_{\mathtt{i}}(\varphi) \Rightarrow K_{\mathtt{i}}(K_{\mathtt{i}}(\varphi))$

Negative $\neg K_{\mathtt{i}}(\varphi) \Rightarrow K_{\mathtt{i}}(\neg K_{\mathtt{i}}(\varphi))$

Different modal logics can be defined based on the valid axioms.

7.6 Temporal logics

Logics based on modal logic with the addition of a temporal dimension. Time is discrete and each world is labeled with an integer. The accessibility relation maps into the temporal dimension with two possible evolution alternatives:

Linear-time From each world, there is only one other accessible world. Linear-time

Branching-time From each world, there are many accessible worlds. Branching-time

7.6.1 Linear-time temporal logic

Operators

Next ($\bigcirc\varphi$) φ is true in the next time step. Next

Globally ($\Box\varphi$) φ is always true from now on. Globally

Future ($\diamond\varphi$) φ is true sometimes in the future. It is equivalent to $\neg\Box(\neg\varphi)$. Future

Until ($\varphi\mathcal{U}\psi$) There exists a moment (now or in the future) when ψ holds. φ is guaranteed to hold from now until ψ starts to hold. Until

Weak until ($\varphi\mathcal{W}\psi$) There might be a moment when ψ holds. φ is guaranteed to hold from now until ψ possibly starts to hold. Weak until

Semantics Given a Kripke structure $M = (S, \pi, K_1, \ldots, K_n)$ where states are represented using integers, the semantic of the operators is the following:

- $(M, i) \models P \iff i \in \pi(P)$.
- $(M, i) \models \bigcirc\varphi \iff (M, i+1) \models \varphi$.
- $(M, i) \models \Box\varphi \iff \forall j \geq i : (M, j) \models \varphi$.
- $(M, i) \models \varphi\mathcal{U}\psi \iff \exists k \geq i : \big((M, k) \models \psi \wedge \forall j.i \leq j \leq k : (M, j) \models \varphi\big)$.
- $(M, i) \models \varphi\mathcal{W}\psi \iff ((M, i) \models \varphi\mathcal{U}\psi) \vee ((M, i) \models \Box\varphi)$.

Model checking Methods to prove properties of linear-time temporal logic based finite state machines or distributed systems. Model checking

8 Probabilistic logic reasoning

Probabilistic logic programming Adds probability distributions over logic programs allowing to define different worlds. Joint distributions can also be defined over worlds and allows to answer to queries. — Probabilistic logic programming

8.1 Logic programs with annotated disjunctions (LPAD)

8.1.1 Syntax — LPAD

`null` Atom that can only appear in the head of a clause and cancels the clause (i.e. equivalent of not having the clause).

The head of each clause is defined as a disjunction of atoms, each with a probability. More specifically, each clause has a probability distribution over its head.

Example.

```
sneezing(X):0.7 ; null:0.3 :- flu(X).
sneezing(X):0.8 ; null:0.2 :- hay_fever(X).
```

8.1.2 Distribution semantics

Worlds Given a clause C and a substitution θ such that $C\theta$ is ground, the following operations are defined for LPAD: — World

Atomic choice An atomic choice (C, θ, i) is the selection of the i-th atom in the head of C for grounding. — Atomic choice

Composite choice A composite choice κ is a set of atomic choices. The probability of a composite choice is the following: — Composite choice

$$\mathcal{P}(\kappa) = \prod_{(C,\theta,i)\in\kappa} \mathcal{P}(C, i)$$

where $\mathcal{P}(C, i)$ is the probability of choosing the i-th atom in the head of C.

Selection A selection σ is a composite choice where an atom from the head of each clause for each grounding has been chosen. In other words, a selection can be defined only when the program is ground. — Selection

A selection σ identifies a world w_σ and has probability:

$$\mathcal{P}(w_\sigma) = \mathcal{P}(\sigma) = \prod_{(C,\theta,i)\in\sigma} \mathcal{P}(C, i)$$

Example. Given the program:

```
sneezing(X):0.7 ; null:0.3 :- flu(X).
sneezing(X):0.8 ; null:0.2 :- hay_fever(X).
```

The possible worlds are:

$P(w_1) = 0.7 \cdot 0.8$

```
sneezing(bob) :- flu(bob).
sneezing(bob) :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

$P(w_2) = 0.3 \cdot 0.8$

```
null :- flu(bob).
sneezing(bob) :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

$P(w_3) = 0.7 \cdot 0.2$

```
sneezing(bob) :- flu(bob).
null :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

$P(w_4) = 0.3 \cdot 0.2$

```
null :- flu(bob).
null :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

Queries Given a ground query Q and a world w, the probability of Q being true in w is trivially: Queries

$$\mathcal{P}(Q \mid w) \begin{cases} 1 & \text{if } Q \text{ is true in } w \\ 0 & \text{otherwise} \end{cases}$$

The overall probability of Q is:

$$\mathcal{P}(Q) = \sum_w \mathcal{P}(Q, w) = \sum_w \mathcal{P}(Q \mid w)\mathcal{P}(w) = \sum_{w \models Q} \mathcal{P}(w)$$

Example. Given the program:

```
sneezing(X):0.7 ; null:0.3 :- flu(X).
sneezing(X):0.8 ; null:0.2 :- hay_fever(X).
```

The probability of `sneezing(bob)` is:

$$\mathcal{P}(\texttt{sneezing(bob)}) = \mathcal{P}(w_1) + \mathcal{P}(w_2) + \mathcal{P}(w_3)$$

9 Forward reasoning

Logical implication Simplest form of rule: Logical implication

$$p_1, \ldots, p_n \Rightarrow q_1, \ldots, q_m$$

where:

Left hand side (LHS) $p_1, \ldots, p_n$

Right hand side (RHS) $q_1, \ldots, q_m$

Modus ponens If A and $A \Rightarrow B$ are true, then we can derive that B is true. Modus ponens

Production rules Approach that allows to dynamically add facts to the knowledge base (differently from backward reasoning in Prolog). Production rules

When a fact is added, the reasoning mechanism is triggered:

Match Search for the rules whose LHS match the fact and (arbitrarily) decide which to trigger.

Conflict resolution Triggered rules are put in an agenda where conflicts are solved.

Execution The RHS of the triggered rules are executed and the effects are performed. The knowledge base is updated with the (copies of the) new facts.

These steps are executed until quiescence as the execution step may add new facts.

Working memory Data structure that contains the currently valid set of facts and rules. Working memory

The performance of a production rules system depends on the efficiency of the working memory.

9.1 RETE algorithm

RETE is an efficient algorithm for implementing rule-based systems.

9.1.1 Match

Pattern The LHS of a rule is expressed as a conjunction of patterns (conditions). Pattern

A pattern can test:

Intra-element features Features that can be tested directly on a fact.

Inter-element features Features that involves more facts.

Conflict set Set of all possible instantiations of production rules. Each rule is described as: Conflict set

$$\langle \text{Rule},\ \text{list of facts matched by its LHS} \rangle$$

Instead of naively checking a rule over all the facts, each rule has associated the facts that match its LHS patterns.

LHS network Compile the LHSs into networks:

Alpha-network For intra-element features. The outcome is stored into alpha-memories and used by the beta network. Alpha-network

Beta-network For inter-element features. The outcome is stored into beta-memories and corresponds to the conflict set. Beta-network

If more rules use the same pattern, the node of that pattern is reused and possibly outputting to different memories.

9.1.2 Conflict resolution

RETE allows different strategies to handle conflicts:

- Rule priority.
- Rule ordering.
- Temporal attributes.
- Rule complexity.

The best approach depends on the use case.

9.1.3 Execution

By default, RETE executes all the rules in the agenda and then checks possible side effects that modified the working memory in a second moment.
Note that it is very easy to create loops.

9.2 Drools framework

RETE-based rule engine that uses Java. Drools

Rule A rule has structure:

```
rule "rule␣name"
    // Rule attributes
when
    // LHS
then
    // RHS
end
```

Quantifiers

`exists P(...)` Trigger the rule once if at least a fact `P(...)` exists in the working memory.

`forall P(...)` Trigger the rule if all the instances of `P(...)` match. The rule can be executed multiple times.

`not P(...)` Trigger the rule if the fact `P(...)` does not exist in the working memory. Note that a negation in different positions might result in different behaviors.

Consequences Drools allows two types of RHS operations:

Logic

Insert Create a new fact and insert it in the working memory. Existing rules may trigger if they match the new fact.

If the conditions of the rule that inserted a fact are no longer true, the inserted fact is automatically retracted.

Retract Remove a fact from the working memory.

Modify A combination of retract and insert executed consecutively. The `no-loop` keyword can be used to avoid loops.

Non-logic Execution of Java code or external side effects.

Conflict resolution

Salience score

Agenda group Associate a group to each rule. The method `setFocus` can be used to prioritize certain groups.

Activation group Only one rule among the ones with the same activation group is executed (i.e. mutual exclusion).

9.3 Complex event processing

Event Information with a description and temporal information (instantaneous or with a duration). Event

Simple event Event detected outside an event processing system (e.g. a sensor). It does not provide any information alone. Simple event

Complex event Event generated by an event processing system and provides higher informative payload. Complex event

Complex event processing (CEP) Paradigm for dealing with a large amount of information. Takes as input different types of events and outputs durative events. Complex event processing

9.3.1 Drools

Drools supports CEP by representing events as facts.

Clock Mechanism to specify time conditions to reason over temporal intervals.

Sliding windows

Time-based window Select events within a time slice.

Length-based window Select the last n events.

Expiration Mechanism to specify an expiration time to events and discard them from the working memory.

Temporal reasoning Allen's temporal operators for temporal reasoning.

10 Business process management

Information system Contains static (raw) data partially describing the flow of a business. Information system

Business process management Methods to design, manage and analyze business processes by mining data contained in information systems. Business process management

Business processes help in making decisions and automations.

Business process lifecycle

Design and analysis Definition of models and schemas.

Configuration Execution of the business process.

Enactment Collect and analyze logs to make predictions.

Evaluation Assess the quality of the process.

Business process types

Organizational vs operational

Organizational Process described with its inputs, outputs, expected outcomes and dependencies.

Operational Process described disregarding its implementation.

Intra-organization vs inter-organization

Intra-organization Only activities executed within the business boundaries.

Inter-organization Part of the activities are executed outside the business and the process does not have control of them.

Execution properties

Degree of automation

Degree of repetition

Degree of structuring

10.1 Business process modelling

Activity instance Represents the actual work done during the execution of a business process. Activity instance

An activity instance can be described as a sequence of temporally ordered events. Formally, an activity instance i is defined as:

$$i = (E_i, <_i)$$

where $E_i \subseteq \{i_i, e_i, b_i, t_i\}$ is an event with:

- i_i for initialization;
- e_i for enabling;

- b_i for beginning;
- t_i for terminating.

$<_i$ is a relation order such that $<_i \subseteq \{(i_i, e_i), (e_i, b_i), (b_i, t_i)\}$.

Activity model Describes a set of similar activity instances. Activity model

10.1.1 Control flow modelling

Process modelling types

Procedural vs declarative

Procedural Based on a strict ordering of the steps. Uses conditional choices, loops, parallel execution, events. Procedural modelling

Subject to the spaghetti-like process problem.

Declarative Based on the properties that should hold during execution. Uses concepts as: executions, expected executions, prohibited executions. Declarative modelling

Closed vs open

Closed The execution of non-modelled activities is prohibited. Closed modelling

Open Constraints to allow non-modelled activities. Open modelling

The most common combination of approaches are:

Closed procedural process modelling

Open declarative process modelling

10.2 Closed procedural process modelling

Process model Set of process instances with a similar structure described as a graph.

Edges Directed arcs to describe temporal orderings.

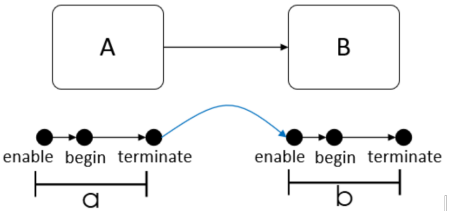
Nodes Nodes can be:

Activity models Unit of work. Activity

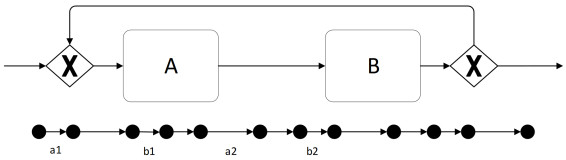
Event models Capture the events that involve activities. Event

Gateway models Control flow constructs. Basic patterns are: `sequence`, `and split`, `and join`, `exclusive or split`, `exclusive or join` Gateway

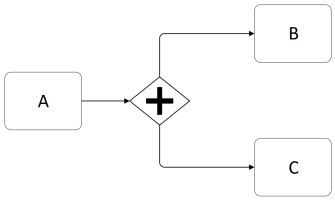
Example. Activity A is executed before activity B (`sequence` arc).



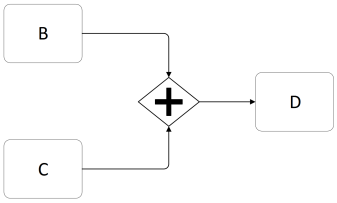
Example. Loop with `exclusive or splits`.



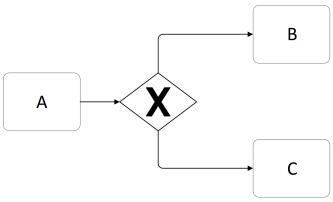
Example.
The `and split` allows to run B and C in parallel.



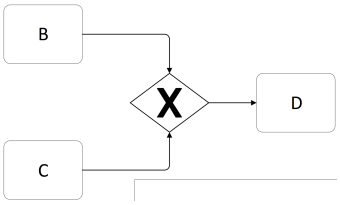
The `and join` allows to wait for both B and C to finish.



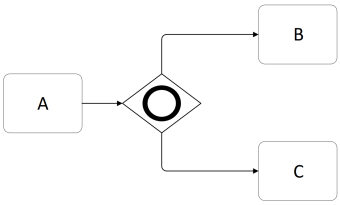
Example.
The `xor split` allows to run only one activity between B and C.



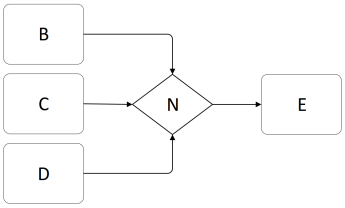
The `xor join` allows to wait for one activity between B and C to finish.



Example. The `or split` allows to run at least one activity between B and C.



Example. The `N-out-of-M join` allows to wait until N activities have finished.



10.2.1 Petri nets

Places Represent the points of execution of a process. Drawn as empty circles. Places

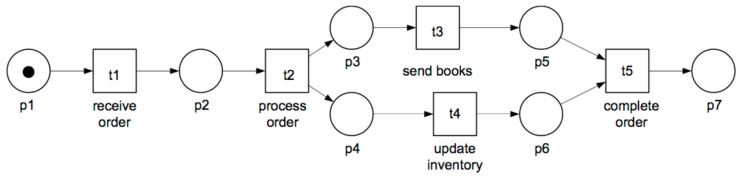
Tokens A token can reside in a place. It marks the current state of the process. Drawn as a small black circle. Tokens

Transitions Have input and output places. Drawn as rectangles. Transitions

Token play A transition is enabled when all the input places of a transition have a token and all the output places have not. An enabled transition can be fired: tokens are removed from the inputs and moved to the outputs.

Connections Arcs to connect places and transitions. Connections

Example.



Transition t_2 is a split. t_5 is a join.

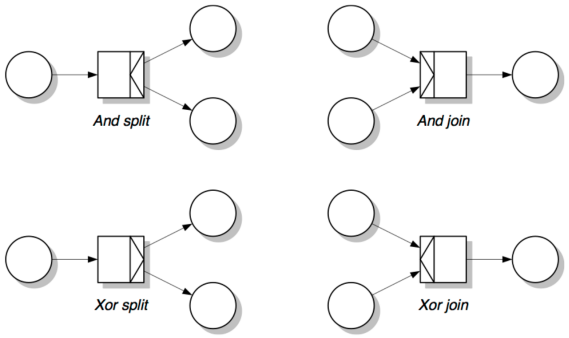
Petri nets	Business process modelling
Petri net	Process model
Transitions	Activity models
Tokens	Instances
Transition firing	Activity execution

Table 10.1: Petri nets and business process modelling concepts equivalence

10.2.2 Workflow nets

Restriction of Petri nets.

Transitions syntactic sugar Transitions



Triggers Can be attached to transitions. Triggers

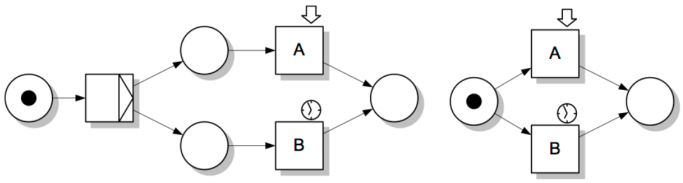
Automatic trigger Activity started automatically (standard behavior).

User trigger Activity started on user input.

External trigger Activity started on external events.

Time trigger Activity started at a certain time.

Example (Workflow nets with explicit and implicit `exclusive or split`).



10.2.3 Business process model and notation (BPMN)

De-facto standard for business process representation.

Basic elements

Activity Drawn as rectangles with optional decorations (e.g. a decorator to represent a task under human responsibility). — Activity

Event Drawn as circles. — Event

- **Start event** Indicates where and how (triggers) a process starts. Drawn as a thin-bordered circle.
- **Intermediate event** Event occurring after the start of a process, but before its end.
- **End event** Indicates the end of a process and optionally provides its result. Drawn as a thick-bordered circle.

Flow Drawn as arrows. — Flow

- **Sequence flow** Flow of processes (orchestration).
- **Message flow** Communication between processes and external entities.

Gateway Parallel, split and join. Drawn as rhombus. — Gateway

Pool Represent an independent participant with its own business process specification. — Pool

Lanes Resource classes within a pool. — Lanes

Data — Data

- **Data object** Local variable representing a temporary unit of information.
- **Data object reference** Reference to a data object.
- **Data object collection** Collection of data objects.
- **Data store** Persistent unit of information accessed by the process and external entities.

Figure 10.1: Data symbols

Reaction to events — Reactions

Throw Signals that something happened.

Catch Responds to a signal.

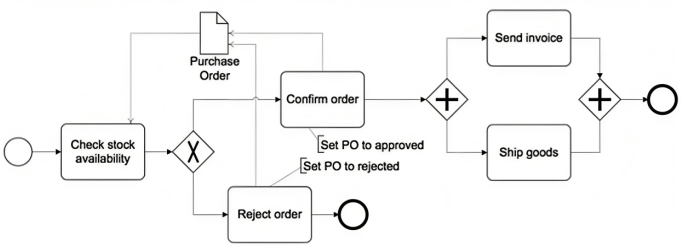


Figure 10.2: Example of BPMN

10.3 Open declarative process modelling

Define formal properties for process models (i.e. more formal than procedural methods). Properties defined in term of the evolution of the process (similar to the evolution of the world in modal logics)

10.3.1 Linear-time temporal logic in BPM

Based on the notion of world. Advancements in the process result in new worlds.

LTL model An LTL model is a set of events that happened in the execution of an instance of a process. LTL model

LTL execution trace An LTL execution trace is an LTL model based on the set of natural numbers. It represents the evolution of the world. LTL execution trace

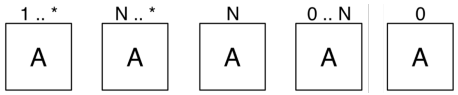
Syntax and semantics Follows from the syntax and semantics of linear-time temporal logic.

10.3.2 DECLARE

Based on constraints that must hold in every possible execution of the system.

Atomic activity Drawn as boxes. Atomic activity

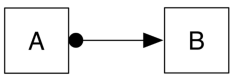
Unary constraints Defined on atomic activities. Unary constraints



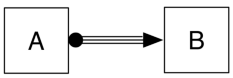
Binary constraints Connects two activities. Binary constraints

A solid circle indicates when the constraint is enforced. A directed arrow indicates time ordering.

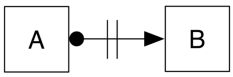
Response An execution of A should be eventually followed by an execution of B.



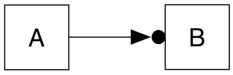
Chained response An execution of A should be immediately followed by an execution of B.



Negated response After the execution of A, B cannot be executed anymore.



Precedence An execution of B should have been preceded by an execution of A.



Semantics The semantic of the constraints can be defined using LTL.

Verifiable properties

Enactment Determine the next possible activities. Enactment

Conformance Check if an instance of a process respects the constraints. Conformance

Interoperability Determine if two DECLARE systems can be merged. Interoperability

Limits

- DECLARE cannot represent quantitative temporal constraints.
- Compared to closed procedural methods, it is more difficult to learn models.

10.4 Business process mining

Event log Sequence of events. Each event (trace) is a sequence of activities. Event log

Example. $L = \{\langle \texttt{abcd} \rangle, \langle \texttt{abcd} \rangle, \langle \texttt{acbd} \rangle\}$

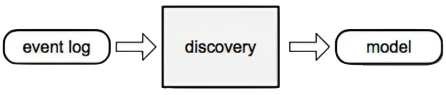
Business process mining Extract knowledge from event logs to improve a process. Business process mining

Possible applications are:

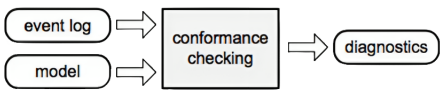
- Automated process discovery.
- Conformance checking.
- Organizational mining.
- Simulations.
- Process extension.
- Predictions.

Process mining types

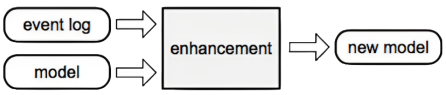
Discovery Takes as input an event log and outputs a model. Discovery



Conformance checking Takes as input an event log and a model and outputs if the log is conformant to the model. Conformance checking



Enhancement Takes as input an event log and a model and outputs a new model. Enhancement



10.4.1 Process discovery

Process discovery Learn a process model representative of the input event log. Process discovery

More formally, a process discovery algorithm is a function that maps an event log into a business process modelling language. In our case, we map logs into Petri nets (preferably workflow nets).

Remark. Process discovery is a unary classification problem. We are interested in learning a model fitted on the entire event log.

α-algorithm α-algorithm fixes the following interesting relations: α-algorithm

$>_L$) $a >_L b$ iff there exists a trace in the event log where a precedes b.

$\rightarrow_L$) $a \rightarrow_L b$ iff $a >_L b$ and $b \not>_L a$.

$\#_L$) $a \# b$ iff $a \not>_L b$ and $b \not>_L a$.

$\|_L$) $a \|_L b$ iff $a >_L b$ and $b >_L a$.

α-algorithm works as follows:

1. Look for the relations in the event log.
2. Focus on the most interesting relations and identify the biggest set of involved activities.
3. Remove redundancies.
4. Represent them as a Petri net.

Example. Given the event log $L = \{\langle \mathtt{abcd} \rangle, \langle \mathtt{abcd} \rangle, \langle \mathtt{abcd} \rangle, \langle \mathtt{acbd} \rangle, \langle \mathtt{acbd} \rangle, \langle \mathtt{aed} \rangle\}$, we want to apply the α-algorithm:

1. We determine the relations:

$$
\begin{aligned}
>_{L_1} &= \{(\mathtt{a},\mathtt{b}), (\mathtt{a},\mathtt{c}), (\mathtt{a},\mathtt{e}), (\mathtt{b},\mathtt{c}), (\mathtt{c},\mathtt{b}), (\mathtt{b},\mathtt{d}), (\mathtt{c},\mathtt{d}), (\mathtt{e},\mathtt{d})\} \\
\rightarrow_{L_1} &= \{(\mathtt{a},\mathtt{b}), (\mathtt{a},\mathtt{c}), (\mathtt{a},\mathtt{e}), (\mathtt{b},\mathtt{d}), (\mathtt{c},\mathtt{d}), (\mathtt{e},\mathtt{d})\} \\
\#_{L_1} &= \{(\mathtt{a},\mathtt{a}), (\mathtt{a},\mathtt{d}), (\mathtt{b},\mathtt{b}), (\mathtt{b},\mathtt{e}), (\mathtt{c},\mathtt{c}), (\mathtt{c},\mathtt{e}), (\mathtt{d},\mathtt{a}), (\mathtt{d},\mathtt{d}), (\mathtt{e},\mathtt{b}), (\mathtt{e},\mathtt{c}), (\mathtt{e},\mathtt{e})\} \\
\|_{L_1} &= \{(\mathtt{b},\mathtt{c}), (\mathtt{c},\mathtt{b})\}
\end{aligned}
$$

   And construct the footprint matrix:

	a	b	c	d	e
a	$\#_{L_1}$	$\rightarrow_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$
b	$\leftarrow_{L_1}$	$\#_{L_1}$	$\|_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$
c	$\leftarrow_{L_1}$	$\|_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$
d	$\#_{L_1}$	$\leftarrow_{L_1}$	$\leftarrow_{L_1}$	$\#_{L_1}$	$\leftarrow_{L_1}$
e	$\leftarrow_{L_1}$	$\#_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$

2. We determine the interesting relations as the pair of sets (P, Q) of activities such that:

$$\forall p \in P, q \in Q : (p \rightarrow q) \wedge (p \# p) \wedge (q \# q)$$

   This can be algorithmically done by building a footprint table whose rows and columns allow to obtain all the combinations of the activities. Therefore, the set of interesting relations is:

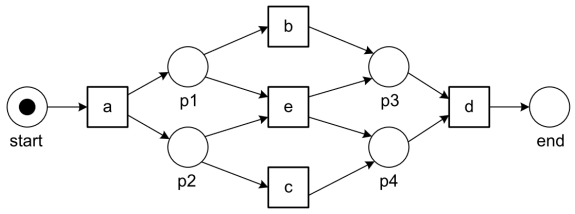
$$
\begin{aligned}
X_{L_1} = \{&(\{\mathtt{a}\},\{\mathtt{b}\}), (\{\mathtt{a}\},\{\mathtt{c}\}), (\{\mathtt{a}\},\{\mathtt{e}\}), (\{\mathtt{a}\},\{\mathtt{b},\mathtt{e}\}), (\{\mathtt{a}\},\{\mathtt{c},\mathtt{e}\}), \\
&(\{\mathtt{b}\},\{\mathtt{d}\}), (\{\mathtt{c}\},\{\mathtt{d}\}), (\{\mathtt{e}\},\{\mathtt{d}\}), (\{\mathtt{b},\mathtt{e}\},\{\mathtt{d}\}), (\{\mathtt{c},\mathtt{e}\},\{\mathtt{d}\})\}
\end{aligned}
$$

3. Then, we remove the redundancies in the set of interesting relations X_{L_1}:

$$Y_{L_1} = \{(\{\texttt{a}\}, \{\texttt{b}, \texttt{e}\}), (\{\texttt{a}\}, \{\texttt{c}, \texttt{e}\}), (\{\texttt{b}, \texttt{e}\}, \{\texttt{d}\}), (\{\texttt{c}, \texttt{e}\}, \{\texttt{d}\})\}$$

   It can be seen that all the relations in X_{L_1} can be derived from $Y_{L_1} \subset X_{L_1}$.

4. With the reduced set of interesting relations, we can build the Petri net.



α-algorithm has the following limitations:

- It can learn unnecessarily complex networks.
- It cannot learn loops.
- The frequency of the traces is not taken into account.

Model evaluation Different models can capture the same process described in a log. This allows for models that are capable of capturing all the possible traces of a log but are unable provide any insight (e.g. flower Petri net).

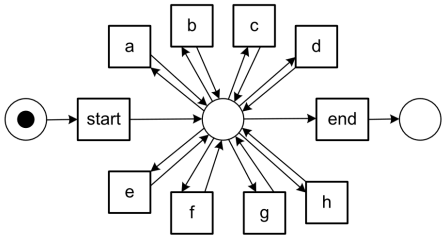


Figure 10.3: Example of flower Petri net

General judging criteria for a model are:

Fitness How well the model fits the majority of the traces. Fitness

Simplicity Based on the structure of the model. Simplicity

Precision How the model is able to capture rare cases. Precision

Generalization How the model generalize on the training traces. Generalization

10.4.2 Conformance checking

Descriptive model discrepancies The model need to be improved. Descriptive model

Prescriptive model discrepancies The traces need to be checked as the model cannot be changed (e.g. model of the law). The deviation of a trace can be desired or undesired. Prescriptive model

Remark. A trace can be seen as a sequence of symbols. It is possible to syntactically check them using a regular expression.

Token replay Given a trace and a Petri net, the trace is replayed on the model by moving tokens around. The trace is conform if the end event can be reached, otherwise it is not. Token replay

A modified version of token replay allows to add or remove tokens when the trace is stuck on the Petri net. These external interventions are tracked and used to compute a fitness score (i.e. degree of conformance).

Limitations:

- Fitness tend to be high for extremely problematic logs.
- If there are too many deviations, the model is flooded with tokens and may result in unexpected behaviors.
- It is a Petri net specific algorithm.

Alignment Given a trace l and the reference traces L_{ref}, alignment is based on the edit distance (i.e. minimum number of modifications) between l and the traces in L_{ref}. Alignment

The fitness score is based on the lowest and highest edit distances.

`<end of course>`