

Appunti FAIKR2

Ontologies and categories	3
Categories	3
Relations between categories	3
Physical composition	3
Measures	4
Things or stuff	4
Reasoning over Time	4
Propositional Logic for representing dynamics, and the Frame problem	4
Frame problem	4
Situation calculus	5
Event calculus	5
EC Ontology	5
Domain-independent axioms	6
Domain-dependent axioms	6
Allen's logic	6
Modal logics	6
Linear-time Temporal Logic	7
Operators:	7
Semantic:	8
Semantic networks	8
Frames	8
Description logics	9
Interpretation	10
Reasoning	10
Extension for DL	11
Protege	11
Semantic Web	11
SW Tools	12
eXtensible Markup Language - XML	12
Resource Description Framework (RDF/RDFS)	12
SPARQL	12
Ontology Web Language	12
Knowledge Graphs	12
Business Process Management	13
Modelling	14
Procedural:	14
Declarative:	14

Closed:	15
Open:	15
Procedural Close Process Modeling	15
Control flow patterns	15
BPM procedural, closed languages	16
Petri Nets	16
Workflow Nets	17
Business Process Model and Notation	18
Formal properties of process	19
Structural control-flow related properties	19
Object Lifecycle	19
Structural Soundness	19
Soundness	19
Relaxed Soundness	20
Weak Soundness	21
Lazy Soundness	21
Soundness criteria	21
Domain-related properties	21
Linear-time Temporal Logic and its use in the Business Process Modelling	22
Declare	22
Business Process Mining	23
Guiding Principles	23
Process Discovery	23
Conformance checking in Process Mining	24
Via Token replay	24
Via Alignment	24
Prolog	25
Basis and Lists	25
Execution and Resolution	25
The CUT operator !	25
Negation and SLDNF	26
Meta-Predicates	27
Meta-Interpreters	27
Dynamically modifying the program	28
Probabilistic Logic Programming - LPAD	28
Rule-based systems and forward reasoning	29
RETE algorithm	30
DROOLS	30
Complex Event Processing (CEP)	31

Ontologies and categories

It's a **formal** (means described by a language), **explicit** (information should be readily available or derivable in finite time), **description** (provide us information) **of a domain of interest** (related to some topic).

We can ever generalize ontologies to an upper level, encountering upper ontologies.

Categories

Knowledge is represented in terms of categories and objects belonging to them.

There are two main ways to represent categories in FOL:

- as a predicate (E.g: Car(aabb1234));
- as an object (E.g. Is(aabb1234,car)) (Reification).

The principal things that we want to represent about categories are membership and subclass. Then categories can be recognised through the object that belong to them and also by the properties that those objects have.

Relations between categories

- **Disjoint(S)**
for every c1 and c2 belonging to S, with c1 different from c2, we have that their intersection will be the empty set. (E.g. Disjoint({animals, vegetables}))
- **ExhaustiveDecomposition(S,C)**
We have that for each element i, it belongs to C exist a C2 belonging to S so that i belongs also to C2.
(E.g.: ExhaustiveDecomposition({Student, Professor}, PeopleInThisRoom))
- **Partition(S,C)** iff Disjoint(S) and ExhaustiveDecomposition(S,C).
(E.g.: ExhaustiveDecomposition({Student, Professor}, PeopleInThisRoom))

Physical composition

Objects can be made of other objects... the relation between an object and its parts is called **meronymy**: some objects (meronyms) are part of a whole (holonym).

is there any structural relation between the parts, and between the parts and the whole?

- Yes: the relation is usually named **PartOf** (it is transitive and reflexive)
(E.g.: PartOf(engine123,car_aa123bc))
- No: the relation is usually named **BunchOf**
(define objects in terms of composition of other countable objects)
(however, a structure in the composition relation is missing).
(E.g.: BunchOf({Bread1, Bread2, Bread3}))

Measures

Among the many types of properties, objects can have measures, and specifically:

- **Quantitative measures**: something that can be measured in a quantitative manner, w.r.t. some unit
- **Qualitative measures**: something that can be measured using terms, and these terms come with an total/partial order relation

Fuzzy Logic provides a (many-values-truth) semantics to qualitative measures.

Things or stuff

There are objects in our real world that defy the individuation criteria.

To solve that problem we can use properties, that can be:

- **Intrinsic**, if they are retained even when some division is applied;
- **Extrinsic**, if they refer to the object, and to the object structure relation with its parts.

A category of objects that will include in its definition only intrinsic properties is a substance;

Stuff is the most general substance category.

A category of objects that will include in its definition any extrinsic property is a count noun;

Thing is the most general object category.

Reasoning over Time

Propositional Logic for representing dynamics, and the Frame problem

Idea is to use propositional logic, and propositions, to represent an agent's beliefs about the world. a current state of the world would be represented as a set of propositions.

- There is a notion of **state** (countables) that captures the world at a certain instant.
The evolution of the world is given as a sequence of states.
- There are **actions**, that affect how each state evolves to the next one.
We would describe such effect in terms of **effect axioms**.
(E.g. $\text{shoot}(t) \Rightarrow (\text{hasArrow}(t) \Leftrightarrow \neg \text{hasArrow}(t+1))$)

Frame problem

The effect axioms fail to state what remains unchanged as the result of an action.

That's because "effect axiom tell us something about the "foreground"... what about the "background", alias frame?

A solution would be the **frame axioms**: for each proposition that is not affected by the action, we will state that it is unaffected.

But is simple to see that If we have m actions and n propositions, the set of frame axioms will be $O(mn)$ (**Representational frame problem**)

Situation calculus

The idea is to focus on the propositions describing the world, that we will be named **fluents**.

Each state is described by a set of fluents F .

Then, we define the following axioms:

$$F(t+1) \Leftrightarrow \text{ActionCauses}F(t) \vee (F(t) \wedge \neg \text{ActionCausesNot}F(t)).$$

- The initial state is called a **situation**;
- If a is an action and s a situation, then $\text{Result}(s, a)$ is a situation;
- A function/relation that can vary from one situation to the next is called a fluent.;
- Introduces preconditions of an action (defined by possibility axioms ($\text{Poss}(a,s)$));
- To avoid non-determinism and/or conflicts/incoherencies, it has a "unique action" axiom: only one action can be executed in a situation.

Problems:

- Is not possible to assert the truthness of a fluent in a situation.
- Actions are discrete, instantaneous, and happen one at a time
- The Situation Calculus defines what is true before the action and after the action, but say nothing on what is true during the action.

Event calculus

- Based on points of time ;
- Reifies both fluents and events into terms: $\text{HoldsAt}(\text{fluent}, \text{situation/action})$;
- Fluents are properties whose truthness value changes over time;
- Allows to link multiple different events to the same state property (named fluent);
- State property changes can depend also from other states;
- Allows to reason on meta-events of state property changes.

The formulation comprises an ontology and two distinct set of axioms:

1. Event calculus "ontology" (fixed)
2. Domain-independent axioms (fixed)
3. Domain-dependent axioms (application dependent)

EC Ontology

- $\text{HoldsAt}(F, T)$: The fluent F holds at time T
- $\text{Happens}(E, T)$: event E (i.e., the fact that an action has been executed) happened at time T
- $\text{Initiates}(E, F, T)$: event E causes fluent F to hold at time T (used in domain-dependent axioms...)
- $\text{Terminates}(E, F, T)$: event E causes fluent F to cease to hold at time T (used in domain-dependent axioms...)
- $\text{Clipped}(T1, F, T)$: Fluent F has been made false between $T1$ and T (used in domain-independent axioms), $T1 < T$
- $\text{Initially}(F)$: fluent F holds at time 0

Domain-independent axioms

Two axioms define when a fluent is true:

- $\text{HoldsAt}(F, T) \Leftarrow \text{Happens}(E, T1) \wedge \text{Initiates}(E, F, T1) \wedge (T1 < T) \wedge \neg \text{Clipped}(T1, F, T)$
- $\text{HoldsAt}(F, T) \Leftarrow \text{Initially}(F) \wedge \neg \text{Clipped}(0, F, T)$

An axiom defines the clipping of a fluent:

- $\text{Clipped}(T1, F, T2) \Leftarrow \text{Happens}(E, T) \wedge (T1 < T < T2) \wedge \text{Terminates}(E, F, T)$

Domain-dependent axioms

A collection of axioms of type $\text{Initiates}(\dots)/\text{Terminates}(\dots)$, and $\text{Initially}(\dots)$

- $\text{Initially}(F)$: the fluent F holds at the beginning
- $\text{Initiates}(\text{Ev}, F, T)$: the happening of event Ev at time T makes F to hold; it can be extended with many (pre-)conditions
- $\text{Terminates}(\text{Ev}, F, T)$: the happening of event Ev at time T makes F to not hold anymore.

EC is so important because it:

- it allows to represent the state of a system in logical terms, in particular FOL
- It allows to represent and reason on how a system evolves as a consequence of happening events
- It is an easier formalism (w.r.t. other solutions)
- It is declarative/logic based
- It is very used in monitoring and simulation systems

Limits:

Allows deductive reasoning only. (What if a new happened event is observed? New query is needed: re-computes from scratch the results... computationally very costly)

Allen's logic

It was created for reasoning over temporal aspects.

Allen proposed 13 temporal operators:

- $\text{Meet}(i, j) \Leftrightarrow \text{End}(i) = \text{Begin}(j)$
- $\text{Before}(i, j) \Leftrightarrow \text{End}(i) < \text{Begin}(j)$
- $\text{After}(j, i) \Leftrightarrow \text{Before}(i, j)$
- $\text{During}(i, j) \Leftrightarrow \text{Begin}(j) < \text{Begin}(i) < \text{End}(i) < \text{End}(j)$
- $\text{Overlap}(i, j) \Leftrightarrow \text{Begin}(i) < \text{Begin}(j) < \text{End}(i) < \text{End}(j)$
- $\text{Starts}(i, j) \Leftrightarrow \text{Begin}(i) = \text{Begin}(j)$
- $\text{Finishes}(i, j) \Leftrightarrow \text{End}(i) = \text{End}(j)$
- $\text{Equals}(i, j) \Leftrightarrow \text{Begin}(i) = \text{Begin}(j) \text{ AND } \text{End}(i) = \text{End}(j)$

Modal logics

Each agents in our worlds has its own knowledge, it act in a certain environment (whose

knowledge is known), it has no way to reason upon its own (or another agent's) knowledge and it takes decision based on what it knows.

The idea was to introduce new, special operators, named Modal operators, that behaves differently from logical operators.

- Each modal operator takes two input: the name of an agent, and the sentence it refers to
- Semantics is extended with the notion of possible worlds
- Possible worlds are related through an accessibility relation (that might depend on the operator)
- Different operators define different logics

Operator K for indicating that agent a knows P: K_aP

An agent in a given scenario is not omniscient so it might think that the unknown is depicted as possible worlds (accessible from its own current world). But we don't want a real notion for accessibility, the idea is that an agent knows p if in any accessible worlds from the current, p is true.

Given a set of primitive propositions, a Kripke structure $M=(S, \pi, K_1, \dots, K_n)$ is defined:

- S is the set of states of the world
- $\pi: \text{Prop} \rightarrow 2^S$ specifies for each primitive proposition the set of states at which the proposition hold
- K_1, \dots, K_n are binary relations over S

Some axioms:

- Axiom A0: All instances of propositional tautologies are valid
- MP Modus Ponens: if A is valid and $A \Rightarrow B$ is valid, then B is valid
- A1: knowledge is closed under implication. (Distribution Axiom)
 $(K_i\phi \wedge K_i(\phi \Rightarrow \psi)) \Rightarrow K_i\psi$
- (Knowledge Generalization Rule) G: for all structures M , if $M \models \phi$ then $M \models K_i\phi$
- (Knowledge Axiom) A2: $K_i\phi \Rightarrow \phi$
- (Positive Introspect Axiom) A3: $K_i\phi \Rightarrow K_iK_i\phi$
- (Negative Introspect Axiom) A3: $\neg K_i\phi \Rightarrow K_i\neg K_i\phi$

Linear-time Temporal Logic

The accessibility relation now is mapped into the temporal dimension.

Each world is labeled with an integer: time is then discrete.

Two possible ways for the world to evolve:

- Linear-time: from each world, there is only one other accessible world
- Branching-time: from each world, many worlds can be accessed

Operators:

- Something is true at the next moment in time $\bigcirc X$
- Something that is always true in the future $\Box X$
- Something that is true sometimes in the future $\Diamond X$

- There exists a moment when Something X holds and Y will continuously hold from now until this moment $Y \cup X$
- X will continuously hold from now on unless Y occurs, in which case X will cease $X \mathcal{W} Y$

Semantic:

A proposition is entailed when:

- $(M, i) \models p$ iff $i \in \pi(p)$
- $(M, i) \models \bigcirc \varphi$ iff $(M, i + 1) \models \varphi$
- $(M, i) \models \Box \varphi$ iff $(M, j) \models \varphi, \forall j \geq i$
- $(M, i) \models \varphi \mathcal{U} \psi$ iff $\exists k \geq i$ s.t. $(M, k) \models \psi$ and $\forall j. i \leq j \leq k (M, j) \models \varphi$
- $(M, i) \models \varphi \mathcal{W} \psi$ iff either $(M, i) \models \varphi \mathcal{U} \psi$ or $(M, i) \models \Box \varphi$

Many tools searching for approaches for facing model checking adopted LTL.

Semantic networks

A **semantic network**, or frame network is a knowledge base that represents semantic relations between concepts in a network. (using geometric shapes and arrows).

Categories and objects have the same representation.

There are four types of relation with the same representation.

Inheritance is easy to represent.

Problems:

- FOL have no negation and we don't know the properties of quantifiers.
- Diamond problem with multi inheritance,

Advantages:

- Default properties can be assigned directly to the categories, and we can also include a call to a procedure/algorithm

Frames

Similar to semantic networks.

Intuitively: a frame is a piece of knowledge that describes an object in terms of its properties

- it has a (unique) name
- each property is represented through couples (slot – filler)

Slots can contain additional information about the fillers, named facets.

In the Frames proposal, an object belongs to a category if it is similar enough to some typical members of the category, named prototypes. (Exceptions allowed for defeasible objects).

Description logics

Description Logic is a (family of) logic that focus on the description of the terms.

We would like to represent:

- individuals (individuals)
- category nouns (concepts)
- relational nouns (roles)

In addition we are really interested in the generalization relation.

Problems of other logics:

- FOL focuses on sentences and does not help you on reasoning on complex categories
- Frames is all user defined, roles can have filler while slots can't

In DL we use logical symbols and non-logical symbols:

Logical Symbols	Non-Logical Symbols
<ul style="list-style-type: none">• punctuation: (.), [,]• positive integers• concept-forming operators: ALL, EXISTS, FILLS, AND• connectives: \sqsubseteq, \doteq, \rightarrow	<ul style="list-style-type: none">• Atomic concepts: Person, FatherOfOnlyGirls (camel casing, first letter capital, same as OOP)• Roles: :Height, :Age, :FatherOf (same as concepts, but precede by columns)• constants: john, federicoChesani (camel casing, but starting with uncapitalized letter)

Complex Concepts can be created by combining **atomic concepts** together, using the **concept forming operators**

- every 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 $d_1 \dots d_n$ are concepts, then $[AND\ d_1 \dots d_n]$ is a concept.

Where the concept forming operators are:

- $[ALL\ r\ d]$ Stands for those individuals that are r -related only to individuals of class d .
- $[EXISTS\ n\ r]$ It stands for the class of individuals in the domain that are related by relation r to at least n other individuals.
- $[FILLS\ r\ c]$ Stands for those individuals that are related r -related to the individual identified by c .
- $[AND\ d_1 \dots d_n]$ Stands for anything that is described by d_1 and by $\dots d_n$. Each individual is a member of all the categories $d_1 \dots d_n$. If we refer to the notion of sets, here we have the idea of intersection.

Taken two concepts d_1 and d_2 and a constant c_1 , $d_1 \text{ sub } d_2$ is a sentence, $d_1 = d_2$ is a sentence and $c \text{ implies } d_1$ is a **sentence**. (intended to be true or false in the domain)

A KB in description logic is a collection of sentences of this form.

Then:

- constants stand for **individuals** in some application domain;
- concepts stand for **classes** or **categories** of individuals;
- roles stand for **binary relations** over those individuals.

In many research area there is a distinction between:

- **A-Box**, Assertion Box: a list of facts about individuals
- **T-Box**, Terminological box: a list of sentences (also called axioms) that describes the concepts

Interpretation

An **Interpretation** \mathfrak{I} is a pair (D, I) where:

- D is any set of objects, called domain;
- I is a mapping called the interpretation mapping, from the non-logical symbols of DL to elements and relations over D

An interpretation can be given also for complex concepts.

Given an interpretation we can also determine if a sentence is true.

If there exists an interpretation \mathfrak{I} such that $\mathfrak{I} \models S$, we say that \mathfrak{I} is a **model** of S .

Entailment is defined as in First Order Logic: A set S of sentences logically entails α if for every interpretation \mathfrak{I} , if $\mathfrak{I} \models S$ then $\mathfrak{I} \models \alpha$.

Reasoning

Given a knowledge base expressed as a set S of sentences:

- Satisfiability
- Subsumption
- Equivalence
- Disjointness
- Does a constant c satisfies concept d ? $S \models (c \rightarrow d)$

They are all reducible to subsumption.

Calculate subsumption:

- **Through Structural Matching:**
We want to prove $KB \models (d \sqsubseteq e)$ and to do that we can put d and e in a special normalized form and then determine whether each part of the normalized e is accounted for by some part of the normalized d
- **Through Tableaux-based Algorithm:**
In order to prove $KB \models (d \sqsubseteq e)$ we will exploit the following result
 $KB \models (C \sqsubseteq D) \Leftrightarrow (KB \cup (x : C \sqcap \neg D))$

Differently by many other formalisms, Description Logics are based on an **Open World Assumption**. If a sentence cannot be inferred, then its truthness value is unknown. So the knowledge base can be incomplete.

Extension for *DL*

- **Adding concept-forming operators:**
 1. [AT-MOST n r] individuals r -related to at most n individuals
 2. [ONE-OF $c_1 \dots c_n$] is a concept satisfied only by c_i used in conjunction with ALL
 3. [EXIST n r d] individuals r -related to at least n individuals instances of d
- **Extending roles**
 1. [SAME-AS r_1 r_2] equalize roles r_1 and r_2
 2. Complex roles as conjunction of more simplex ones
- **Adding rules**

In the language presented there is no way of saying that all instances of one concept are also instances of another. So we can add something like $d_1 = d_2$ between different concepts.

A DL is named upon the operators included, therefore a set of letters indicate the expressivity of the logic and name the logic

Protege

It is an ontology editor. =(

Semantic Web

Goal: “use” and “reason upon” all the available data on the internet automatically

How? By extending the current web with knowledge about the content (semantic information)

The Semantic Web is about two things:

- common formats for integration and combination of data
- language for recording how the data relates to real world objects

Simple addition of information is not enough, are necessary:

- Structured informations (ontologies)
- some inference mechanism (Logic. DL in particular)
- Inference of new knowledge through proof.

SW Tools

eXtensible Markup Language - XML

Created for supporting data exchange between heterogeneous systems (hardware and software)

- No presentation information
- Human readable and machine readable
- Extensible, so that anyone can represent any type of data
- Hierarchically structured

Resource Description Framework (RDF/RDFS)

- Standard W3C
- XML-based language for representing "knowledge"
- A design criteria: provide a "minimalist" tool
- Based on the concept of triple: < subject, predicate, object >
- Good expressive power, can represent types, collections and meta-sentences

A possible representation is through a graph where subject and object are nodes and predicates are arrows.

Or through XML.

RDFS (schema) allows to describe classes and relations with other classes/resources.

RDFa is a specification for attributes to express structured data in XHTML.

SPARQL

SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware.

Ontology Web Language

The Web Ontology Language (OWL) is a family of knowledge representation languages for authoring ontologies.

Knowledge Graphs

Basic idea: just store the information in terms of nodes and arcs connecting the nodes.

- Logical formulas are missing.
- T-Box and A-Box are stored at the same "level".
- factual knowledge, in form of "triplettes"
- No conceptual schema
- No reasoning

- Graph algorithms used to fast traverse the graph, looking for the solution

Some key questions when evaluating the quality of a Knowledge Graph:

- Coverage
- Correctness
- Freshness

Relation with Machine Learning: entities and link predictions.

Difference between Semantic Web and Knowledge Graphs

The main difference between SW and KG is that in SW are present both A-box (containing facts) and T-box containing logical formulas that once applied to the facts in the A-box can generate more information.

So we can notice that reasoning have a fundamental role in SW and the expressive level of the formulas in the T-box can also be high order logic.

Obviously these reasoning operations have a cost that is not ever affordable, especially on big amounts of data, so KG offer a better solution under this point of view.

In fact KG have only a flat A-Box containing the facts, then all the solutions are found through a search in the graph constructed over them, without any kind of reasoning.

As a result of this, the logical consistency is not yet a requisite and this characteristic is particularly suitable for the web, where available information is often inconsistent.

Business Process Management

A **business process** is an activity or set of activities that accomplish a specific organizational goal.

BPM includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of BPs. Those represent the lifecycle of a business process.

Goals of BPM:

- Better understanding of business operations and their relationships
- Achieve flexibility
- Continuous process improvement
- Narrow the gap between the company view of its business, and its implementation

Two types of business process:

- **Organizational:** They describe inputs, outputs, expected outcomes, and dependencies
- **Operational:** Activities (and their relationships) are specified, while implementation is disregarded

Usually one organizational -> many operational

From a high level perspective

- **Intra-organization**
- **Inter-organization**

From an execution-oriented perspective:

- Degree of **automation**
- Degree of **repetition** (how often?)
- Degree of **structuring**: from fully predictable processes (production workflows) to ad-hoc processes.

An Activity **Instance** represents the actual work conducted during the execution of a business instance, i.e., the actual (minimal) unit of work.

An Activity **Model** describes a set of similar Activity Instances

An Activity Instance, during its life, passes through different states (represented as points in time).

Modelling

Three main aspects:

- Control flow
- Data
- Resources

Workflow Data Patterns (recurrent paradigms used to specify how BPs handle data):

- Data visibility
- Data interaction
- Data transfer
- Data-based routing

Workflow Resource patterns (recurrent allocation strategies of work items to resources):

- Direct allocation
- Role-based allocation
- Deferred allocation
- Authorization allocation
- Separation of duties
- History-based

And about control flow?

when considering the flow aspects, at least two dimensions must be considered: **Procedural vs. Declarative** process modelling and **Open vs. Closed** modelling.

Procedural:

Procedural approaches focus on the order of steps composing the process.

- based on concepts such as sequence and choices
- they try to ensure general properties of process models
- they implicitly impose time constraints
- they allow to compute time properties
- mainly based on the orchestration perspective
- Easy to deploy but suffer the "spaghetti-like process problem"

Declarative:

Declarative approaches focus on the properties that should hold along the process execution.

- based on concepts such as expected executions, prohibited executions
- implicitly support parallel execution, but sequences can be imposed as well
- they implicitly impose time constraints between activity executions
- guarantee the domain-related properties of the business process
- Easier to define because is more rule-like but loose support for automatic execution (more oriented to monitoring)

Closed:

Closed process models allows the execution of only the activities that are indeed envisaged by the process, at the right moment along the process execution.

- Activities not envisaged are prohibited by default: if any non-envisaged activity should happen, the process instance is faulty
- Envisaged activities happening out of the prescribed order will make the process faulty
- Easy to implement/deploy but it can be a too rigid approach

Open:

Open process models clearly specify:

- which activities can be done
- which activities are prohibited (within specific time windows)
- nothing about the "other" activities. +practically, the "other" activities are defined any way as a finite set...

Usually the couple are **Procedural Close** and **Declarative Open**

Procedural Close Process Modeling

Process Model: a blue print for a set of process instances with a similar structure. Each process model consists of nodes and directed edges to express the relationship between nodes.

Nodes divide into:

- Activity models (units of work)
- Event models (occurrences of relevant states)

- Gateway models (to express control flow constructs)

Control flow patterns

A number of patterns for expressing the control flow aspects.

There are a set of basic patterns (sequence, and split, and join, exclusive or split, exclusive or join). Other patterns are built on top of the basic ones.

- A **sequence** pattern states that an activity instance *b* in a process instance *p* is enabled after the completion of an activity instance *a* in *p*.
Activities model *A* and *B* are connected through a gate which type is sequence.
- **and** join (parallel join): opposite to and split \Rightarrow multiple threads join into one. wait for all the threads to be ended
- **xor** split (and xor join): only one of the several branches is chosen when splitting. During the join there is no synchronization (no wait for all the threads but only one)
- **or** split gate in a process model indicates the point where at least one of several branches is chosen.
- **or** join: multiple threads converge into one. We can both wait until all the branches end (deadlock possible) or trigger asap the first branch ends
- multiple **merge**: multiple threads of control join without synchronization
- **discriminator**: waits for one of the incoming branches to complete and trigger the subsequent activity (then wait for the other branches and ignore them)
- **n out of m**: wait for *N* out of *M* ($n \leq m$) branches to be completed (and ignore the remaining)
- **arbitrary circles**: introduce loop using xor split/join
- **multiple instances**: concatenation of several instances

Flows might incur in deadlocks. In general, we might want to check for a number of formal properties (this would be possible only in presence of formal semantics).

BPM procedural, closed languages

Petri Nets

Abstract Formalism for representing processes, and processes languages in general.

Graphical representation with an equivalent mathematical formalization

- Formal: semantics is well-defined and not ambiguous
- Abstract: no info about the execution environment
- Able to model dynamic systems with a static structure.

Structure:

- Places: indicated with an empty circle, that (possibly) will contain (at runtime) a token
- Transitions: rectangles, usually with a label. They have input and output places.

- Connections: directed arcs that connect places with transitions, and transitions with places.

More formally, petri nets are bipartite graphs: each arc connects a place and a transition, or a transition and a place.

Tokens: small, black filled circles, residing on places. Tokens change their position according to the static structure, and according to a set of fixed firing rules.

The distribution of tokens in a petri net represents the current state of a system/process.

Firing of a transition: it can fire if it is enabled... A transition is enabled if there is a token in each of its input places.

When a transition fires: one token is removed from each input place and one is added to each output place.

The movement of tokens, according to the structure and the firing rules, is called token play

Petri Nets and BPM:

- Petri Net → Process Model
- Transitions → Activity Models
- Instances → Tokens
- Firing of a transition → activity execution
- Process instance → at least one token

Problem: In classical Petri Net tokens cannot be distinguished one from each other. hence, they can host a single process instance at a time.

Each place can host one token in standard version, but in case we want to implement multiple instance nets there is possibility of hosting more tokens and consume more than one in transitions.

There are also coloured petri nets where process instances are recognisable by color of tokens.

Summing up: advantages of Petri Nets:

- Formal Semantics
- Graphical Representation and an immediate mapping into a mathematical representation
- Transition rules are mapped into (simple) mathematical functions as well
- Analysis of Process Properties is possible
- Tool Independence (from industrial viewpoint)

Workflow Nets

They are Petri Nets, where:

- Places represent conditions
- Transitions represent activity instances

- Tokens represent process instances
- Tokens are coloured, but data and expression are not indicated in the graphical representation
- Hierarchical structuring is fully supported

In addition there is a structural condition: only one initial place I, only one final place O and all the transitions are located in a path between I and O.

The environment around the process is represented through triggers with symbols identifying the meaning of the block

Final considerations:

- The exclusive or semantics is usually obtained through the definition of an expression criteria (a decision rule)
- Data is handled in an abstract way
- Decision criteria are supported, but only implicitly: no way to express them, no way to reason on them and on their properties
- Transitions fire instantly (the evolution of the net is instantaneous). Real activity instances instead have a duration

YAWL (Yet Another Workflow Language)

Lack of support for a number of control flow patterns

- Multiple instances of specific activities
- Or split / join
- Nonlocal firing behaviour: for example, the cancellation pattern would affect different parts/paths of a petri net
- Direct arcs between transitions (when places are anonymous conditions)
- Explicit split and join behaviour attached to the transitions
- Support to nonlocal behaviour
- Handling of multiple instance activities

Is based on YAWL Nets, so workflow nets with some extensions.

Business Process Model and Notation

BPMN is a model and notation to represent the logic behind “standard” business processes. BPMN reveals the allowed order of activities and when they happen. Nothing is said about where or why (not the internal tasks).

Very similar to flowcharts, but:

- is a formal specification.
- describes event-triggered behavior
- Tackles how different process communicate with each other.

Process (or orchestration) can be: (intra-organizational point of view)

- Private, non-executable (for documentation purposes (abstract))
- Private, executable (with fully specified information to enable executability)
- Public (interaction between a private BP and an external one)

Collaboration: interaction between two or more business entities:

- Multiple private processes with message exchange.
- Choreography (contract (expected behavior) between interacting participants)
- Conversation (message exchange)

Problems:

- No Method – how to make a good model?
- No Styling rules

We have several **Graphical elements**(organized in families) such as flow objects, data, connecting objects and other elements, swimlanes that organize grouping of modelling elements, artifacts that have additional info.

We have different **modelling levels**:

- descriptive process modelling (traditional flowchart),
- analytic process modelling (expansion of previous with reactive behaviours and exception handling),
- executable BPMN (enable process execution)





In BPM an **activity** is a unit of work performed in the process by a performer. Each activity is either a task (atomic unit of work labelled using a verb-noun form) or a subprocess (compound unit of work, new child BPMN process).

BPMN includes **gateway** of two types: split (one control flow splits into many) and join (viceversa). It also includes xor split/join, and gateway, a start and an end event.

Diamond with an inner X: XOR gateway.

Diamond with an inner +: AND gateway.

Basic elements:

	Activity.
	Event.
	Flow.
	Gateway.

Formal properties of process

If a process model guarantees a property, all process instances will ensure that property.

Two different types of properties:

- Structural, control-flow related properties
- Domain related properties

Structural control-flow related properties

Object Lifecycle

Process instances act on information objects (capture a set of information that are related to some aspect of my process instance)

Information objects should be manipulated following a set of rules: about the data (e.g., integrity constraints) or about how data evolve, i.e. about their behaviour

Information objects have a lifecycle that define which states, how, and when an information objects can pass through.

Structural Soundness

Already introduced in workflow nets to fix many strange behaviors in the net.
(initial place, final place and all the transitions are in a path between)

Soundness

Structural Soundness might not be enough (e.g. presence of deadlock)

A workflow system (PN, i) with a workflow net $PN=(P,T,F)$ is sound if and only if:

- For every state M reachable from state $[i]$ there exists a firing sequence leading from M to $[o]$
- State $[o]$ is the only state reachable from state $[i]$ with at least one token in place o .
- There are no dead transitions in the workflow net in state $[i]$.

Soundness can be checked through a reachability analysis (reachability graph)

Note: The reachability graph suffer of the state explosion issue. This means that for real world applications is not practicable

A different approach proceed as follows:

- Take a Workflow net PN
- Create a new workflow net PN' , by adding a transition t^* , and link $o \rightarrow t^*$ and $t^* \rightarrow i$

Theorem

Let $PN=(P,T,F)$ be a workflow net, and $t^* \notin T$. PN is **sound** if and only if (PN', i) , such that $PN'=(P', T', F')$, $T' = T \cup \{t^*\}$, and $F' = F \cup \{(o, t^*), (t^*, i)\}$, is **live** and **bounded**.

A Petri Net is **live** (aka live1) if for each transition t there is a firing sequence starting from $[i]$ that actives t .

A place in a Petri net is called **k-bounded** if it does not contain more than k tokens in all reachable markings, including the initial marking.

It is said to be **safe** if it is 1-bounded; it is **bounded** if it is k-bounded for some k.

A (marked) Petri net is called k-bounded, safe, or bounded when all of its places are.

A Petri net (graph) is called (structurally) bounded if it is bounded for every possible initial marking.

Note that a Petri net is bounded if and only if its reachability graph is finite.

Note: For arbitrary Petri Nets, liveness and boundedness are still complex, expect exponential run-time behaviour...(for some other is required polynomial time).

Relaxed Soundness

Soundness is a very strong property and quite often, BP models derived from real industrial cases do not fit with soundness

The idea behind relaxed soundness is that process models are acceptable if they allow process instances with the desired properties and undesired process instances are not disallowed (while soundness doesn't permit them!)

Definition: Let $S=(PN,i)$ be a workflow system. Let s, s' be firing sequences and M, M' be states. s is a sound firing sequence if it leads to a state from which a continuation to the final state $[o]$ is possible.

A workflow system $S=(PN, i)$ is relaxed sound if and only if each transition of PN is an element of some sound firing sequence.

allows models with certain "wrong" parts, such as deadlocks... (but each activity must take part to at least one good process instance).

Weak Soundness

What happens if we use workflow nets to describe choreographies deriving from communication between two nets?

A problem arises because, due to the composition, some paths/transition will be never enabled in the composed system... the resulting workflow would not be sound, nor relaxed sound

A workflow system (PN, i) is weak sound if and only if the following holds:

- For every state M reachable from state $[i]$ there exists a firing sequence leading from M to $[o]$,
- State $[o]$ is the only state reachable from state $[i]$ with at least one token in place o .

In this way we obtain weak soundness, that does not allow deadlocks, but it permits that certain parts will never take part to any process instance

Lazy Soundness

- It allows activities to be executed after the final state has been reached
- Deadlocks are not permitted before the final state has been reached
- Activities running after the reaching of the final state are named lazy

Let (PN, i) be a workflow system, with a workflow net $PN=(P,T,F)$. (PN, i) is lazy sound if and only if:

- For each state reachable from $[i]$, there is a sequence that will lead me to the end (but other token might remain around... lazy!!!)
- No state is reachable that will have more than one token in the final place o .

Soundness criteria

- P1: **Termination**: any process instance starting from $[i]$ will finally terminate into $[o]$
- P2: **Proper termination**: $[o]$ is the only state reachable from $[i]$, with a token in the final place
- P3: **No dead transitions**: each transition contribute to at least one process instance
- P4: **Transition Participation**: each transition participates to at least a process instance that starts from the initial state and terminates in the final state

Soundness \Leftrightarrow P1 and P2 and P3

Weak Soundness \Leftrightarrow P1 \wedge P2

Relaxed Soundness \Leftrightarrow P4

Domain-related properties

(we are not interested in) ???

Linear-time Temporal Logic and its use in the Business Process Modelling

Formally, an LTL time structure F , also called frame, models a single linear timeline.

F is a totally ordered set $(K, <)$.

In the following K will be assumed to be the set of natural numbers.

We are interested in mapping business processes, and in particular to reasons about events happening in the system.

The idea is that the state of a system at each time instant can be described by the events happening at that time instant, and that the evolution of a system is given by the sequences of system states, i.e. by the sequence of happened events.

Differences within classic LTL:

- Trace length: classic LTL models are used for reasoning and represent infinite traces. Clearly, in BPM such assumption is wrong, since BPM traces must be finite.
Consequence: classical LTL semantics over infinite trace need to be adjusted for finite traces.
- Can event happens simultaneously? In classical LTL, yes, they do!; when using them in BPM, no.
Assumption: at each time instant at most one event can happen;
further assumption: at each time instant at least one event can happen.

LTL formulae are made of three ingredients:

1. Atomic propositions: the events that are happening in a certain state, plus two special constants true and false.
2. Classical Logical propositional connectives
3. Temporal operators: the ones already seen plus some new.

The semantic is given w.r.t. an LTL execution trace and to a state. We use the logical entailment \models_{LT} and using $L, M, i \models_{LT} \varphi$ we mean that φ is true at time i in model M .

We can check properties in an LTL system because the system is represented as an LTL formula, so we can negate the property to be verified and build two different automata for both formula (negated and non) and look for deadlock or liveness properties.

Declare

Declarative, Open Process modelling (Declare) is based on declarative approach: constraint about the happening/prohibition of events and based on open (any not prohibit activity is allowed).

We can have unary constraint (atomic activities, existence, existence_n, absence)

We can have binary constraint that connect two or more activities and can impose a temporal ordering: one, two or three line (between activities) have a different meaning.

Two lines \Rightarrow no repetition of the triggering activity before the constraint is satisfied

Three lines \Rightarrow no other activities allowed before the constraint is satisfied

Which properties can be verified:

- **enactment** of a process model
- **conformance checking**: given a log, check if it respect all the constraints
- **interoperability**: combining different system built via DECLARE methods

Problem: temporal constraint and data constraints (data object life cycle)

Business Process Mining

idea: discover, monitor and improve real processes by extracting knowledge from event logs.

Three main types of process mining, from the input/output perspective:

- Discovery
- Conformance Checking

- Enhancement

Assumption: It is possible to sequentially record events. So a complete event log should exist.

Guiding Principles

1. **Event Data should be treated as first-class citizens** (and a standard for event logs has been proposed). The log should respect some minimum requirements.
2. **Log Extraction should be driven by questions:** It is rarely the case that logs are already available in the right format
3. **Concurrency, Choice and other basic control-flow constructs should be supported**
4. **Events should be related to model elements**
5. **Models should be treated as purposeful abstractions of reality**
6. **Process mining should be a continuous process**

Process Discovery

Once I have a log I want to learn a model which is representative.

We can use the **alpha-algorithm**:

Let us decide which are the relation we are interested in instauring ordering relations. For example we are surely interested in those relation where an event appears immediately after another event... -> ordering relations.

Idea: The -algorithm generate a footprint matrix (another way to represent data) and looks for sets (A,B) such that $A \rightarrow B \wedge A \# A \wedge B \# B$ by swapping rows and cols looking for a pattern.

Limits: it can learn complex nets that are not necessarily implicit transition places. No loops of dimension 1, 2. No local-dependencies and does not take into account frequencies and duplicate activities (nor silent transitions)

There are lots of other learning algorithms to use, and ask which is the best model to learn is wrong. It depends. Only the owner of the business knows which is the best model to use.

Conformance checking in Process Mining

The terms conformance checking has a specific meaning in Process Mining: it focuses on events on the log, and relates them to the process executions forecasted by the model.

Via Token replay

The starting point is a Petri Net.

Given a trace, it can be interpreted as the ordered list of transitions that must be activated. It is possible to replay a trace, following tokens that moves around in the PetriNet.

If a model can replay a trace, then the model fits that trace.

Naïve idea: let's count how many traces can be replayed by the model, and compute the fitness as the ratio between replayed traces and total number of traces in the log

Very limited approach: in this method, either a trace can be replayed or not... independently of how much part of the trace can be effectively replayed.

Better idea: let us replay the trace, and when needed, let us add/remove tokens then let us keep track of how many tokens we added/remove, and compute a more precise fitness function.

some drawbacks:

- fitness values tend to be too high for extremely problematic logs
- if there are many deviations, the net get flooded of tokens, thus allowing for any behavior
- the approach is PetriNet-specific
- if a case does not fit at all, an ad-hoc path is not considered/created
- more difficult to achieve if there are duplicated activities and/or silent transitions

Via Alignment

Alignment is based on aligning the case with the possible instance generated by the model. To compute fitness, two elements have to be determined: the optimal alignment defined as the alignment with lowest possible cost and the worst alignment possible.

Note that for each move of misalignment, different costs can be associated, on the base of the type of misalignment, and on the specific activity involved.

Prolog

Basis and Lists

Prolog program composed of: Facts (A.), Rules (A:- B,C,D.) and Goal (?:-A,B,C.)

In the rules, A is the head and (B,C,D) is the body of the clause. An atomic formula has the form $p(t_1 \dots t_n)$ where t_i is a term. A term can be a constant (lowercase), a variable (uppercase) or a function symbol $f(t_1 \dots t_n)$

A procedure is a set of clauses of a program P, whose heads have the same predicate symbol and the predicate symbols have the same arity. In the head we have the formal parameter and in the call we have the actual parameter. Unification is used as the mechanism for passing parameter.

Integers number can be represented using Peano notation $s(s(0))$ where s is the successor function.

conjunction: “ , ”, disjunction: “ ; ”.

We can evaluate expression using $\text{is} \rightarrow T \text{ is Expr}$ (of course Expr must be evaluated and unified with T). We can compare expression values ($<, >, \geq, \leq, =, \neq$). There is no iteration but we can use recursion as a way to get iterative behaviors.

Lists are the most used data structure. $[] \Rightarrow$ empty list. $[T|L]$ where T is a term and L is a (sub)list.

Possible operation on lists: $\text{isList}(L)$, $\text{member}(E, L)$, $\text{length}(L, N)$, $\text{append}(L1, L2, \text{Res})$, $\text{reverse}(L, \text{Res})$.

Execution and Resolution

A computation is the attempt to prove, through resolution, that a formula logically follows from the program (it is a theorem). Moreover, it aims to determine a substitution for the variables in the goal, for which the goal logically follows from the program.

The resolution adopted by LP is the SLD that has two non-determinisms:

- The rule of computation
- The search strategy

Prolog adopts SLD with the following choices:

- Rule of computation: – Rule "left-most";
- Search strategy – Depth-first with chronological backtracking

Note: SLD resolution is the basic inference rule used in logic programming.

The CUT operator !

The execution process is build upon at least two stacks: execution stacks (contains the activation records) and backtracking stack (set of open choice points). The evaluation of the CUT make some choice as definitive and non-backtrackable (removing block from backtracking stack) \Rightarrow this alter the control of the program and the logic itself. CUT can be used to achieve mutual exclusion between clauses.

E.g. suppose to have a clause in the form $p :- q_1, q_2, \dots, q_i, !, q_{i+1}, \dots, q_n$

The evaluation of ! succeeds, and it is ignored in backtracking;

All the choices made in the evaluation of goals q_1, q_2, \dots, q_i and goal p are made definitive.

Nothing changes for further computations of $q_{i+1} \dots q_n$.

Negation and SLDNF

Prolog allows only definite clauses, no negative literals. Moreover SLD does not allow to derive negative information.

Prolog don't use Closed World Assumption, that states: "If a ground atom A is not logical consequence of a program P, then you can infer $\sim A$ ".

This way of working is easy to implement but bring a bad consequence: due to FOL undecidability, there is no algorithm that establishes in a finite time if A is not logical consequence of P. Operationally, it happens that if A is not logical consequence of P, SLD resolution is not guaranteed to terminate and the consequence is that use of CWA must be restricted to those atoms whose proof terminates in finite time, i.e. those atoms for which SLD does not diverge.

Prolog use a less powerful version of CWA named Negation As Failure, which derives only the negation of atoms whose proof terminates with failure in a finite time. Given a program P, we name FF(P) the set of atoms for which the proof fails in a finite time and NF rule: $NF(P) = \{ \sim A \mid A \hat{=} FF(P) \}$.

Moreover If an atom A belongs to FF(P), then A is not logical consequence of P, but not all the atoms that are not logical consequence of P belong to FF(P).

SLDNF is a proposed extension of SLD resolution which includes NAF.

Let $:-L_1, \dots, L_m$ be the goal, where L_1, \dots, L_m are literals (atoms or negation of atoms). A SLDNF step is defined as:

- Do not select any negative literal L_i , if it is not "ground";
- If the selected literal L_i is positive, then apply a normal SLD step
- If L_i is $\sim A$ (with A "ground") and A fails in finite time (it has a finite failure SLD tree), then L_i succeeds, and the new resolvent is: $:- L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_m$

The selection of ground negative literals only is needed to ensure correctness and completeness of SLDNF.

Or simply:

To prove $\sim A$, where A is an atom, the Prolog interpreter try to prove A

If the proof for A succeed, then the proof of $\sim A$ fails

If the proof for A fails in a finite time, then $\sim A$ is proved successfully

Problem: Prolog implementation does not respect exactly the way of working of SLDNF, Prolog does not use a safe selection rule: it selects ALWAYS the left-most literal, without checking if it is ground. In this way can happen that are chosen terms containing unbounded variables and the computation may not give back the correct result.

The problem lies in the meaning of the quantifiers of variables appearing in negative literals:

E.g. $:- \sim f(X)$ is equivalent to ask if exist an X so that f(X) does not hold

instead in SLDNF we are searching for a proof for f(X), so exists X. f(X)

and then the result is negated, becoming forall X ($\sim f(X)$)

That is different from what we are searching for.

Meta-Predicates

In Prolog predicates (i.e., programs) and terms (i.e., data) have the same syntactical structure. As a consequence, predicates and terms can be exchanged and exploited in different roles.

There are several pre-defined predicates that allows to deal with these structures, and work with them.

- **call(T)**: the term T is considered as a atom (a predicate), and the Prolog interpreter is requested to evaluate (execute) it. Obviously, at the moment of the evaluation, T must be a non-numeric term
- **fail** takes no arguments (its arity is zero) and its evaluation always fails. As a consequence, it forces the interpreter to explore other alternatives.
- **setof and bagof** (X, pred, S) S is a set or list of elements X that satisfy the predicate pred.
- **findall**: particular case of bagof (with variables other than X are fixed with existentiality)
- **var(Term)** true if Term is currently a variable
- **nonvar(Term)** true if Term currently is not a free variable
- **number(Term)** true if Term is a number
- **ground(Term)** true if Term holds no free variables.
- **clause(Head, Body)** true if (Head, Body) is unified with a clause stored within the database program

Meta-Interpreters

Meta interpreters as meta-program who execute other programs. In Prolog a meta interpreter for a language L is defined as an interpreter of L, but written in Prolog.

The Prolog vanilla meta-interpreter is represented by the following clauses:

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

We can change this interpreter to obtain a new one with different features (for example the rightmost rule selection)

Dynamically modifying the program

When a Prolog program is consulted/loaded, its representation in terms of data structures (terms) is loaded into a table in memory.

Such table is often referred as the program database and we can change it.

- **assert(T)** Clause T is added to the database program.

- (asserta(T) at the beginning and assertz(T) at the end)
- **retract(T)** The first clause in the database that unifies with T is removed

When using assert and retract, the declarative semantics of Prolog is lost.
They can be used also for lemma generation to improve further computations.

Probabilistic Logic Programming - LPAD

Prolog allows to represent and to reason upon some knowledge but it does not deal with uncertainty and Probabilistic Reasoning \Rightarrow CRISP logic: a clause can be true or false, no uncertainty and probabilities.

A Probabilistic Logic Program (PLP) allow the definition of probabilities distributions over normal logic programs (called possible worlds). The distribution is extended to a joint distribution over worlds and interpretations (queries). The probability of a query is then obtained from this distribution.

In **LPAD** (Logic Programming with Annotated Disjunctions) the head of a clause is extended with disjunctions and each disjunction is annotated with a probability.

E.g. sneezing(X):0.7 ; null:0.3 :- flu(X).
 sneezing(X):0.8 ; null:0.2 :- hay_fever(X).
 flu(bob).

Worlds will be obtained by selecting one atom from the head of every grounding of each clause:

1. Ground the program
2. For each atom in each head, choose to include it or not

Grounding:

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

Generating the worlds: (e.g. of a possible world)

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

Then is calculated the probability of each generated world to become able to answer the queries.

Rule-based systems and forward reasoning

Rules are useful because defining rules allow to automatize lots of processes. In addition they

are easy to understand, to be expressed and humans adopt rules in every situation.

- Backward Chaining: start from consequences and look for premises that make the consequences true
- Forward Chaining: start from the premises and look which consequences are true.

Forward reasoning in rule-based systems can be described as follows:

- the knowledge base is defined before the reasoning step;
- the consequences are derived during the reasoning step;
- the knowledge base is augmented with only facts that have been the result of the application of the Modus Ponens rule to facts and rules in the knowledge base

In this kind of reasoning there is one main problem: What if, during the reasoning step, new information/new facts become available?

A naive solution can be restart the reasoning from scratch when needed (adding the new informations gained), but it is obviously a too expensive approach and we should also define when is better to restart the computation.

But there is another way of working: In the production rules approach, new facts can be dynamically added to the knowledge base and if new facts matches with the left hand side of any rule, the reasoning mechanism is triggered, until it reaches quiescence again.

In production rule systems RHS can have side effects like insertion, deletion and consequent retriggering of rules.

Production rules systems follow 4 steps, initially and when a new fact is added to the knowledge base:

1. Match: search for the rules whose LHS matches the fact and decide which ones will trigger.
2. Conflict Resolution/Activation: triggered rules are put into the Agenda, and possible conflicts are solved (FIFO, Saliency, etc).
3. Execution: RHS are executed, effects are performed, KB is modified, etc.
4. Go to step1.

An example of algorithm that implements these four steps is **RETE**.

Talking about the **differences** with relation to backward reasoning: forward chaining as the name suggests, start from the known facts and move forward by applying inference rules to extract more data, and it continues until it reaches to the goal (data-driven inference technique or bottom-up), whereas backward chaining starts from the goal, move backward by using inference rules to determine the facts that satisfy the goal (goal-driven inference technique or top-down).

RETE algorithm

The Rete algorithm is a **pattern matching algorithm** for implementing rule-based systems. In particular its aim is to solve the Many Pattern/Many Object Pattern Match Problem. RETE focuses on determining the so called "conflict set", i.e., the set of all possible instantiations of production rules such that

< Rule (RHS), list of elements matched by its LHS >

The idea for avoiding iteration over facts is storing, for each pattern, which are the facts that match it.

And the idea for avoiding iteration over rules is "compile" a LHS into a network of nodes.

There are (at least) two types of pattern:

- Patterns testing intra-elements features
- Patterns testing inter-elements features

We can see the differences between these two type of patterns in a simple example:

" If a professor has name X, and a student has name X, then notify the system"

In this case professor and students are intra-elements because they relates only to a single element per time and aren't in any kind of relation outside the element itself. Instead name X represent an inter-element features that consider more elements at time for the evaluation of the rule.

Then these two elements are compiled in two different type of networks: alpha and beta respectively.

The two next steps are **conflict resolution**, in which is defined the order in which the rules are selected when there are more available and the **execution** step, in which each activated instance of a rule will be executed.

DROOLS

DROOLS is a framework that encapsulate the three complementary business modeling techniques:

1. Business Rules Management
2. Business Processes Management
3. Complex Event Processing

Forward reasoning approach based on an evolution of the RETE algorithm.

Rules have the structure:

```
rule "Any string as id of the rule" // possible rule attributes
    when // LHS: premises or antecedents
    then // RHS: consequents or conclusions... side effects
end
```

Facts: LHS are defined using patterns and facts are puts into the WM. Facts can be Javabased

or defined within the rule with explicit list of fields.

RHS: Consequences can be of two types: logic if they affect the WM and non logic (such as piece of java code executed)

We can create new facts and insert them into the WM using insert that will possibly trigger some rules, or using insertLogical and the fact is removed when the condition in the rules that inserted the facts are no longer true. We can also remove facts using retract or change facts using modify/update. In order to avoid loops when dealing with facts we can use the clause no-loops

Complex Event Processing (CEP)

Complex Event Processing (CEP) is a paradigm (older than IoT and Big Data), for dealing with such huge amount of information

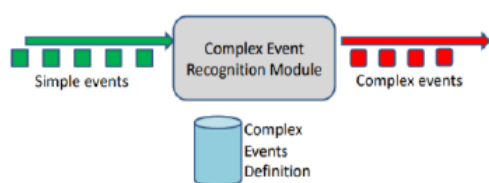
Collected information is described in terms of events:

- A description of something, in some (logic?) language
- A temporal information, about when something happened.
- Events can be instantaneous, or can have a duration

Complex Event Processing is about dealing with simple events, reason upon them and derive complex events.

Simple and complex events do not refer to the information payload, but rather to their ability to provide answer to the application question.

The term CEP capture the whole framework needed to recognize events. Usually, authors refer also to Complex Event Recognition Languages.



An event is characterized by input (instantaneous events, durative events or context informations) and output (durative events)

CEP and Drools

Drools Fusion provide support from CEP: events are particular facts that have a timestamp and support the Allen Algebra.

Of course we need a “clock” to reason over temporal intervals ⇒ so we can use the sliding windows that can be time-based.

CEP allow reasoning about events. In order to reason about the state of a system we need to use Event Calculus Framework.

