DISCLAIMER

These notes are supposed to explain some major topics covered in the lessons.

While they may not provide an exhaustive overview, they strive to complement the information presented in the slides with some explanations taken from additional readings in various textbooks.

If someone finds some errors or have any suggestion, it's possible to contact me on mmeschini001@gmail.com

MM

Cognitive Architectures for Robotics

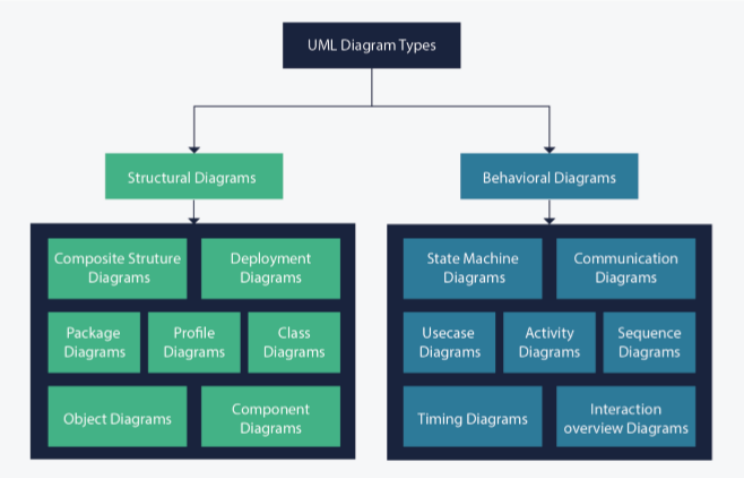
Unified Modelling Language

It is a general purpose language aimed at defining a standard method to visualize the way a data processing architecture is designed. UML is used to describe the behaviour and the structure of a data processing architecture.

Temporal sequence — How everything works

Note: UML is not a programming language, rather it is a visual language

We need it because in cognitive architectures for robots there are multiple modules and hence, they require a clear and concise way to design the information flow. It is used also to interact with cognitive scientists and colleagues from other fields. Finally, UML is linked with the object-oriented software design and analysis.



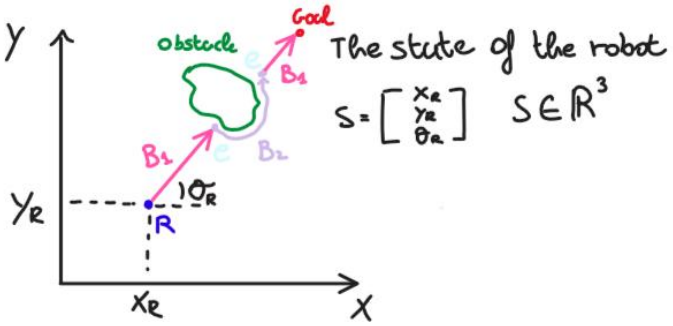
Structural diagrams capture static aspects. E.g. classes, objects, components

Behavioural diagrams capture dynamic aspects or behaviours E.g. state machine activity, use case sequence, timing, interaction

Relevant concepts

- **Object**: the fundamental unit of a software architecture, representing an independent entity
- **Class**: defines the structure and the functions of an object (it's like a template)
- **Inheritance**: a mechanism by which objects belonging to a "child" class inherit the properties of a "parent" class
- **Abstraction**: a mechanism allowing for hiding implementation details
- **Encapsulation**: a mechanism to cluster data and define how it can be used by means of an interface
- **Polymorphism**: functions and entities can exhibit a context-aware behaviour

Example:

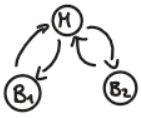


Behaviour 1: go ahead until an obstacle is detected or the goal is reached

Behaviour 2: follow the obstacle's contour until there is free space to reach the goal

How to change the behaviours?

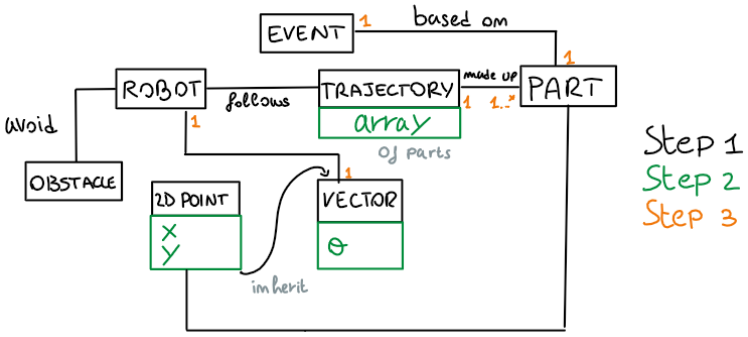
STATE MACHINE: switch state when an event occurs



CENTRALIZED another entity controls the switch

MM

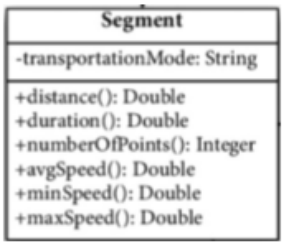
For that example it's possible to create a diagram like this:



Class diagram

Depicts the static structure of a software architecture by showing the classes, their methods and their attributes. They help the designers to identify relationships between classes or objects. Classes are represented using boxes organized in three compartments:

- Top: contains the class name
- Middle: contains the attributes
- Bottom: contains the operations that can be executed on the attributes

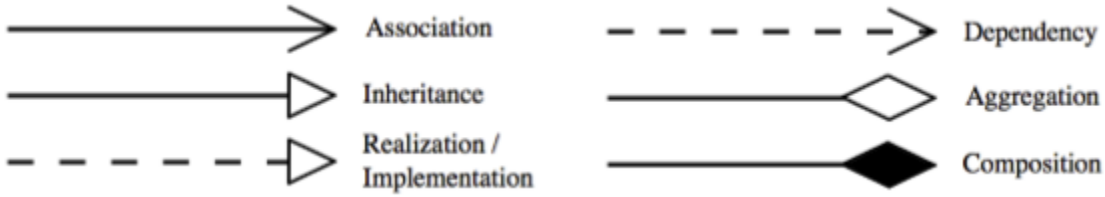


Visibility

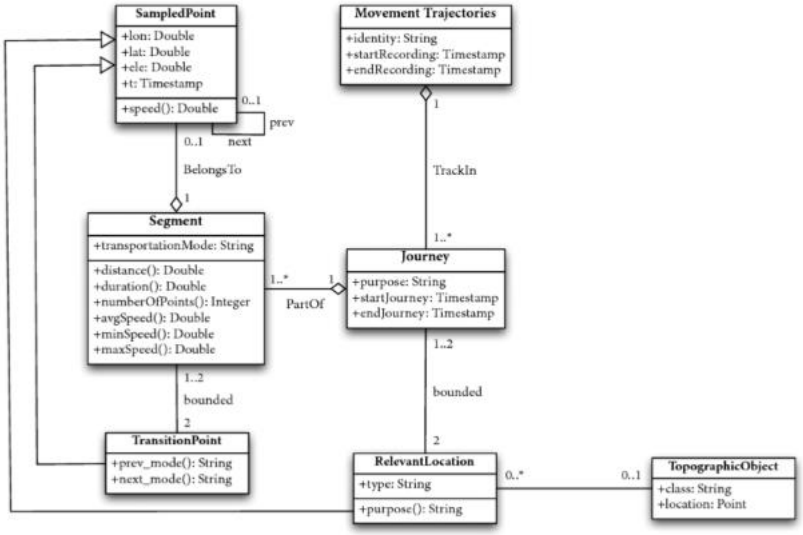
public	+	anywhere in the program and may be called by any object within the system
private	-	the class that defines it
protected	#	(a) the class that defines it or (b) a subclass of that class
package	~	instances of other classes within the same package

Relationship between classes

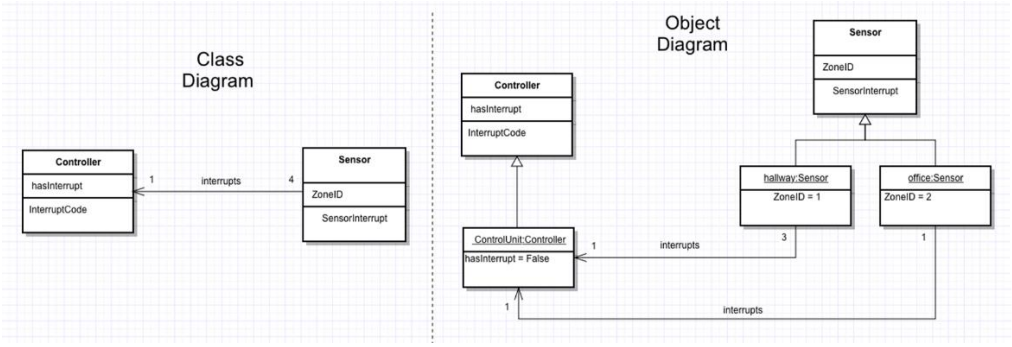
- **Association:** bi-directional and uni-directional (“aggregations”). They describe the reasons for the relationship and the rules that govern the relationship
- **Inheritance**: a sub-class is considered a specialised form of the super-class
- **Realization/Implementation**: a relationship between two elements, in which one implements or executes the behaviour that the other model elements specifies. (interface -> class)



Example: A complete model for trajectory

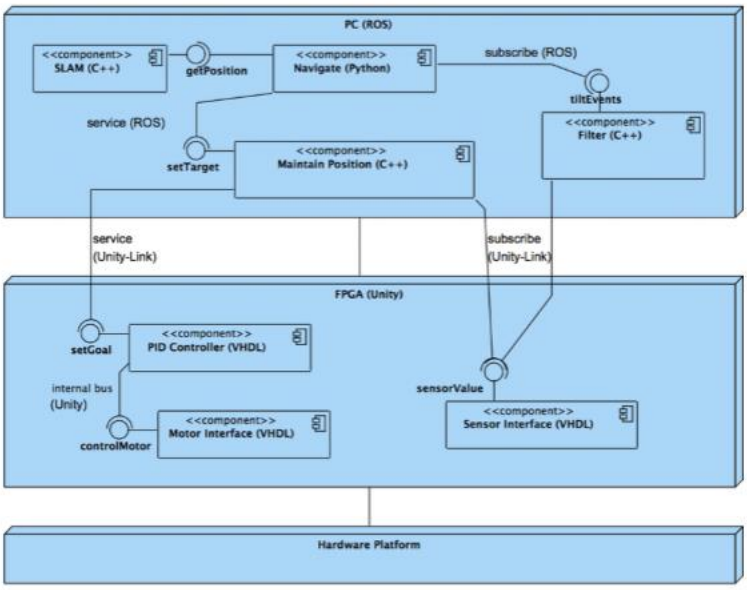


Object diagram

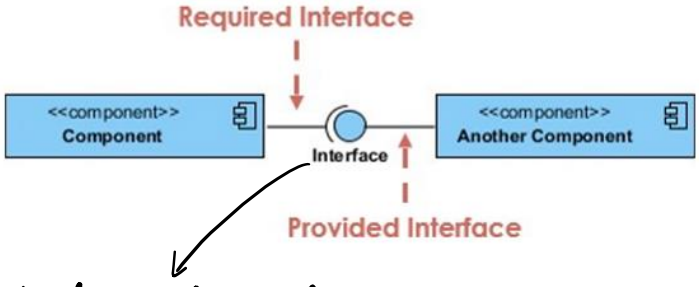


It shows the instances of the classes; it can explicit the relationship among objects. Using them is possible to study the behaviour of the architecture at a particular instant.

Component diagram

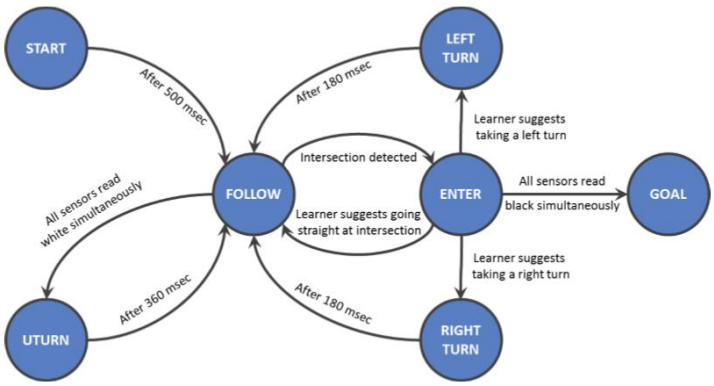


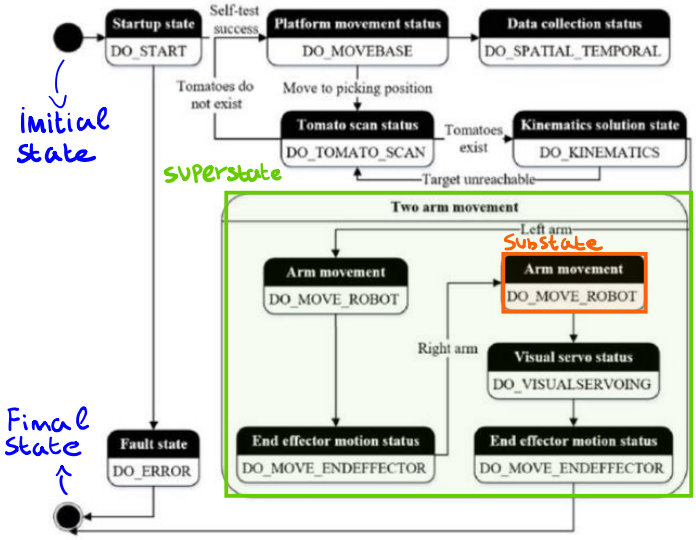
Depicts the structural relationship between software architecture elements and helps to understand if requirements have been met. It represents how physical components in an architecture are organised.



State Machine diagram

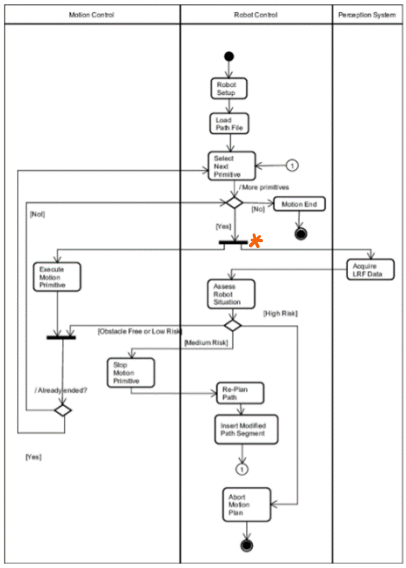
Used to describe the state-dependent behaviour of an object (each object has its own state machine, even if each state is encoded in the class). A state machine diagram describes all events, states and transitions for a single object. An object may respond differently to the same event depending on what state it is in.





MM

Activity diagram



Time ↓

Used to describe dynamic aspects of an architecture. It models the flow from one activity to another activity, describing how activities are coordinated.

* sends messages to other modules

Design Patterns

Motivation and background

Algorithm
1) COMPUTABLE (Neuromorphic computing, Quantum computing) → Turing Machine → Vom Neumann Architecture
2) NON-COMPUTABLE → Incompleteness theorem

In the past there was a connection between Cognition and computability
Our cognitive architectures will be based on Vom Neumann Architecture

COMPUTABLE —implemented→ Turing Machine → Jacopimi-Böhm theorem
Turing Machine —using software→ Vom Neumann Architecture

Computable requires:
1) Do procedures (sequence of statements)
2) Jumps among points in program
3) Loops

OSS: without one or more requirements I can solve only a subset of problems

There are three different definitions for **design pattern:**

1. A general solution to a **common** problem in architectural design
2. A **formal** description or template for how to solve a design problem, which can be used in different situations
3. A **recurring** solution to a standard, well-defined problem in software design

Design patterns are useful to prevent issues which may emerge later. They enforce reliable architecture choices by providing tested development solutions. They improve data flow structure and its readability.

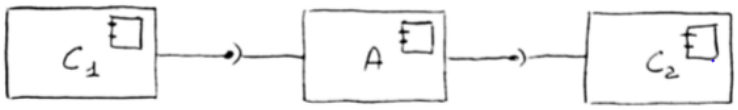
When used together, design patterns constitute a language to define the structure of a cognitive architecture, this language is not formal, but is a collection of techniques and best practices.

There are three different patterns:

1. **Communication patterns** are related to what happens between two or among cognitive components
2. **Component patterns** are related to the internal behaviour of components
3. **System architecture patterns** refer to how a great number of components interact to form sub-systems, typically meant at specific behaviours and/or contexts

Adapters

They are classified as components pattern. They translate one component interface into an interface compatible with another component (E.g. they translate calls to its interface to calls to the original component).
Adapters allow components, which normally would not work together for interface compatibility issues, to do it.
The result is achieved by providing "client" components with the adapted interface, while internally it is used the original interface. Adapters can be used on a data path; in this way it is not necessary to modify components with slightly different interface representing the same information in different ways.



Note: The amount of code needed to achieve the adaptation should be small to minimise the delay and jitter of inter-component communication. They should react quickly to reduce delay that can lead to instability.

Remark

A bit of overhead at the communication level must be considered, even if ROS can be considered as a negligible overhead. Adapters do not have internal "state" and they react synchronously to inputs.

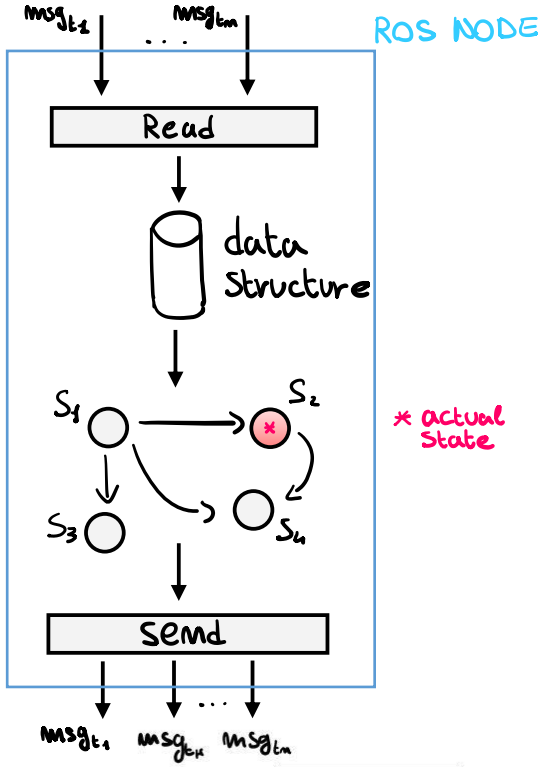
Computational

Classified as component pattern. It follows the sequent behaviour:

1. It collects all inputs using "data ports"
2. Determines its own computational state (state machine)
3. Performs one run of the main function "on_update()"
4. Publishes all the outputs on "data ports"
5. If needed, it updates again its computational state

An important note is that the data structure can have different implementations and different algorithms (such as ML program, message sorting, etc.). Moreover, since this component is implemented as a state machine it is possible to have a state dependent behaviour and also branching given by the type of messages received.

Overall, this block processes inputs and publishes outputs according to its current state. This component must be seen as an **atomic** activity.



Example

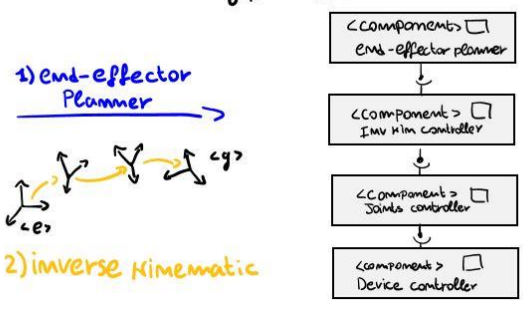
USE CASE/ A MOTION CONTROL STACK COMMANDING THE MOTION OF A ROBOT ARM FROM THE REFERENCE POSE IN A CARTESIAN REFERENCE FRAME TO THE REFERENCE MOTOR CURRENTS.

ENDEFFECTORPLANNER/ THE MAIN CONTROLLER GENERATING THE SEQUENCE OF POSES OF THE END-EFFECTOR STARTING FROM A REFERENCE POSE IN CARTESIAN SPACE.

INVERSEKINEMATICSCONTROLLER/ CONVERTS EACH POSE IN THE SEQUENCE TO JOINT ANGLES FOR ALL ROBOT JOINTS.

JOINTSCONTROLLER/ IT HAS AN INTERNAL LOOP TO SET THE DESIRED POSITION OF ANY SINGLE AXIS, I.E., THE SET POINT, SUCH AS A PID OR A PASSIVE IMPEDANCE CONTROL ALGORITHM, FOR EACH JOINT ANGLE.

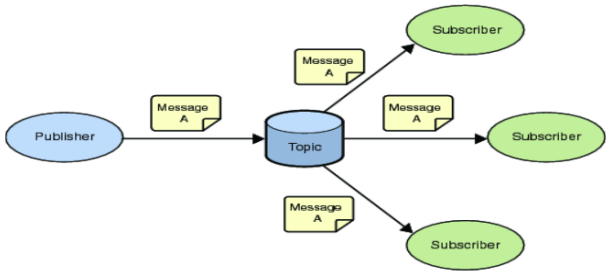
DEVICECONTROLLER/ THE APPROPRIATE MOTOR CURRENT MUST BE SENT TO THE HARDWARE THROUGH A LOW-LEVEL INTERFACE, E.G., A CAN-BASED FIELDBUS.



Oss: Surely a unique sequential code is computationally more efficient with respect to using modules, but in this way, it is possible to have a better structure, clearer and simpler to modify.
Oss: In this simple version the components run at the same frequency, but this may not be always true. Moreover, there can be overhead given by inter-component communication (E.g. delays).

Publish-Subscribe

It is classified as a communication pattern. In this case there is one component (the **Publisher**) that is responsible for generating data and one or more components need to access those data (the **Subscribers**). The behaviour is:



1. The subscriber sends a request to the publisher to "express its desire" to receive data associated with a topic
2. Every time a message on a topic is published, a 1-way message containing the data is sent to all the subscribed components
3. The data are pushed by the publisher and not pulled by the subscribers

Important to note that there is a components de-coupling. In fact, the publisher does not need to know whether there are subscribers to a given topic; moreover, there is no assumption about the frequency of the data update.
In conclusion, this is an asynchronous communication and this pattern does not guarantee any coordination among the components involved in the communication.
Oss: Typically, this communication is used at "low levels" for sensors and components, whereas at "high levels," it is preferable to utilize services.

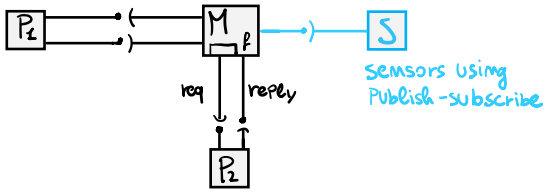
Request-process-reply (services)

It is classified as a communication pattern. This one allows to do branching and launching subroutines. There is one component (consumer) which invokes a computation by sending a request to another component (provider).
The behaviour follows these steps:

1. Send request message, that may or may not contain data (if not contain data then is called signal)
2. The request of the consumer may be either blocking (synchronous call), or non-blocking (asynchronous call)
3. The service provider executes a method of function (callback) associated with the specific computation upon the reception of the request
4. The provider sends a reply back if and only if the associated callback has a "return value"

A consumer can invoke a computation if and only if they have the same service contract, meaning that they agree on the kind of messages used. The provider does not know how many consumers can use its computations and the communication is always initiated by the consumer. The computation provider is expected to **react synchronously** to the request.
Oss: a component can be at the same tame both a consumer and a provider.

There are four different use cases: information retrieval, behaviour tuning, on-demand processing and send commands.
Information retrieval: typically, there is a software component working as memory (useful when there are a few modules that have to access the same data). Other processes use services to request data from the memory. Most often the publish-subscribe is a better alternative, especially when high frequence calls are used. This modality can be used to retrieve information which is updated frequently by the provider but are required sporadically by the consumer.



Services
- High importance data
- Bigger priority
- Slow and Heavy

VS

Publish-Subscribe
- Use UDP Protocol (Possibility to lose data) → Low importance data
- Fast and light

Behaviour tuning: a component may want to set a parameter of another component (E.g. a client which set a parameter server in ROS). In this case, the component wants to receive an ack that the change has been successfully applied.

On demand processing: the computation provider is passive and usually silent. The only goal of the provider is to process data received by the consumer and send back the processed results.

Send commands: the computation consumer's goal is to execute different callback functions exposed by different providers in a logic sequence managed by the consumer. The request message may contain command parameters. An example of this, is the module that control the switch among behaviours in the example of the mobile robot.

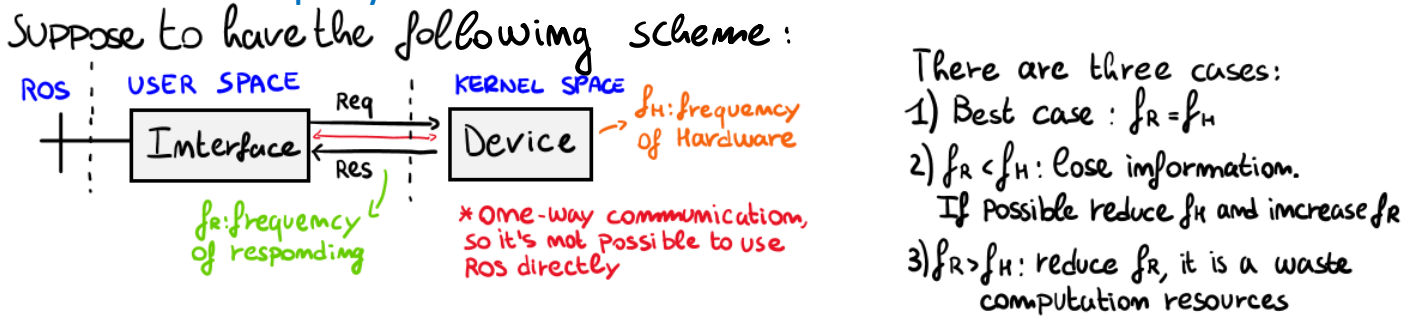
Note: be aware on the use of this pattern, synchronous calls should be avoided in cognitive components which are executed periodically, since they may be blocked for a non-predictable amount of time. Any blocking function is characterised by the risks related to deadlocks.

Sensor device/interface

It is a component pattern and it is one of the most typical pattern in robotics, specifically to integrate sensory data. Robots are equipped with many sensors, each one is associated with a driver developed by the hardware manufacturer. Each driver may access the data through different communication ports, usually in a blocking fashion. Most device drivers are not meant at being multi-client (so cannot directly send messages on ROS).

The behaviour is simple, sensor device components may be provided with many configuration parameters. Typically, they are periodic and they work at the maximum frequency allowed by the controlled device. Moreover, they should provide diagnosis information about the device status.

Observations about frequency and data distribution



Oss: considering more than one device that is sending information, there are some feasible solutions to consider.
Case 1: one interface for each device, this is useful to have different flows of data and different topics.
Case 2: one interface for all the devices, this is useful to have synchronised data.

There are three primary architectures: hierarchical, reactive (behaviour-based) and hybrid reactive-deliberative.

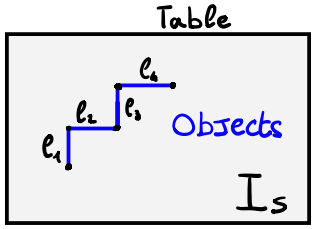
Hierarchical architectures

Hierarchical architectures, such as the **Sense-Plan-Act** scheme, form the basis of our discussion. Before the advent of STRIPS, initial attempts at making inferences and deductions relied on First Order Logic (FOL). Within this framework, **situations** were utilized to describe changes in FOL, reflecting the world as a sequence of discrete snapshots. Notably, predicates and functions capable of temporal variation, known as fluents, necessitated the inclusion of a situation

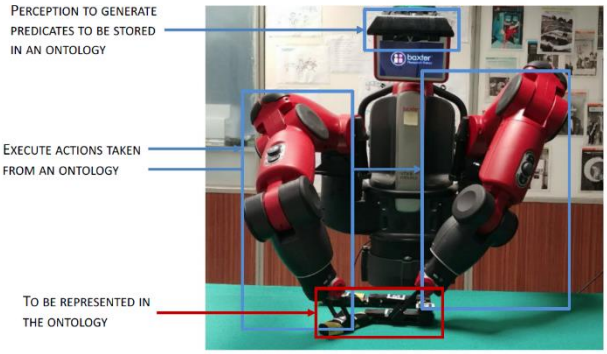
argument. For instance, the predicate at(?r – robot, ?a – area) evolved into at(?r – robot, ?a – area, s_i), where s_i denotes a specific situation or temporal point.

The language-specific result() operator specifies how the world changes from one situation to the next one e.g., result(forward, s_0) = s_1, result(turn(right), s_1) = s_2, result(forward, s_2) = s_3. The states are represented in a recursive way using results.

Subsequently, **STRIPS** emerged as a more efficient means of representing the world. The problem was then how to retrieve the truth value of the predicates and how to execute the actions.



Supposing to assign predicates to the objects using perception and scene analysis



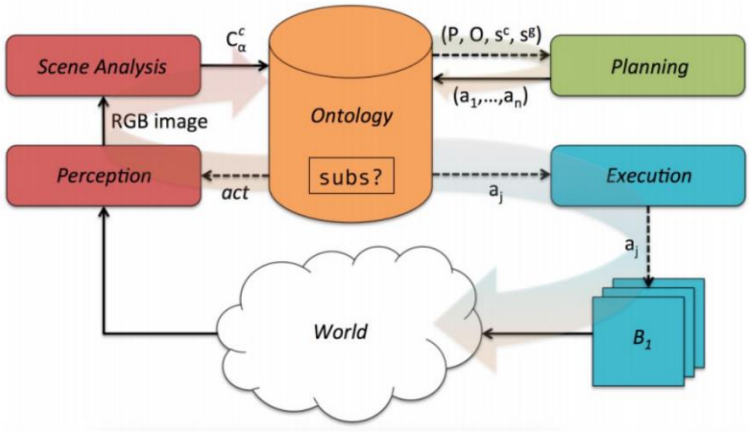
Them, we have that:

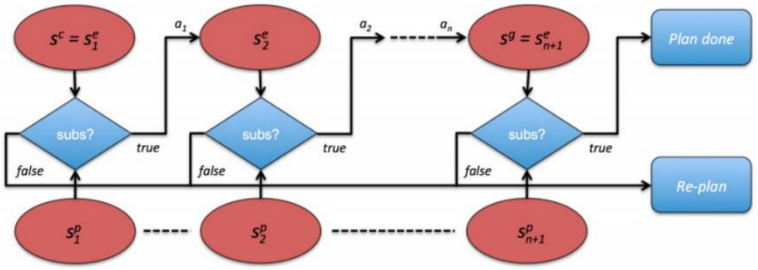
$$I_S = \begin{bmatrix} V(l_1) \\ H(l_2) \\ V(l_3) \\ H(l_4) \end{bmatrix}$$ (using closed world assumption)

The goal state can be: $G_S = \begin{bmatrix} V(l_1) \\ H(l_2) \\ H(l_3) \\ H(l_4) \end{bmatrix}$ and so the actions to take are:

I_S → Rotate(l_3, clockwise) → S_1 → Rotate(l_4, counterclockwise) → G_S

These actions are executed by the arms.



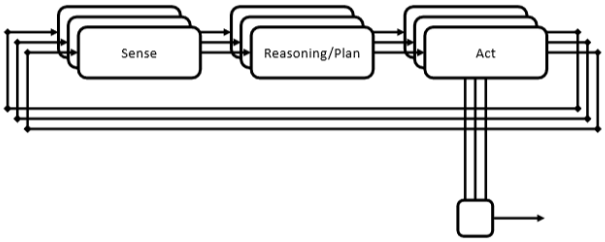


Subs is a function that takes the actual and the planned one checking if they are the same

Important notes are that planning components are look-ahead modules. They differ from problem solvers in their use of more flexible, architecture-friendly, representation of states, actions, goals, and plans. STRIPS-like languages capture much of the expressive power of situation calculus.

Bio-inspired architectures

Recalling that sense-plan-act (hierarchical) architectures are purely sequential and they require a complex representation of states, actions, goals and plans. Also, they do not capture "biologically inspired" cognitive capabilities but are very software engineering oriented. We can distinguish two architectures: **reactive**, which don't have any reasoning, or **behaviour-based**.



The goal is to develop some possible relationships between biological behaviour and sense-plan-act behaviours in robots. This can be done by connect the ideas coming from ethology to computational theory

Reactive architectures

A straightforward illustration can be found in Valentino Braitenberg's vehicles. These machines possess a basic internal structure and will either move towards or away from light sources depending on input from their sensors and motors.

VEHICLE 1: "APPROACH"/

1/ THE SPEED OF THE MOTOR (RECTANGULAR BOX IN THE TAIL) IS CONTROLLED BY A SENSOR (HALF CIRCLE AT THE FRONT).

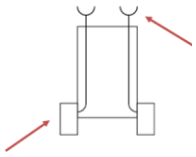
2/ MOTION IS ALWAYS FORWARD, IN THE DIRECTION OF THE ARROW.

VEHICLE 2A: "FEAR"/

1/ THE VEHICLE SPENDS MORE TIME IN AREAS WITH LESS STIMULATION.

2/ IT SPEEDS UP WHEN EXPOSED TO MORE STIMULATION.

3/ IF THE STIMULATION IS DIRECTLY AHEAD, IT MAY HIT THE SOURCE, OTHERWISE, IT WILL TEND TO TURN AWAY FROM THE STIMULATION.

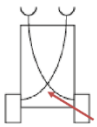


VEHICLE 2B: "AGGRESSION"/

1/ IF THE SENSOR-MOTOR CONNECTIONS ARE CROSSED, THE BEHAVIOUR CHANGES.

2/ IF THE STIMULATION IS DIRECTLY AHEAD, THEN THE VEHICLE MOVES DIRECTLY TOWARDS IT AS BEFORE.

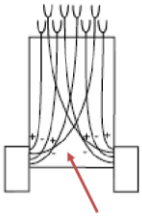
3/ IF THE STIMULATION IS ON ONE SIDE, THEN THE VEHICLE TENDS TO VEER TOWARDS IT WITH INCREASING SPEED.



VEHICLES 3C: "MULTIPLE VALUES"/

1/ MULTIPLE SENSORS ARE ATTACHED TO EACH WHEEL, EITHER EXCITATORY (+) OR INHIBITORY (-).

2/ USED SENSORS INCLUDE: LIGHT, OXYGEN, TEMPERATURE, CONCENTRATION AND AMOUNT OF ORGANIC MATTER.



Many other examples of system without "reasoning" (only hardware) can be done by changing the pattern. This highlights the importance of connection between sensing and actuation, linear and non-linear interaction, inhibition and multiple sensor stimulation.

Behaviour-based architectures

The question that arises is whether the presented vehicles are intelligent, and more generally, what intelligence is. The working definition of intelligence will be: "*the ability to improve an animal or human's likelihood of survival in the real world, and, where appropriate, to compete or cooperate successfully with other agents to do so.*"

An important observation is about the use of the word agent: an entity that is self-contained, independent, has its own "brain", can interact with the world to make sense of what is happening or to make changes

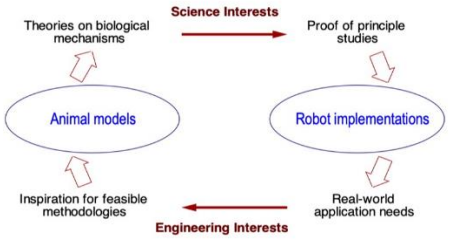
TINBERGEN'S FOUR QUESTIONS/

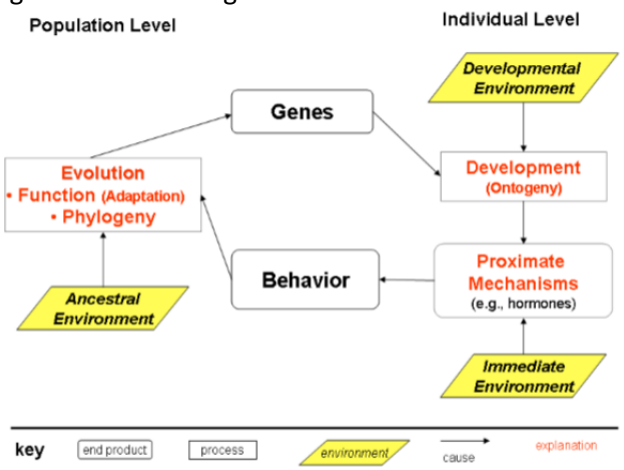
FUNCTION/ HOW DOES A BEHAVIOUR AFFECT THE ANIMAL'S CHANCES OF SURVIVAL AND REPRODUCTION? WHY DOES THE ANIMAL RESPOND THAT WAY INSTEAD OF SOME OTHER WAY?

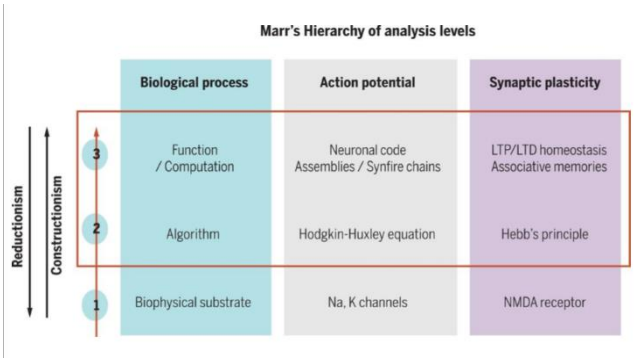
CAUSATION/ WHAT ARE THE STIMULI THAT ELICIT THE RESPONSE, AND HOW HAS IT BEEN MODIFIED BY RECENT LEARNING?

DEVELOPMENT/ HOW DOES THE BEHAVIOUR CHANGE BY AGE, AND WHAT EARLY EXPERIENCES ARE NECESSARY FOR THE ANIMAL TO DISPLAY THE BEHAVIOUR?

EVOLUTIONARY HISTORY/ HOW DOES THE BEHAVIOUR COMPARE WITH SIMILAR BEHAVIOUR IN RELATED SPECIES, AND HOW MIGHT IT HAVE BEGUN THROUGH THE PROCESS OF PHYLOGENY?







LEVEL 1/ EXISTENCE PROOF OF WHAT CAN/SHOULD BE DONE (ETHOLOGY, HUMAN BEHAVIOUR).
LEVEL 2/ DECOMPOSE THAT "WHAT" INTO PROCESSES WITH WELL-DEFINED INPUT-OUTPUT RELATIONSHIPS (BLACK BOXES – COMPONENTS, FLOW CHARTS – FINITE STATE MACHINES).
LEVEL 3/ IMPLEMENT ALL THE PROCESSES (HW VS SW).

Behaviour: is a mapping of sensory inputs to a pattern of motor actions that are used to achieve a task. Behaviours can be of three types:

- **Reflexive**: stimulus-response and hard-wired
- **Reactive**: are learned and compiled down
- **Conscious:** refer to deliberate thought

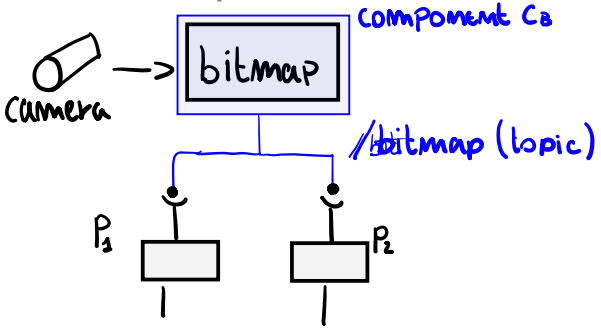
Moreover, we can distinguish between: **reflexes,** rapid, automatic and involuntary responses triggered by environmental stimuli. The response persists and is correlated to stimulus' strength. **Taxes,** behavioural responses orienting an animal towards or away from a stimuli; they occur in response to visual, chemical, mechanical, electromagnetic phenomena. **Fixed-action patterns**, are temporally-extended response patterns triggered by a stimulus, that last longer than the stimulus itself; they may be motivated. Finally, there could exists an **innate releasing mechanism**, a combination of stimuli, which elicit a specific response to a particular biological or environmental situation. To do so could be present a releaser, a signal or Boolean triggered by a stimulus and controlling the activation of a certain behaviour.

Example: Toad's architecture

TOAD'S ALGORITHM/
1/ IF THE TOAD *SEES* A PREY, IT *ORIENTS* TOWARDS IT.
2/ IF IT *DETECTS* A PREDATOR, IT *JUMPS* TOWARDS THE DARKEST REGION IT CAN SEE.

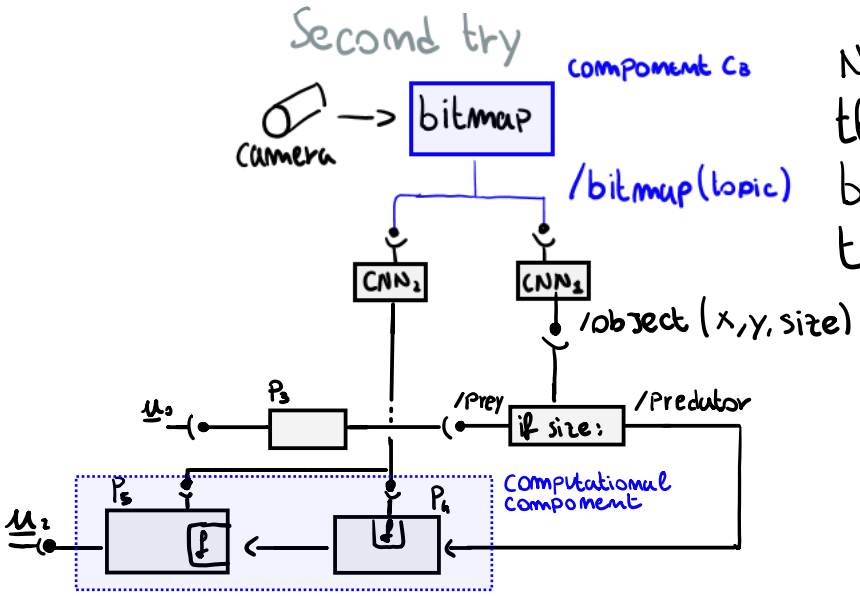
1) Camera (s) → sensor
2) Images → data
3) P_1: bitmap (Prey)
4) P_2: bitmap (Predator)
5) P_3: turn: prey.x, prey.y
   ↳ motion ($\underline{u}_0$)
6) P_4: turn 2: dark.x, dark.y
   ↳ motion ($\underline{u}_2$)
7) Jump: ? → $\underline{u}_3$

First try



- Using C_B (Publisher-Subscriber) cause duplicated memory, too fast to process a full bitmap
- Blob detection to find a prey
- The predator is only a trigger, while the necessary data is the dark region
  ↳ The model is not correct

MM



NOTE: The if is a releaser that chooses the behaviour according to the prey dimensions

Toad's hardware: the retina has about 1 million receptors connected to thousands of neurons. These neurons form a 4-layers map and each layer operate in parallel and is specialised in different kinds of object recognition

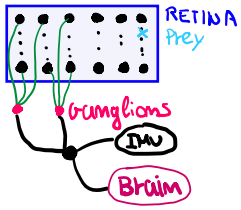
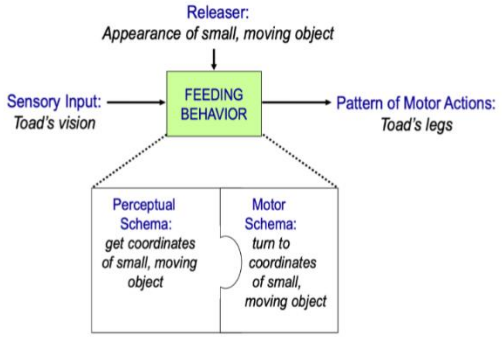


Photo receptors do convolutions
The connections are clustered in ganglions
The IMU is used to control the vision during head motion

Brain Map: to change behaviour

m_1 is the firing neuron. The frequency and strenght follow a Gaussian

This brain surface will fire only when a prey is detected, another surface will be used for the predator. The dimension of the Gaussian depends on the neuron connections and this connections depend on the behaviour selected



IN SUMMARY/

1/ BEHAVIOURS TAKE SENSORY INPUTS AND PRODUCE MOTOR ACTIONS AS OUTPUT.

2/ BEHAVIOURS CAN BE REPRESENTED AS SCHEMAS.

3/ BEHAVIOURS ARE ACTIVATED BY RELEASERS.

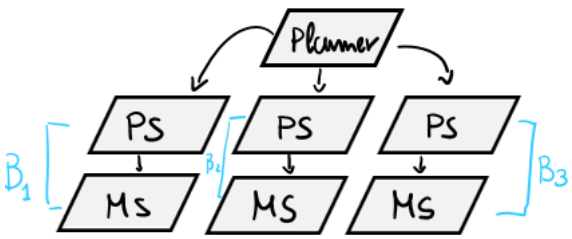
4/ THE TRANSFORMATIONS OF SENSORY INPUTS INTO MOTOR ACTION OUTPUTS CAN BE DIVIDED INTO TWO *SUB-PROCESSES*: A PERCEPTUAL SCHEMA AND A MOTOR SCHEMA.

OPEN ISSUES/

1/ HOW DO WE SET-UP AND/OR LEARN NEW SEQUENCES OF BEHAVIOURS? (SEE AI4RO2)

2/ WHEN ARE EXPLICIT KNOWLEDGE REPRESENTATION AND MEMORY NECESSARY? (SEE THE DISCUSSION ABOUT BRAITENBERG VEHICLES).

3/ HOW DO WE RESOLVE CONFLICTS BETWEEN CONCURRENT BEHAVIOURS?

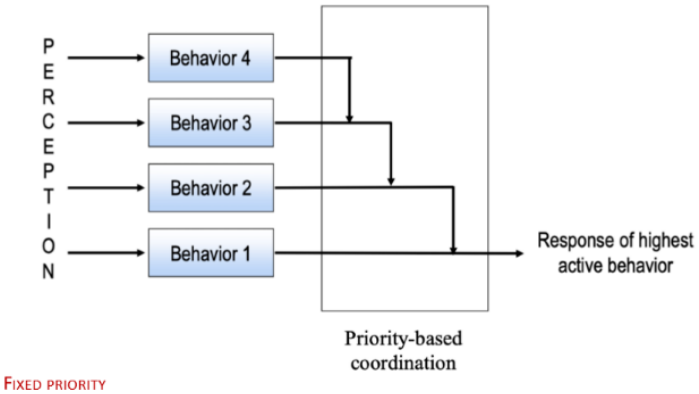


Many behaviours can be active at the same time this is possible thanks to the planner (coordinator)

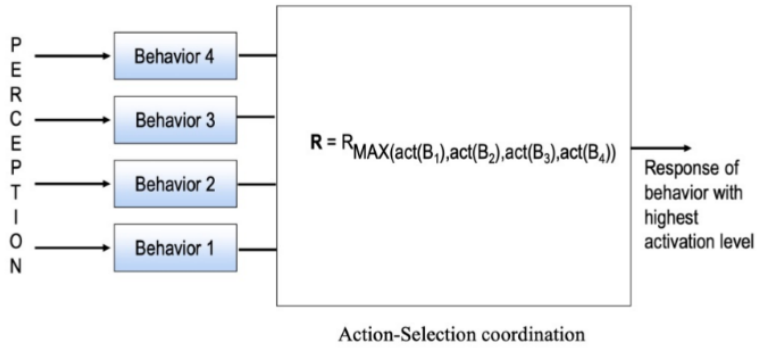
Concurrency of robot behaviours

There are two possible classes:

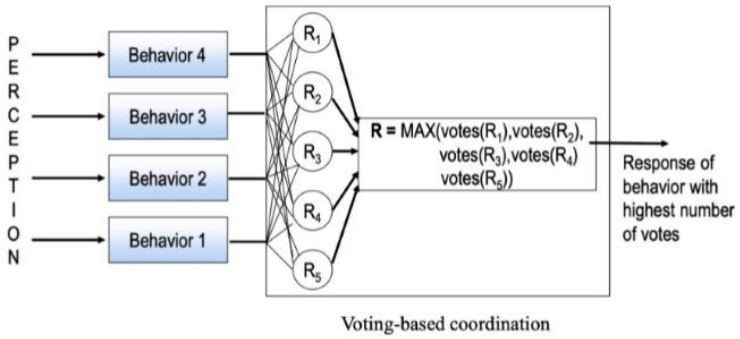
- **Competitive methods** ("Winner takes all"): fixed priority, action selection, vote generation. In this situation only one behaviour will be active (simples approach)
- **Cooperative methods**: behavioural summation/fusion. Multiple behaviour active at the same time.



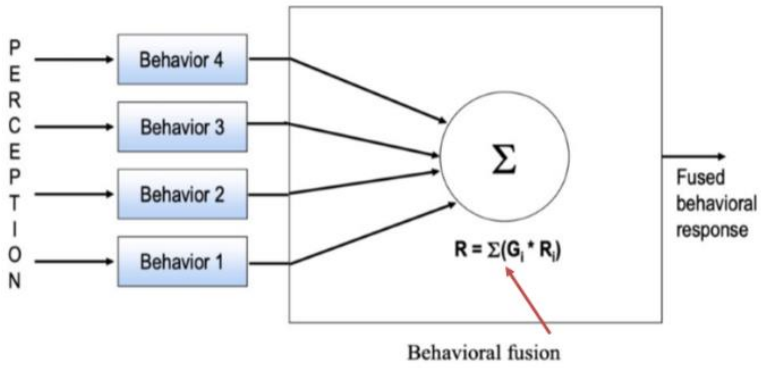
FIXED PRIORITY



ACTION SELECTION



VOTE GENERATION

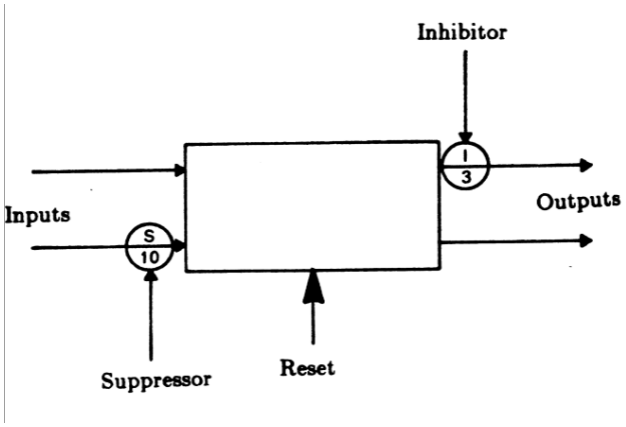


BEHAVIOURAL SUMMATION/FUSION

Subsumption architecture

The hierarchical solution doesn't allow to parallelize tasks and so the agent cannot be "intelligent. To solve this problem, the idea is to create an architecture where tasks are parallelised and higher level behaviours subsume lower level ones. Moreover, behaviours in the lower levels have no awareness of those in the upper levels; the architectural topology is defined by a fixed priority hierarchy. To coordinate various behaviours is necessary to use two systems:

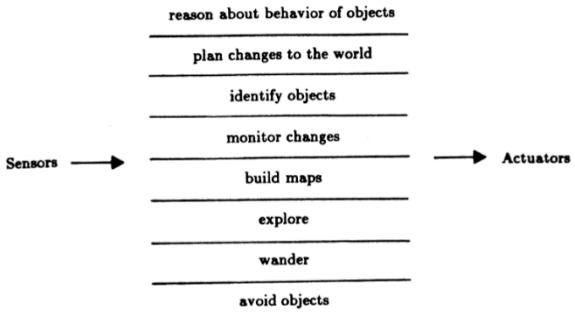
- **Inhibition:** is used to prevent a signal (message) being transmitted to a behaviour
- **Suppression:** prevents the current signal from being transmitted by a behaviour to the actuators, and replaces it with a suppressing signal



Oss: Reset is a releaser to activate and deactivate the behaviour

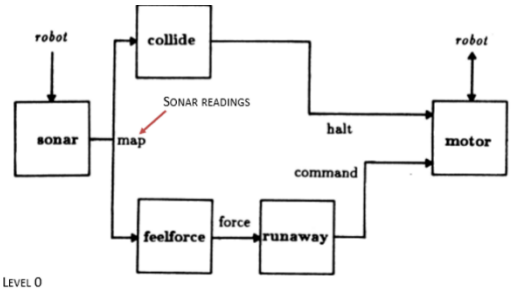
Oss: These three components are built in hardware

Example: subsumption architecture



So, the idea is to take the diagram on the left and build levels of competence from the bottom to the top, by adding functionalities to previous levels

Level 0: avoid obstacles

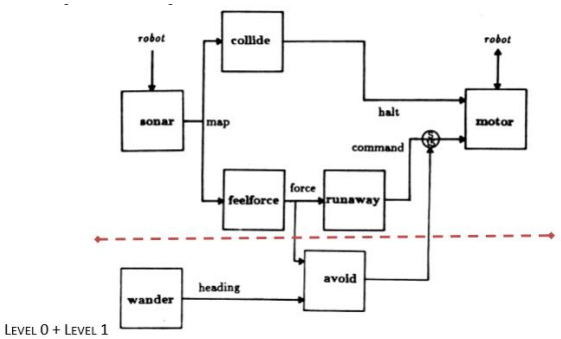


LEVEL 0

This level use sonar readings to build the following modules:

- Collide module: determines if there is an imminent collision and sends an halt command to the motors
- Feelforce module: treats the sonars as repulsive forces acting on the robot and compute a force vector
- The previous module sends the resultant force to the runaway module, which computes motor commands in accord with the perceived force

The resulting level 0 behaviour induces the robot to move away from obstacles



LEVEL 0 + LEVEL 1

- Wander module selects an arbitrary heading
- This heading plus the force are the input to the avoid module that computes motor commands combining the desired heading with the necessity of avoiding nearby objects
- To prevent the runaway module taking over the control of the avoid module, this last one suppresses the output of the runaway module

The resulting behaviour of level 0 + level 1 is a robot which wanders avoiding obstacles

Note: The suppressor can be represented as a service, which reply decides the module that will go. In general, publish-subscribe is used on horizontal level, while services are used in vertical levels communication

Advantages: subsumption architectures can be compiled directly on FPGA, then they support parallelism, given that each behavioural layer can run independently and asynchronously, finally costum behaviours can be created for specific task-environment pairs.

Neither strength nor weakness: robustness, in fact the architecture in a real robot, but is often hard to implement; this is also connected to the length for the development

Weakness: priority-based coordination mechanism and hard-wired aspects limit its adoption. Furthermore, behavioural reuse is not easy in practice

Human-Robot Collaboration

In Human-Robot Collaboration (HRC), humans and robots are expected to work together to reach a common goal and share their workspace. It becomes necessary to design robot cognitive capabilities to comply with human-centred performance metrics and exchange information between the two.

One example of this collaboration is the cooperative assembly. This problem is difficult to model, given that the assembly order may have variations; moreover, in this situation, humans and robot engage in a turn-taking process, in which the robot is expected to assist and adapt to the operators. In this context, PDDL-based planners cannot be used.

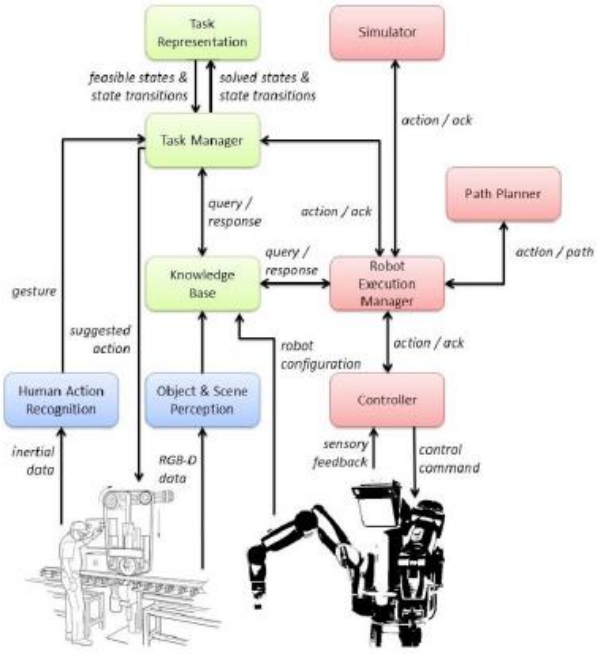
The principal aspects to consider are:

- **Task level flexibility**: human operators and robot cooperate on similar tasks, without requiring a different task representation. This should be transparent to human operators and so the structure of tasks should abstract perceptual variabilities and uncertainties
- **Team level flexibility:** human operators should not be forced to follow a strict, predefined sequence of operations. They should be allowed to decide which actions to perform on the fly, if those actions do not jeopardize the whole cooperation goals. Robots must then decide which actions to perform next or react appropriately to human actions
- **Intelligibility:** while cooperation unfolds, human operators should be capable of understanding robot (future) actions. This may imply the use of a symbolic, linguistic level of communication. This level should integrate action planning (the subject of communication) from motion planning and control
- **Naturalness:** literature shows that natural and efficient cooperation is possible only with a reasoned trade-off between cooperation objectives (i.e., an assemblage) and the human or robot degrees of autonomy when the task is only partially well-defined
- **Adaptability:** collaborative robots need to detect and classify meaningful actions carried out by human operators and adapt to them
- **Decision making:** collaborative robots must plan goal-oriented action sequences, estimate the execution of planned, future behaviours, integrate such plan with the outcomes of human operators' actions. Planning s

REPRESENTATION LEVEL/ MAINTAINS KNOWLEDGE ABOUT ALL POSSIBLE STATES AND TRANSITIONS MODELLING COOPERATIVE TASKS.

PERCEPTION LEVEL/ DEALS WITH PERCEIVING THE ROBOT WORKSPACE, THE ENVIRONMENT, AND HUMAN OPERATORS.

ACTION LEVEL/ IMPLEMENT A LIGHT FORM OF TMP WITH IN-THE-LOOP ROBOT MOTION SIMULATION (USING RRTS).

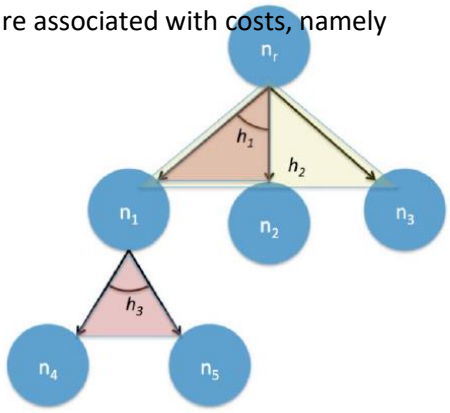


AND/OR graphs

AND/OR graphs are meant at representing problems or procedures to follow, which can be decomposed in sub-problems (as sub-graphs), as well as the logic relationships among these sub-problems (i.e., the interconnectivity of the graph). There exists a **root node** representing the solution and "solving" the problem, or "following" the procedure means **traversing** the graph from **leaf nodes** to the root node according to its structure. This graphs can take into account limited forms of non-determinism or uncertainty via the availability of various paths leading to the solution.

Propositional AND/OR graphs

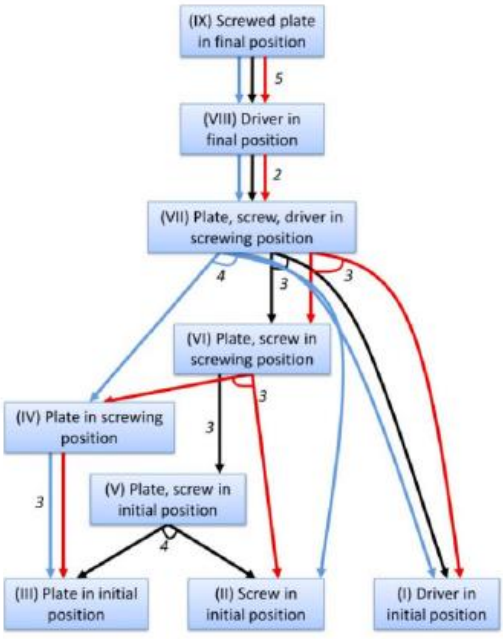
An AND/OR graph *G(N,H)* is defined as a data structure where *N* is a set of $n_1, \dots, n_{|N|}$ nodes and *H* is a set of $h1, \dots, h_{|H|}$ hyper-arcs. Nodes in *N* define reachable states, whereas hyper-arcs in *H* define transition relationships among states. Each hyper-arc $h_i \in H$ defines many-to-one transition relationship between a set of *c* child nodes $c(h_i) = (n_{h_{i,1}}, \dots, n_{h_{c,1}})$ and a parent node $p(h_i) = n_k$. The child nodes of a hyper-arc are in logical *and*, while different hyper-arcs with the same parent node are in logical *or*. Both nodes and hyper-arcs are associated with costs, namely $w_{n_1}, \dots, w_{n_{|N|}}$ and $w_{h_1}, \dots, w_{h_{|H|}}$.



In FLEXHRC, each hyper-arc h_i models a set A_i of actions $a_1, \dots, a_{|A_i|}$, and an action $a_j \in A_i$ can be assigned either to a human or a robot. If the order in which to execute actions in A_i is important, A_i is defined as an *ordered* set such that $A_i = (a_1, \dots, a_{|A_i|}; \preceq)$, i.e., a *temporal* sequence is assumed in the form $a_1 \preceq a_2 \preceq \dots \preceq a_{|A_i|}$. Initially, all actions $a_j \in A_i$ are labeled as *unfinished*, i.e., $\neg e(a_j)$. When an action a_j has been executed, it is labeled as *finished*, i.e., $e(a_j)$. For all actions in A_i, if $e(a_j)$ holds then h_i is *done*, and the notation $d(h_i)$ is used. If an ordering is induced, $d(h_i)$ holds if and only if also the temporal execution sequence is satisfied.

A node can be either solved or unsolved. A node $n_k \in N$ is *solved*, specified with $s(n_k)$, if there is at least one hyper-arc $h_i \in H$ such that $p(h_i) = n_k$, $d(h_i)$ holds, and for all $n_l \in c(h_i)$ it holds that $s(n_l)$. A node n_k is *unsolved* otherwise, and specified with $\neg s(n_k)$. *Leaves* in G, i.e., nodes n_k for which there is no hyper-arc $h_i \in H$ such that $p(h_i) = n_k$ (as is the case of n_2, n_3, n_4 and n_5 in Figure 3.1), are initialized as solved or unsolved, depending on the initial state of the cooperation. An AND/OR graph G is traversed from leaves to the *root* node $n_r \in N$. When $s(n_r)$ holds, then G is *solved*, i.e., $s(G)$. During the traversal procedure, a node $n_k \in N$ is *feasible*, i.e., $f(n_k)$ holds if there is at least one hyper-arc $h_i \in H$ such that $p(h_i) = n_k$ and for all nodes $n_l \in c(h_i)$ it holds that $s(n_l)$. In this case h_i is labeled as *active*, i.e., $a(h_i)$. Otherwise, n_k is *unfeasible*, i.e., $\neg f(n_k)$. Leaves that are not solved at the start of the cooperation are initialized as feasible. While the cooperation unfolds, there is a set of active hyper-arcs $H_a \subset H$ in G.

We define the *graph representation state* S_G as the set of all feasible nodes and active hyper-arcs in G, i.e., possible action alternatives for the human or the robot. A cooperation path P in G is defined as a sequence of visited nodes and hyper-arcs. Each cooperation path is associated with a *traversal* cost, namely $cost(P)$, which defines how effortful following P is, on the basis of the involved nodes and hyper-arcs weights. A cooperation model M is a ordered sequence of $|M|$ actions, such that $M = (a_1, \dots, a_{|M|}; \preceq) \subset S_G$, corresponding to an allowed cooperation path in G. At any given time instant, there is one current cooperation model M_c as well as one current cooperation path P_c.



AND-OR graphs are used to orchestrate the behaviours and give them a structure. In the states there is also an active perception allows to decide the actual state and the "path" to follow.

To solve a general problem, there are two phases:

- **Offline phase:** it is used to prepare all the relevant data and structures for later use. For example, in this phase there is the loading of the AND/OR description, the updating of all node feasibilities, generation of all possible cooperation path and determination the next possible robot actions.
- **Online phase:** this is done in a loop and allow to find the best robot action to reach root node, based on the currently feasible nodes and human operator's actions. After that, node feasibilities and cooperation paths are updated

Summary

There are two possible approaches:

- **Planning** (PDDL-like): request the definition of the domain ($I_s, G_s, A, P_1, \dots, P_n$). This choice is flexible but requires a heavy computation
- **Orchestration approaches:** Use of AND/OR graphs, HTN or behaviour trees. These are les flexible but requires less computation

Final recap