# Intelligence of an autonomous robot
# Behaviour, Reasoning and Planning

Corrado Santoro

**ARSLAB - Autonomous and Robotic Systems Laboratory**
Dipartimento di Matematica e Informatica - Università di Catania, Italy
`santoro@dmi.unict.it`

Robotic Systems

# Behaviour, Reasoning and Planning

- Models and algorithms studied till now allow us :
  - To drive the robot's actuators on the basis of a certain kinematic model
  - To consider the constraints of the environment and create proper paths
- However, the various movements must be **coordinated** with the aim of executing more complex actions
- The set of coordinated movements is then part of a **strategy** that makes the robot achieveing a precise goal
- However, given a certain **goal**, **different strategies** may exist: a **planning process** is thus needed to select the strategy which is more suitable in that moment

# Environment and Perception

- The environnment is always dynamic and uncontrollable
- During behaviour programming we are not interested in “control aspects” (such as state variables, system model, etc) ...
- ... but in aspect like “how the environment is made?” o “what is happening in this moment”
- **Modelling the environment** thus becomes one of the fundamental aspects of robot behaviour programming
- Environment modelling is also strictly tied to the type of **sensors** used

# Perception and Environment State

Perception and Environment State

# Perception and Environment

- A physical environment is pervaded by:
    - **inanimate objects**, unable to perform autonomous actions,
    - **animated objects** (humans, other robots), which are autonomous
- They are characterised by suitable **properties** like *shape*, *color*, *position*, etc.
- Some properties are **immutable** (*shape*, *color*), other could vary during time (*position*)

- **Modelling the environment** implies to define proper **computer entities** (classes, objects, variables, etc.) able to represent the objects of the environment and the related properties
- These informations **must be percepted** by the sensors chosen for the robot

# Case-Study: The Block World

## The Block World

- Let us consider an enviroment populated by solids with different shape, color and position
- Peroperties:
  - **Shape:** *prism, sphere, cylinder*
  - **Color:** *yellow, red, green, black, white*
  - **Dimensions:** *small, big*
  - **Position:** $(x, y, z)$

## Possible representation in C/C++

```
typedef enum { PRISM, SPHERE, CYLINDER } t_shape;
typedef enum { YELLOW, RED, GREEN, BLACK, WHITE } t_color;
typedef enum { SMALL, LARGE } t_size;
typedef struct {
  t_shape shape;
  t_color color;
  t_size size;
  float x, y, z;
} t_block;

t_block my_blocks[....];
```

# Case-Study: The Block World

## The Block World

- If the blocks can be stacked, this characteristic must be modelled:
  - by means of a **boolean function** that uses the coordinates of the blocks, ...

## Possible representation in C/C++

```
typedef enum { PRISM, SPHERE, CYLINDER } t_shape;
typedef enum { YELLOW, RED, GREEN, BLACK, WHITE } t_color;
typedef enum { SMALL, LARGE } t_size;
typedef struct {
  t_shape shape;
  t_color color;
  t_size size;
  float x, y, z;
} t_block;

t_block my_blocks[....];

bool upon(t_block * block1, t_block * block2)
{
   //....
}
```

# Case-Study: The Block World

## The Block World

- If the blocks can be stacked, this characteristic must be modelled:
  - ... or by adding **another property** that rappresents the link between two blocks

## Possible representation in C/C++

```
typedef enum { PRISM, SPHERE, CYLINDER } t_shape;
typedef enum { YELLOW, RED, GREEN, BLACK, WHITE } t_color;
typedef enum { SMALL, LARGE } t_size;
typedef struct t_block {
  t_shape shape;
  t_color color;
  t_size size;
  float x, y, z;
  struct t_block * upon_block;
} t_block;

t_block my_blocks[....];

bool upon(t_block * block1, t_block * block2)
{
  return block1->upon_block == block2;
}
```

# Case-Study: The Block World

## The Block World

- If the blocks are **captured**, also this condition must be modelled:
    - by means of **two arrays**

## Possible representation in C/C++

```
typedef enum { PRISM, SPHERE, CYLINDER } t_shape;
typedef enum { YELLOW, RED, GREEN, BLACK, WHITE } t_color;
typedef enum { SMALL, LARGE } t_size;
typedef struct t_block {
  t_shape shape;
  t_color color;
  t_size size;
  float x, y, z;
  struct t_block * upon_block;
} t_block;

t_block free_blocks[....], captured_blocks[....];
```

# Case-Study: The Block World

## The Block World

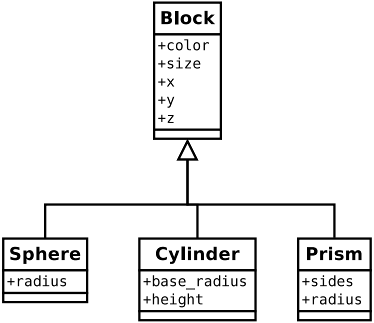
- If the blocks are **captured**, also this condition must be modelled:
    - ... or by adding a **boolean property** that indicates whether the block has been captured

## Possible representation in C/C++

```
typedef enum { PRISM, SPHERE, CYLINDER } t_shape;
typedef enum { YELLOW, RED, GREEN, BLACK, WHITE } t_color;
typedef enum { SMALL, LARGE } t_size;
typedef struct t_block {
  t_shape shape;
  t_color color;
  t_size size;
  float x, y, z;
  struct t_block * upon_block;
  bool captured;
} t_block;

t_block my_blocks[....];
```

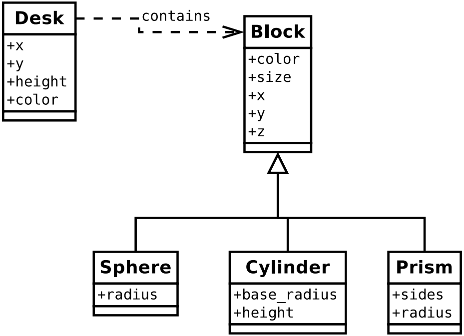
# "Object-Oriented" Representation

- Since the physical enviroment is pervaded by **objects**, a representation widely used is the **object-oriented** one
- Indeed the *object-orientation* was born in 1965, within the AI, just to represent the "worlds" of artificial systems



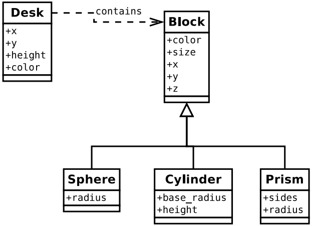
# Ontologies

- The **object-oriented** model can be used to represent not only the things that are in enviroment but also the **relationships** among them
- The result is a **conceptual map**, named **ontology**, that represents the reference enviroment/context in which the robot operates



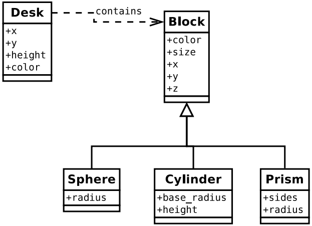
# The Knowledge Bases

- Any representation of the enviroment must be “queryable” in order to allow the robot to reason in some way
- For example, if the robot would **capure the green object plaed in the white table**, the robot must be able to obtain the related (computer) object (instance of `Block`) in order to reach the (physical) object
- Given an ontology (or a similar representation), the information must be organised in a **knowledge base** that can be easily queried



# The Knowledge Bases

- A **Knowledge Base** is thus a data structure that memorises the **knowledge the robot has on the reference enviroment**
- It is made by **concepts** that the robot **believes true** (beliefs)
- It has to support **queries**, **browsing** and **inferencing** new knowledge



# Representation by means of First-Order Logic

- The **logic model** is a classical model to represent the knowledge
- It is based on the definition of **facts** (beliefs) represented as **predicates in first-order logic**
- The inference and query is based on proper **logic formulas**
- There are programming languages, like **PROLOG**, that natively supports this type of model

# Elements of PROLOG

## Elements of PROLOG

# Elements of PROLOG

- **PROLOG** (PROgramming in LOGic) is a programming language based on logic predicates
- Among the classical numeric types, PROLOG introduces the concept of **atom**: if is a **literal** that starts with **lowercase letter**, and is a **indivisible symbol** (if is not a "string"!)
- A literal that starts with **Uppercase letter** is instead a **variable**:
  - `cube` is an atom
  - `Cube` is a variable

# Representation by means of “Facts” (Beliefs)

## Facts or Beliefs

In PROLOG, the knowledge base is made by **facts** represented by atomic formulae

## Example

- The cylinder, prism and sphere are “blocks”:

  ```
  block(cylinder)
  block(prism)
  block(sphere)
  ```

- The cylinder is red, the prism is white, the shpere is black:

  ```
  color(cylinder, red)
  color(prism, white)
  color(sphere, black)
  ```

- The cylinder is upon table 1, the prism and the sphere are upon the table 2:

  ```
  upon(desk1, cylinder)
  upon(desk2, prism)
  upon(desk2, sphere)
  ```

# Representation by means of "Facts" (Beliefs)

## Knowledge Base

To populate the knowledge base the PROLOG **assert** is used:

```
?- assert(block(cylinder)).
true.

?- assert(block(prism)).
true.

?- assert(upon(prism,green_desk)).
true.

?-
```

# Queries and Derived Knowledge

## Queries on the Knowledge Base

The knowledge base can be queried to understand whether a fact is true o false

```
?- block(cylinder).
true.

?- block(cube).
false.

?- block(sphere).
true.
```

# Queries and Derived Knowledge

## Queries on the Knowledge Base

The knowledge base can be also queried using variables **universally quantified**

```
?- assert(block(cylinder)).
true.

?- assert(block(prism)).
true.

?- assert(block(sphere)).
true.

?- block(X).
X = cylinder ;
X = prism ;
X = sphere.
```

# Queries and Derived Knowledge

## Queries on the Knowledge Base

The knowledge base can be also queried using variables **universally quantified**

```
?- block(X).
X = cylinder ;
X = prism ;
X = sphere.
```

The query **`block(X)`** is equal to:

$$\forall x : block(x)$$

# Queries and Derived Knowledge

## Queries on the Knowledge Base

Queries can be combined by using the comma “,” that as the role of **AND** connective:

```
?- block(Obj),color(Obj,Col).
Obj = cylinder,
Col = red ;
Obj = prism,
Col = white ;
Obj = sphere,
Col = black.
false.

?-
```

**All the objects with the related color**

# Queries and Derived Knowledge

## Queries on the Knowledge Base

Queries can be combined by using the comma “,” that as the role of **AND** connective:

```
?- block(Obj),upon(desk2,Obj),color(Obj,Col).
Obj = prism,
Col = white ;
Obj = sphere,
Col = black.
false.

?-
```

**All the objects, with the related color, that are on “desk2”**

# Queries and Derived Knowledge

## Derived Knowledge

By means of first-order clauses, new predicates can be defined to derive new knowledge

Example: let's define the predicate "black object":

```
black_object(X) :- block(X), color(X, black).

?- black_object(Obj).
Obj = sphere.

?-
```

The definition is equivalent to the logic implication:

$$\forall x : block(x) \wedge color(x, black) \Rightarrow black\_object(x)$$

# Queries and Derived Knowledge

## Queries on the Knowledge Base

Queries can be combined by using the comma "," that as the role of **AND** connective:

```
?- block(Obj),upon(desk2,Obj),color(Obj,Col).
Obj = prism,
Col = white ;
Obj = sphere,
Col = black.
false.

?-
```

**All the objects, with the related color, that are on "desk2"**

# Queries and Derived Knowledge

## Derived Knowledge

By means of first-order clauses, new predicates can be defined to derive new knowledge

Example: let's define the predicate "black object":

```
black_object(X) :- block(X), color(X, black).

?- black_object(Obj).
Obj = sphere.

?-
```

The definition is equivalent to the logic implication:

$$\forall x : block(x) \wedge color(x, black) \Rightarrow black\_object(x)$$

Indeed, the PROLOG symbol "**:–**" is somewhat the symbol "$\Leftarrow$"

# Queries and Derived Knowledge

## Derived Knowledge and Negation

Negations can also be defined

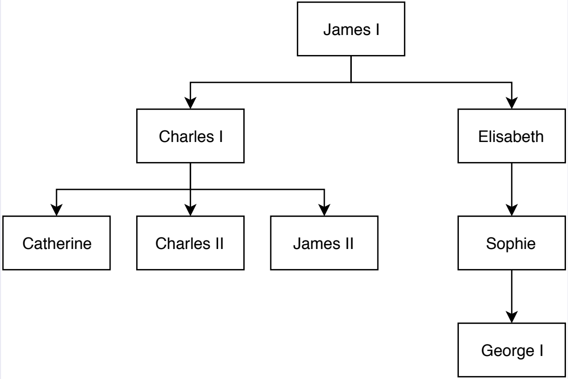
Example: let's define the predicate "free desk": it is **true** if no object is on that desk:

```
free_desk(Desk) :- \+upon(Desk, _).

?- free_desk(desk1).
false.

?- free_desk(desk2).
false.

?- free_desk(desk3).
true.
```

Equivalent to:

$$free\_desk(x) \Leftarrow \nexists y : upon(y, x)$$

# Example: the Genealogic Tree



James I

Charles I

Elisabeth

Catherine

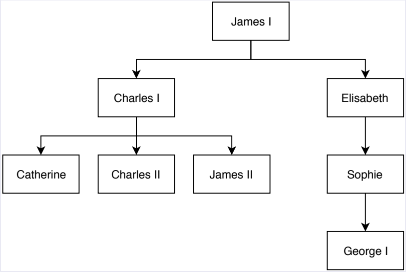
Charles II

James II

Sophie

George I

# Example: the Genealogic Tree



## Facts

- **`male(X)`** → X is a man
- **`female(X)`** → X is a woman
- **`parent(X, Y)`** → X is the parent of Y

# Example: the Genealogic Tree

## Fatti

```
:-
    assert(male(james1)),
    assert(male(charles1)),
    assert(male(charles2)),
    assert(male(james2)),
    assert(male(george1)),

    assert(female(catherine)),
    assert(female(elizabeth)),
    assert(female(sophia)),

    assert(parent(james1, charles1)),
    assert(parent(james1, elizabeth)),
    assert(parent(charles1, charles2)),
    assert(parent(charles1, catherine)),
    assert(parent(charles1, james2)),
    assert(parent(elizabeth, sophia)),
    assert(parent(sophia, george1)).
```

# Example: the Genealogic Tree

## Derived Knowledge

- **`father(X, Y)`** → X is the Y's father:

$$\forall x, y : parent(x, y) \land male(x) \Rightarrow father(x, y)$$

- **`mother(X, Y)`** → X is the Y's mother:

$$\forall x, y : parent(x, y) \land female(x) \Rightarrow mother(x, y)$$

```
father(X,Y) :- parent(X,Y), male(X).

mother(X,Y) :- parent(X,Y), female(X).
```

# Example: the Genealogic Tree

## Derived Knowledge

- **`sibling(X, Y)`** → X is the sibling of Y (X and Y have the same parent):

$$\exists p, \forall x, y : parent(p, x) \wedge parent(p, y) \wedge x \neq y \Rightarrow sibling(x, y)$$

```
sibling(X,Y) :- parent(P, X), parent(P, Y), X \= Y.
```

# Esempio: L'Albero Genealogico

## Conoscenza Derivata

- **`brother(X, Y)`** → X is the Y's brother:

$$\forall x, y : sibling(x, y) \wedge male(x) \Rightarrow brother(x, y)$$

- **`sister(X, Y)`** → X is the Y's sister:

$$\forall x, y : sibling(x, y) \wedge female(x) \Rightarrow sister(x, y)$$

```
brother(X,Y) :- sibling(X,Y), male(X).
sister(X,Y) :- sibling(X,Y), female(X).
```

# Behaviour, Goals, Planning

# Reactivity and Proactivity

- In an autonomous robot, there are two kind of behaviours:
  - **reactive**
  - **proactive**

# Reactivity and Proactivity

## Reactivity

- Execution of a **prefixed** sequence of actions on the basis of the occurence of a **sporadic** event
- It is equivalent to the **instintual reaction** of humans
- Examples
    - Stopping a robot when a distance sensor detects an obstacle
    - Activating an arm when an object is near

# Reactivity and Proactivity

## Proactivity

- **To plan** the **proper** sequence of actions that lead to the achievement of a specific goal
- It is equivalent to the human **reasoning**
- Examples
  - To identify and gather an object
  - To adopt proper maneuvers to avoid a collision

# Reactive Tasks

## Reactive Tasks

# Reactivity and Paradigm “ECA”

## Event-Condition-Action

- Reactive is usually programmed by using the paradigm **Event-Condition-Action (ECA)**
  - **E**, triggering event
  - **C**, condition (predicate) that must be met by event parameters, enviroment state, and system state
  - **A**, action (computation) that must be executed given that the condition is true

## Example: Obstacle Detection

- **Event**, data sampled from the distance sensor
- **Condition**, check on the distance that must be less than a certain threshold
- **Action**, robot stop

# Reactivity and Paradigm “ECA”

## ECA: Implementation

- The implementation of reactive tasks is based, in general, on “callback” functions, involked on the basis of a specific **event**
- The event can be a sensor sampling or the production of data by another task of the control system
- The condition (in general) is a predicate (**if**) applied to the representation of the system/environment (knowledge base)

# Reactivity and Paradigm “ECA”

## ECA: Temporal Constraints

- In some cases, reactive tasks can be characterised by temporal constraints
- Obstacle detection is one of these cases: it needs the event to be processed **tempestively**, or, better, **within a certain temporal “deadline”**
- “Real-time” requirements seen for periodic control tasks are applied (in some cases) also to reactive tasks, which (according to the terminology used in real-time systems) are called **sporadic tasks**

# Proactivity, Goals and Reasoning

# Reasoning and Planning

1. Given a certain **goal**
2. Let’s **percieve** the environment to determinate its **state** and check if the goals has not yet been achieved
3. Then let’s determine the **strategy** (set of actions) that could lead to goal achievement
4. Let’s **Execute** the actions
5. Go back to step 1

# Perception, Representation and Goals

- A **goal** represent a well-defined **environment state** that the robot aims to reach
- A goal can be thus represented as a predicate on the variables that represent the environment (knowledge base)

## The Block World

- **Goal: To capture all the blocks**
- The goals is reached when all the fields `captured` of each element of the array `my_blocks` are `true`

## Possible C/C++ representation

```
typedef struct t_block {
  ....
  bool captured;
} t_block;

t_block my_blocks[....];

bool goal_done()
{
   for (i = 0; i < NUM_OF_BLOCKS; i++) if (!my_blocks[i].captured) return false;
   return true;
}
```

# Goals and sub-goals

- Achieving a goal implies to execute a certain set of actions
- But sometimes a goal hides, in its internals, some sub-goals
- Example: **capturing all the blocks implies capturing them one by time**
- Goal: **to capture all the blocks**
- Sub-goals: *to capture the shpere*, *to capture the cylinder*, *to capture the prism*, etc.
- Sub-goals must not be executed according to a prefixed sequence
- However some sub-goals could not be feasible (example: the cylinder cannot be captured because the prism is on the top of it)
- A planning/selection of sub-goals is thus needed

# Goals with planning and “simple” Goals

## Planning is not always required

- Goals: **to capture all the blocks**
- Sub-goals: *to capture the sphere*, *to capture the cylinder*, *to capture the prism*, etc.

## To capture all the blocks

- **Planning** the proper sequence of sub-goals, or
- **Selecting**, one by one, the most opportune sub-goal

## To Capture X

- **To execute** the proper sequence of actions to capture object X

# Goals that do not require planning

## Plans

- These goals are made of a sequence of actions, each corresponding to the activation of an actuator (arms, wheels, etc.)
- The execution of a sequence often implies to wait for the completion of each action (success) or that the condition the impedes the action's success (failire) are detected
- The goals that have these characteristics are often called **plans** (piani)
- From the implementation point of view, plans can be thought as the sequence:
  - function call 1
  - if/switch-case on the outcome of call 1
  - function call 2
  - if/switch-case on the outcome of call 2
  - ...
- The model is thus similar to a **finite-state machine, FSM**

# Plans

## Example: the Block World

```
bool gather_object(const char *object_type)
{
   float x,y,z;
   if (!knowledge_base.get_object_position(object_type, &x, &y, &z) {
      // uh? I don’t␣know␣object’s position, let us try to find it
      if (!camera_sensor.detect(object_tpe, &x, &y, &z))
          return false; // cannot detect object, the sub-goal fails
      knowledge_base.update_object_position(object_type, x, y, z);
   }

   robot.drive_to(x, y, z);
   while(true) {
      if (robot.position_reached(x,y,z)) break; // we got the position
      if (robot.motion_blocked()) return false; // goal failed
      if (timeout()) return false;              // goal failed
   }

   robot.pick_the_object();
   while(true) {
      if (robot.object_picked()) break; // we got the object
      if (timeout()) return false;      // goal failed
   }

   knowledge_base.mark_object_picked(object_type);
   return true;
}
```

# Reasoning and Deliberation

## Goals and Planning

- In the behaviour of a robot, often a **goal** requires the choice among different **plans** $p_1, ..., p_n$
- **AND**-Plans: achieving the goal implies to execute all the plans $p_i$, but their order is chosen at run-time
- **XOR**-Plans: achieving the goal implies to execute (at least) one the plans $p_i$, according to a proper choice
- The choice implies a selection based on **contextual information** that include aspects like:
  - plan feasibility
  - importance/priority
  - environment state
  - robot state
  - etc.

# Reasoning and Deliberation

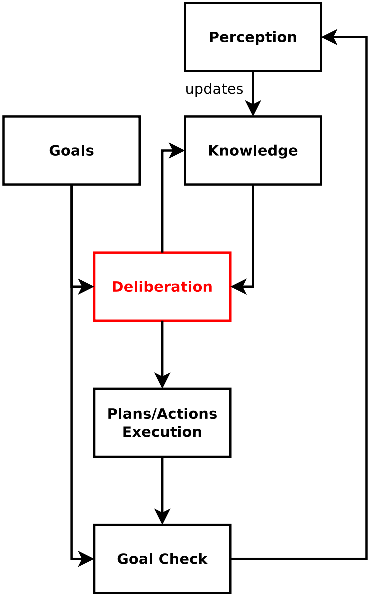
## Example: Goal "Gather Objects"

- AND-composition of plans:
  - `gather_object(PRISM)`
  - `gather_object(CYLINDER)`
  - `gather_object(CUBE)`

## Planning and Deliberation Criteria

- Which plan to choose?
- Example: "the one relative to the nearest object"
- We must have to know (time by time)
  - the robot pose
  - the position of each object (that could also vary during time)
- We must base our code on robot and environment state information obtained by sensors

# Reasoning and Deliberation Process Model



# Reasoning and Deliberation

## Plan Model

$$plan ::= \{feasibility(\cdot), opportunity(\cdot), task(\cdot), post\_condition(\cdot)\}$$

- *feasibility:* pre-condition to evaluate whether, if this instant, there are the conditions to execute the plan
- *opportunity:* numeric evaluation of the importance to execute this plan before another plan
- *task:* set of commands to execute the plan
- *post_condition:* condition (on the state of the robot/environment) that must be met in order to consider the plan successful

# Deliberative Systems

## The Paradigm “Belief-Desire-Intention” (BDI)

- Three base concepts: **Beliefs**, **Desires**, **Intentions**
- **Beliefs**, what the system believes, i.e. the information about the state of the robot and the environment, perceived by the sensors and/or elaborated according to an inference process
- **Desires**, what the system desire to do, i.e. the *goals* of the robot
- **Intentions**, what the system intend to do to achive the goals, i.e. the *plans* (computational part)

# Deliberative Systems

## BDI: Deliberative Process

1. Update of the **beliefs** on the basis of data coming from sensors
2. Analysis of the **goals** and detection (on the basis of the beliefs) of the ones that could be achieved
3. Extraction of the **intentions** (plans) from the goals and selection of plan to execute
4. Execution of the plan and update of the derived beliefs

# Intelligence of an autonomous robot
# Behaviour, Reasoning and Planning

Corrado Santoro

**ARSLAB - Autonomous and Robotic Systems Laboratory**
Dipartimento di Matematica e Informatica - Università di Catania, Italy
`santoro@dmi.unict.it`

Robotic Systems