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
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 achieving 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.
The environment 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?” or “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
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 perceived** by the sensors chosen for the robot.
Let us consider an environment populated by solids with different shape, color and position

Properties:

- **Shape**: prism, sphere, cylinder
- **Color**: yellow, red, green, black, white
- **Dimensions**: small, big
- **Position**: \((x, y, z)\)

Possible representation in C/C++

```c
typedef enum { PRISM, SPHERE, CYLINDER } t_shape;
ttypedef enum { YELLOW, RED, GREEN, BLACK, WHITE } t_color;
ttypedef enum { SMALL, LARGE } t_size;
ttypedef struct {
t _shape shape;
t _color color;
t _size size;
float x, y, z;
} t_block;
t_block my_blocks[...];
```
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++

```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 represents the link between two blocks

Possible representation in C/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;
}
```
The Block World

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

Possible representation in C/C++

```c
typedef enum { PRISM, SPHERE, CYLINDER } t_shape;
t typedef enum { YELLOW, RED, GREEN, BLACK, WHITE } t_color;
t typedef enum { SMALL, LARGE } t_size;
t 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[....];
```
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++

```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[....];
```
Since the physical environment 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.
The **object-oriented** model can be used to represent not only the things that are in environment but also the **relationships** among them.

The result is a **conceptual map**, named **ontology**, that represents the reference environment/context in which the robot operates.
Any representation of the environment must be “queryable” in order to allow the robot to reason in some way.

For example, if the robot would capture the green object placed 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.
A **Knowledge Base** is thus a data structure that memorises the knowledge the robot has on the reference environment.

- It is made by **concepts** that the robot **believes true** (beliefs).
- It has to support **queries**, **browsing** and **inferencing** new knowledge.

![Diagram showing relationships between different objects and their properties](image-url)
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
**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
In PROLOG, the knowledge base is made by **facts** represented by atomic formulae.

**Example**

- The cylinder, prism and sphere are “blocks”:
  
  ```prolog
  block(cylinder)
  block(prism)
  block(sphere)
  ```

- The cylinder is red, the prism is white, the sphere is black:
  
  ```prolog
  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:
  
  ```prolog
  upon(desk1, cylinder)
  upon(desk2, prism)
  upon(desk2, sphere)
  ```
To populate the knowledge base the PROLOG `assert` is used:

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

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

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

?- 
```
Queries on the Knowledge Base

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

?- block(cylinder).
  true.

?- block(cube).
  false.

?- block(sphere).
  true.
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 \texttt{block(X)} is equal to:

\[ \forall x : \text{block}(x) \]
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:

```prolog
?- 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”*
By means of first-order clauses, new predicates can be defined to derive new knowledge.

Example: let’s define the predicate “black object”:

\[
\text{black\_object}(X) :\text{-} \text{block}(X), \text{color}(X, \text{black}).
\]

?- black\_object(Obj).
Obj = sphere.

The definition is equivalent to the logic implication:

\[
\forall x : \text{block}(x) \land \text{color}(x, \text{black}) \Rightarrow \text{black\_object}(x)
\]
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”**
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:

\[ \forall x : block(x) \land color(x, black) \Rightarrow black_object(x) \]

Indeed, the PROLOG symbol “:-” is somewhat the symbol “\(\Rightarrow\)”. 
Derived Knowledge and Negation

Negations can also be defined

Example: let’s define the predicate “free desk”: it is \textbf{true} if no object is on that desk:

\begin{verbatim}
free_desk(Desk) :- \+upon(Desk, _).
\end{verbatim}

?- free_desk(desk1).
false.

?- free_desk(desk2).
false.

?- free_desk(desk3).
\textbf{true}.

Equivalent to:

\[
\text{free\_desk}(x) \iff \not\exists y : \text{upon}(y, x)
\]
Example: the Genealogic Tree

Corrado Santoro
Intelligence of an autonomous robot
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)).
### 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:

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

```prolog
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
```
Example: the Genealogic Tree

Derived Knowledge

\[ \text{sibling}(X, Y) \rightarrow X \text{ is the sibling of } Y \text{ (} X \text{ and } Y \text{ have the same parent):} \]

\[ \exists p, \forall x, y : \text{parent}(p, x) \land \text{parent}(p, y) \land x \neq y \Rightarrow \text{sibling}(x, y) \]

\[ \text{sibling}(X, Y) :- \text{parent}(P, X), \text{parent}(P, Y), X \neq Y. \]
Esempio: L’Albero Genealogico

Conoscenza Derivata

- \( \text{brother}(X, Y) \rightarrow X \) is the \( Y \)'s brother:

\[
\forall x, y : \text{sibling}(x, y) \land \text{male}(x) \Rightarrow \text{brother}(x, y)
\]

- \( \text{sister}(X, Y) \rightarrow X \) is the \( Y \)'s sister:

\[
\forall x, y : \text{sibling}(x, y) \land \text{female}(x) \Rightarrow \text{sister}(x, y)
\]

\[\text{brother}(X,Y) :- \text{sibling}(X,Y), \text{male}(X). \]
\[\text{sister}(X,Y) :- \text{sibling}(X,Y), \text{female}(X). \]
Behaviour, Goals, Planning
In an autonomous robot, there are two kinds of behaviours:

- reactive
- proactive
Reactivity and Proactivity

Reactivity

- Execution of a **prefixed** sequence of actions on the basis of the occurrence of a **sporadic** event
- It is equivalent to the **instinctual 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
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, environment 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
The implementation of reactive tasks is based, in general, on "callback" functions, invoked 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).
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 **perceive** the environment to determine its **state** and check if the goals have 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.
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 goal is reached when all the fields **captured** of each element of the array **my_blocks** are **true**

**Possible C/C++ representation**

```c
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 sphere, 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

<table>
<thead>
<tr>
<th>Planning is not always required</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goals:</strong> to capture all the blocks</td>
</tr>
<tr>
<td><strong>Sub-goals:</strong> to capture the sphere, to capture the cylinder, to capture the prism, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>To capture all the blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Planning</strong> the proper sequence of sub-goals, or</td>
</tr>
<tr>
<td><strong>Selecting</strong>, one by one, the most opportune sub-goal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>To Capture X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To execute</strong> the proper sequence of actions to capture object X</td>
</tr>
</tbody>
</table>
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 (failure) 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**
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;
}
In the behaviour of a robot, often a goal requires the choice among different plans \( p_1, \ldots, p_n \).

**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.
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
Plan Model

\[ \text{plan} ::= \{ \text{feasibility}(\cdot), \text{opportunity}(\cdot), \text{task}(\cdot), \text{post \_condition}(\cdot) \} \]

- **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
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 achieve the goals, i.e. the plans (computational part).
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