Intelligenza di un robot autonomo
Comportamento, ragionamento, pianificazione

Corrado Santoro

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

Programmazione Sistemi Robotici

Comportamento, ragionamento, pianificazione

- I modelli/metodi/algoritmi studiati finora ci permettono di:
  - Comandare il moto degli attuatori sulla base di un certo modello cinematico
  - Tenere presente i vincoli dell'ambiente e creare i percorsi opportuni
- Tuttavia i vari movimenti devono essere **coordinati** tra loro allo scopo di eseguire compiti più complessi
- L'insieme dei movimenti coordinati può poi far parte di una **strategia** che consente al robot di portare a termine un obiettivo ben preciso
- Tuttavia, dato un certo **obiettivo**, è possibile esistano **differenti strategie**: occorre dunque **pianificare** la strategia più adatta in quel momento specifico

Ambiente e Percezione

- L'ambiente è un sistema dinamico a tutti gli effetti
- Tuttavia, durante la programmazione del comportamento di un robot, non siamo interessati agli aspetti prettamente "controllistici" (variabili di stato, modello del sistema, etc.)
  ...
- ... ma sicuramente ad aspetti del tipo "com'è fatto l'ambiente" o "cosa accade in questo preciso momento"
- **Modellare l'ambiente** diventa pertanto uno degli aspetti fondamentale della programmazione del comportamento di un robot
- La modellazione dell'ambiente è inoltre strettamente legata alla tipologia di **sensori** impiegati per lo scopo

Percezione e Stato dell'Ambiente

Ambiente e Percezione

- Un ambiente fisico è pervaso da:
  - **oggetti inanimati**, incapaci di azioni autome,
  - **oggetti animati** (umani, altri robot), dotati di autonomia
- Essi sono caratterizzati da opportune **proprietà** quali *forma*, *colore*, *posizione*, etc.
- Alcune proprietà sono **immutabili** (*forma*, *colore*), altre possono variare nel tempo (*posizione*)

- **Modellare l'ambiente** implica definire delle **entità informatiche** (classi, oggetti, variabili, etc.) in grado di rappresentare gli oggetti dell'ambiente e le relative proprietà
- Tali informazioni **devono essere percepibili** (determinabili) dai sensori scelti per il robot

Esempio: Il Mondo dei Blocchi

Il Mondo dei Blocchi

- Un ambiente è pervaso da solidi di forma, colore e dimensioni diverse
- Proprietà:
  - **Forma:** *prisma, sfera, cilindro*
  - **Colore:** *giallo, rosso, verde, nero, bianco*
  - **Dimensioni:** *piccolo, grande*
  - **Posizione:** (x, y, z)

Possibile rappresentazione 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[....];
```

Esempio: Il Mondo dei Blocchi

Il Mondo dei Blocchi

- Se supponiamo che i blocchi possono trovarsi uno sopra l'altro è necessario modellare anche questa caratteristica:
  - attraverso una **funzione booleana** che si basa sulle coordinate dei blocchi, ...

Possibile rappresentazione 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)
{
   //....
}
```

Esempio: Il Mondo dei Blocchi

Il Mondo dei Blocchi

- Se supponiamo che i blocchi possono trovarsi uno sopra l'altro è necessario modellare anche questa caratteristica:
  - ... oppure aggiungendo **un'ulteriore proprietà** che rappresenta il link tra due blocchi

Possibile rappresentazione 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;
}
```

Esempio: Il Mondo dei Blocchi

Il Mondo dei Blocchi

- Se i blocchi vanno **catturati** è necessario modellare anche questo tipo di operazione:
  - utilizzando **due array**, uno per i blocchi non catturati ed un per quelli catturati ....

Possibile rappresentazione 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[....];
```

Esempio: Il Mondo dei Blocchi

Il Mondo dei Blocchi

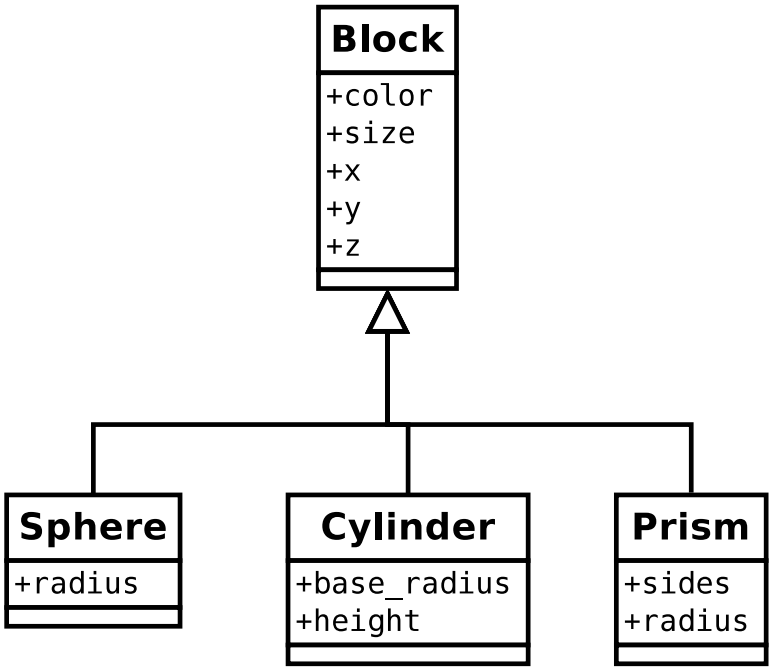
- Se i blocchi vanno **catturati** è necessario modellare anche questo tipo di operazione:
  - ... oppure aggiungendo un **booleano** che indica se il blocco è stato catturato

Possibile rappresentazione 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[....];
```

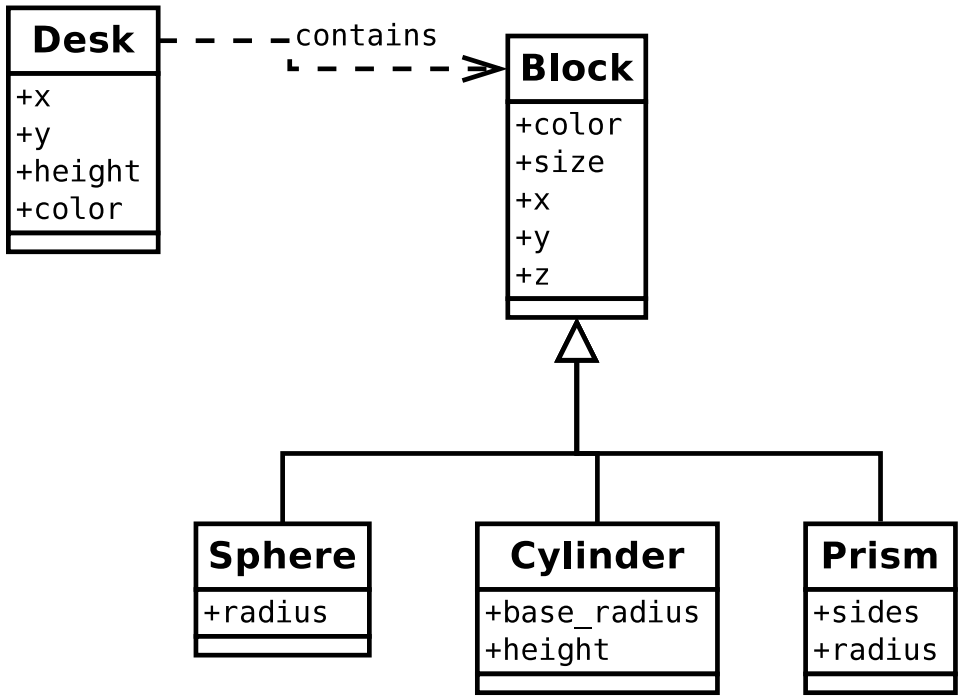
Rappresentazione “Object-Oriented”

- Poichè l’ambiente fisico è pervaso da **oggetti**, una rappresentazione spesso usata è basata sul modello **object-oriented**
- In realtà l’*object-orientation* è nata nel 1965, all’interno dell’AI, proprio per poter rappresentare i “mondi” dei sistemi artificiali



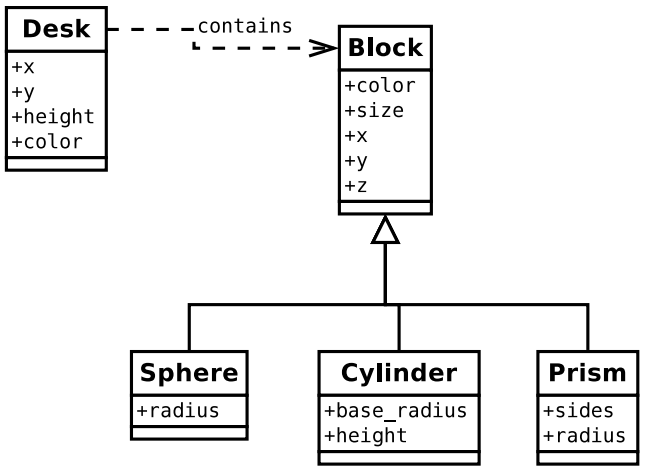
Le Ontologie

- Il modello **object-oriented** permette non solo di rappresentare ciò di cui è composto l'ambiente ma anche le **relazioni** tra le entità
- Il risultato è una **mappa concettuale**, denominata **ontologia**, che rappresenta (nel modo più fedele possibile) l'ambiente di riferimento/il contesto in cui si opera



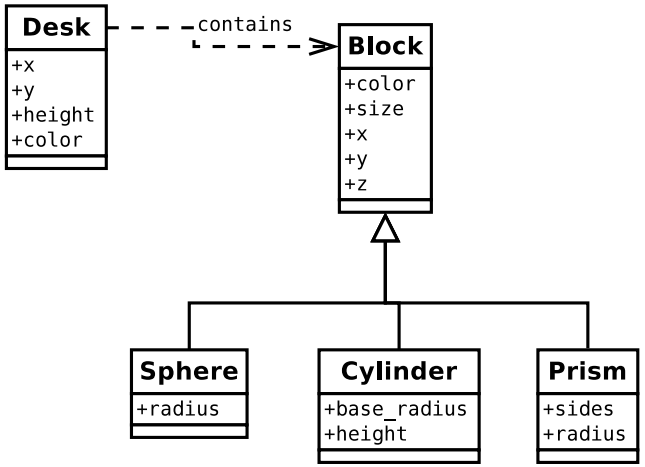
Le “Basi di Conoscenza” (Knowledge Base)

- Qualunque tipo di rappresentazione deve essere “interrogabile” al fine di consentire una forma più o meno complessa di ragionamento
- Ad esempio, se il robot volesse **catturare l’oggetto verde che si trova sul tavolo bianco**, esso dovrà essere in grado di risalire all’oggetto (istanza di `Block`) relativo per poterlo poi raggiungere
- Data un’ontologia (o una rappresentazione analoga), le informazioni devono essere organizzate in un’apposita **base di conoscenza** (**knowledge base**) che possa essere interrogabile facilmente



Le “Basi di Conoscenza” (Knowledge Base)

- Una **Knowledge Base** è pertanto una struttura dati che permette di memorizzare la **conoscenza che il robot possiede sull’ambiente** di riferimento
- E’ costituita da **concetti** che il robot **ritiene veri** (credenze, beliefs)
- Deve consentire la **interrogazione**, **navigazione** e l’**inferenza** di nuova conoscenza



Rappresentazione tramite logica del primo ordine

- Il modello **logico** costituisce un'alternativa per la rappresentazione della conoscenza
- E' basato sulla definizione di **fatti** (credenze, beliefs) rappresentati da **predicati in logica del primo ordine**
- L'inferenza/interrogazione in questi sistemi si basa su **formule logiche**
- Esistono linguaggi di programmazione, come il **PROLOG** supportano direttamente questo tipo di modello

Elementi di PROLOG

Elementi di PROLOG

- Il **PROLOG** (PROgramming in LOGic) è un linguaggio di programmazione simbolico basato su predicati logici
- Oltre ai classici tipi numerici, il PROLOG introduce il concetto di **atomo**: esso è un **letterale** che inizia per lettera **minuscola**, ed è un **simbolo indivisibile** (non è una "stringa"!)
- Un letterale che inizia per lettera **Maiuscola** è invece una **variabile**:
  - `cube` è un atomo
  - `Cube` è una variabile

Rappresentazione tramite “fatti” (Beliefs)

Fatti o Beliefs

In PROLOG, La base di conoscenza è formata da **fatti certi**, i quali sono rappresentati tramite formule atomiche

Esempi

- Cilindro, prisma e sfera sono dei blocchi:

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

- Il cilindro è rosso, il prisma è bianco, la sfera è nera:

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

- Il cilindro è sul tavolo 1, il prisma e la sfera sono sul tavolo 2:

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

Rappresentazione tramite “fatti” (Beliefs)

Base di Conoscenza

Per popolare la base di conoscenza, è sufficiente utilizzare lo statement PROLOG **assert**

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

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

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

?-
```

Interrogazioni sulla Base di Conoscenza

La base di conoscenza può essere interrogata per sapere se un determinato fatto è vero o falso.

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

?- block(cube).
false.

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

Interrogazioni sulla Base di Conoscenza

La base di conoscenza può essere interrogata anche usando variabili **quantificate universalmente**

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

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

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

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

Interrogazioni sulla Base di Conoscenza

La base di conoscenza può essere interrogata usando variabili **quantificate universalmente**

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

L'interrogazione `block(X)` equivale a:

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

Interrogazioni sulla Base di Conoscenza

E' possibile combinare le interrogazioni utilizzando la "," che ha ruolo di connettivo **AND**:

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

?-
```

Tutti gli oggetti, con il relativo colore

Interrogazioni sulla Base di Conoscenza

E' possibile combinare le interrogazioni utilizzando la "," che ha ruolo di connettivo **AND**:

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

?-
```

Tutti gli oggetti, con il relativo colore, che si trovano sul "desk2"

Conoscenza Derivata

Tramite clausole del primo ordine è possibile definire nuovi predicati e derivare nuova conoscenza.

Esempio: definiamo il predicato "oggetto nero":

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

?- black_object(Obj).
Obj = sphere.

?-
```

La definizione equivale all'implicazione logica:

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

Conoscenza Derivata

Esempio: definiamo il predicato "oggetto nero":

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

?- black_object(Obj).
Obj = sphere.

?-
```

Oppure:

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

In realtà il simbolo PROLOG "`:-`" è una stilizzazione del simbolo "$\Leftarrow$"

Conoscenza Derivata e Negazione

E' anche possibile inserire negazioni.

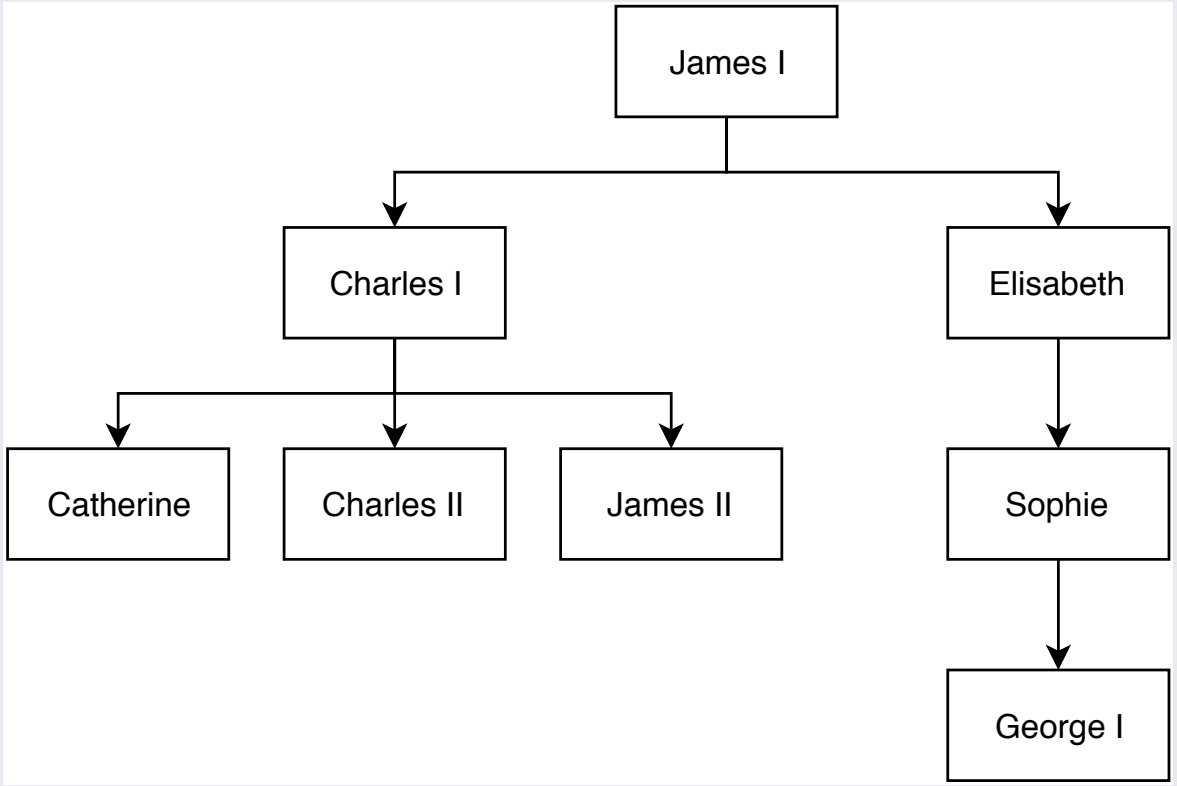
Esempio: definiamo il predicato "tavolo libero" il quale è **true** se non c'è alcun oggetto su di esso:

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

?- free_desk(desk1).
false.

?- free_desk(desk2).
false.

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

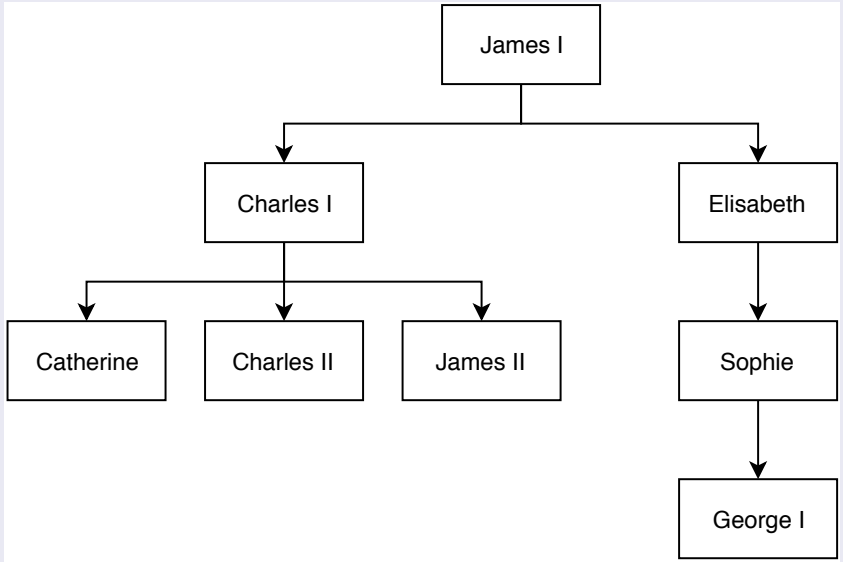
Equivalente a:

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

Esempio: L'Albero Genealogico



Esempio: L'Albero Genealogico



Fatti

- `male(X)` → X è un uomo
- `female(X)` → X è una donna
- `parent(X, Y)` → X è genitore di Y

Esempio: L'Albero Genealogico

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)).
```

Esempio: L'Albero Genealogico

Conoscenza Derivata

- **`father(X, Y)`** $\rightarrow$ X è il padre di Y:

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

- **`mother(X, Y)`** $\rightarrow$ X è la madre di Y:

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

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

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

Esempio: L'Albero Genealogico

Conoscenza Derivata

- **`sibling(X, Y)`** $\rightarrow$ X è fratello o sorella di Y (X e Y hanno lo stesso genitore):

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

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

Esempio: L'Albero Genealogico

Conoscenza Derivata

- **`brother(X, Y)`** $\rightarrow$ X è fratello di Y:

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

- **`sister(X, Y)`** $\rightarrow$ X è sorella di Y:

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

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

Comportamento, Obiettivi, Pianificazione

Reattività e Proattività

- In un sistema robotico autonomo, si distinguono due tipologie di comportamento:
  - **reattivi**
  - **proattivi**

Reattività e Proattività

Reattività

- Esecuzione di una sequenza di azioni **prefissata** sulla base di un evento **sporadico**
- Equivalente alla **reazione istintiva** degli umani
- Esempi
  - Arrestare il robot quando un sensore di distanza rileva un ostacolo
  - Attivare un braccio quando si passa accanto ad un oggetto

Reattività e Proattività

Proattività

- **Pianificare** l'adeguata sequenza di azioni che possa portare a raggiungere un determinato obiettivo
- Equivalente al **ragionamento** degli umani
- Esempi
  - Identificare e recuperare un oggetto
  - Adottare opportune manovre evasive per evitare una collisione

Task Reattivi

Reattività e Paradigma “ECA”

Event-Condition-Action

- La reattività viene di solito programmata adottando un paradigma **Event-Condition-Action (ECA)**
  - **E**, evento scatenante
  - **C**, condizione (predicato) a cui devono soddisfare i parametri dell’evento, o lo stato dell’ambiente, o lo stato del sistema in generale
  - **A**, azione (computazione) che deve essere eseguita qualora la condizione sia vera

Esempio: Obstacle Detection

- **Event**, dato campionato dal sensore di distanza
- **Condition**, controllo se la distanza letta è inferiore ad una certa soglia
- **Action**, stop del robot

ECA: Implementazione

- L’implementazione di task reattivi si basa, in genere, su funzioni di “callback” invocate sulla base di un **evento** specifico
- L’evento può essere il campionamento di un sensore oppure la produzione di un dato da parte di un altro task del sistema di controllo
- La condizione (in genere) è un predicato (**if**) applicato alla rappresentazione del sistema/ambiente (base di conoscenza)

ECA: Requisiti Temporali

- In taluni casi, i task reattivi possono essere caratterizzati da requisiti temporali ben precisi
- L’obstacle detection è uno di questi casi e richiede che l’evento venga processato **tempestivamente**, o meglio **entro una certa “deadline” temporale**
- I requisiti “real-time” che abbiamo visto per i task periodici di controllo si applicano dunque (in taluni casi) anche ai task reattivi, i quali (nel gergo dei sistemi real-time) vengono denominati **task sporadici**

Proattività, Obiettivi e Ragionamento

Ragionamento e Pianificazione

1. Dato un certo **obiettivo**
2. **Percepisco** l'ambiente per determinarne lo **stato** e verificare che l'obiettivo non sia stato effettivamente raggiunto
3. Determino la **strategia** (insieme di azioni) che potrebbe portare al raggiungimento dell'obiettivo
4. **Eseguo** le relative azioni
5. Ritorno allo step 1

Percezione, Rappresentazione e Obiettivi

- Un **obiettivo** (**goal**) rappresenta uno **stato dell'ambiente** che si desidera raggiungere
- Un obiettivo può pertanto essere espresso tramite un predicato (o funzione booleana) sulle variabili che rappresentano l'ambiente

Il Mondo dei Blocchi

- **Obiettivo/Goal: catturare tutti i blocchi**
- L'obiettivo è raggiunto quando i campi **`captured`** di tutti gli elementi dell'array **`my_blocks`** sono `true`

Possibile rappresentazione in C/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;
}
```

Obiettivi e sotto-obiettivi

- Raggiungere un obiettivo implica eseguire un certo insieme di azioni
- E tuttavia a volte un obiettivo cela, al suo interno, dei sotto-obiettivi
- Esempio: **catturare tutti i blocchi implica catturarli uno alla volta**
- Obiettivo: **catturare tutti i blocchi**
- Sotto-obiettivi: *catturare la sfera*, *catturare il cilindro*, *catturare il prisma*, etc.
- I sotto-obiettivi non devono necessariamente essere eseguiti secondo una sequenza prefissata
- Tuttavia alcuni sotto-obiettivi potrebbero non essere fattibili (esempio: il cilindro non può essere catturato perchè c'è il prisma sopra)
- Occorre pertanto un processo di pianificazione/scelta dei sotto-obiettivi

Goal con pianificazione e Goal “semplici”

Non sempre la pianificazione è richiesta

- Obiettivo: **catturare tutti i blocchi**
- Sotto-obiettivi: *catturare la sfera*, *catturare il cilindro*, *catturare il prisma*, etc.

Catturare tutti i blocchi

- **Pianificare** l’opportuna sequenza di sotto-obiettivi, oppure
- **Scegliere**, di volta in volta, il sotto-obiettivo più opportuno

Catturare X

- **Eseguire** l’opportuna sequenza di azioni per catturare l’oggetto X

Goal che non richiedono pianificazione

Plans

- Questi goal sono costituiti da una sequenza di azioni, ognuna corrispondente all'attivazione di una movimentazione (bracci, ruote, etc.)
- L'esecuzione della sequenza molto spesso implica l'attesa che ogni azione si completi (successo), o che vengano identificate delle condizioni per cui il completamento è impossibile (fallimento)
- I goal che ricadono in questa tipologia vengono spesso chiamati **plans** (piani)
- Dal punto di vista implementativo, i plans posso pensarsi basati su una sequenza:
  - chiamata di funzione 1
  - if/switch-case sull'esito della chiamata 1
  - chiamata di funzione 2
  - if/switch-case sull'esito della chiamata 2
  - ...
- Il modello è dunque assimilabile a un **automa a stati finiti** (finite-state machine, FSM)

Esempio: il mondo dei blocchi

```
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;
}
```

Goal e Pianificazione

- Nel comportamento di un robot, spesso un **goal** richiede un scelta tra diversi **plans** $p_1, ..., p_n$
- Piani in **and**: raggiungere il goal significa eseguire tutti i plans p_i, ma il loro ordine è basato su una scelta a run-time
- Plans in **xor**: raggiungere il goal significa eseguire (almeno) uno dei plans p_i, secondo una scelta opulata
- La scelta implica una selezione basata su **informazioni contestuali** che includono aspetti come:
  - fattibilità del plan
  - importanza/priorità
  - stato dell'ambiente
  - stato del robot
  - etc.

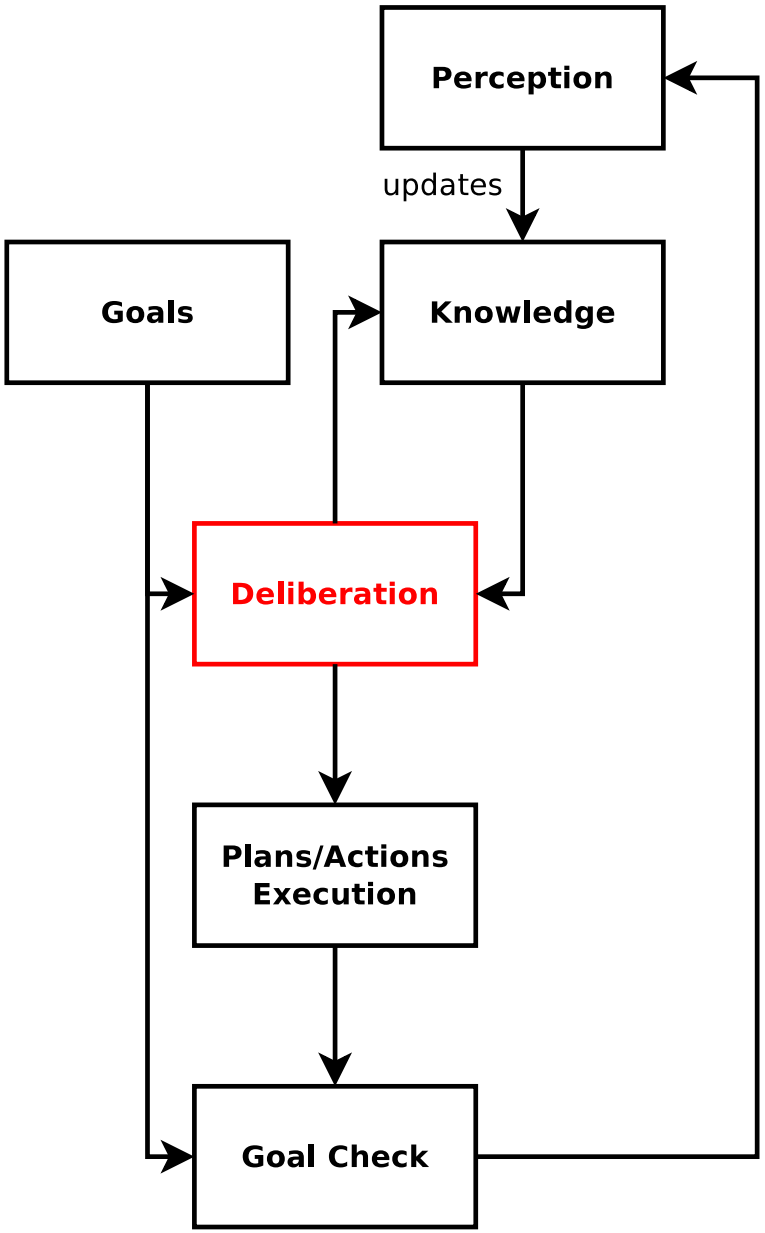
Esempio: Goal “Gather Objects”

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

Criteri di Pianificazione e Deliberazione

- Quale scegliere?
- Esempio: “quello relativo all’oggetto più vicino”
- E’ necessario dunque conoscere (istante per istante)
  - la posa del robot
  - la posizione di ogni oggetto (che potrebbe anche variare nel tempo)
- Occorre cioè basarsi su informazioni di stato del robot e dell’ambiente ottenute dai sensori

Modello del processo di ragionamento e deliberazione



Modello di Plan

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

- *feasibility:* pre-condizione per valutare se, in questo determinato istante, esistono le condizioni per poter eseguire plan
- *opportunity:* valutazione numerica dell'importanza di eseguire questo plan prima di un altro plan
- *task:* insieme di istruzioni per l'esecuzione del plan
- *post_condition:* condizione (sullo stato del sistema/ambiente) che deve essere verificata al fine di considerare il plan eseguito con successo

Il Paradigma “Belief-Desire-Intention” (BDI)

- Tre concetti base: **Beliefs**, **Desires**, **Intentions**
- **Beliefs**, ciò che il sistema crede, cioè le informazioni di stato sul robot e sull’ambiente, percepite dai sensori e/o elaborate a seguito di un processo di inferenza
- **Desires**, ciò che il sistema desidera fare, ossia gli *obiettivi/goal* del robot
- **Intentions**, ciò che il sistema ritiene di poter fare per raggiungere gli obiettivi, ossia i *plans* (parte computazionale)

BDI: Processo Deliberativo

1. Aggiornamento dei **beliefs** sulla base dei dati dai sensori
2. Analisi dei **goals** e individuazione (sulla base dei beliefs) di quelli che possono essere raggiunti
3. Estrazione delle **intentions** (plans) dai goal e selezione del piano da eseguire
4. Esecuzione del piano e eventuale aggiornamento dei belief derivati

Intelligenza di un robot autonomo
Comportamento, ragionamento, pianificazione

Corrado Santoro

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

Programmazione Sistemi Robotici