

*Mini caso di studio relativo alla progettazione e alla realizzazione di un piccolo robot in grado di compiere missioni di navigazione autonoma, evidenziando come la progettazione ad oggetti sia applicabile alla robotica e l'intelligenza artificiale*

## **Navigazione autonoma di robot: definizioni e concetti base**

### **Introduzione**

In questo articolo esamineremo alcuni aspetti legati all'intelligenza artificiale e alla robotica che costituiscono il necessario fondamento sul quale costruire sia il design, sia l'implementazione del progetto. In particolare affronteremo due principali tematiche: quali sono le caratteristiche che rendono un robot "intelligente" e quali sono i principali paradigmi di progettazione dai quali possiamo trarre ispirazione per elaborare un'architettura software per il nostro robot.

### **Cosa significa per un robot essere "intelligente"?**

Il termine robot spesso è associato ad una figura antropomorfa nel pensiero comune, probabilmente grazie ai molti film e libri di fantascienza prodotti negli ultimi decenni. Tuttavia l'apparenza, ossia il "vestito" esterno con cui modelliamo il robot, non costituisce una definizione utile, né la sola somiglianza all'essere umano lo rende di per sé "intelligente". Basti pensare ai robot impiegati nell'automazione industriale che non presentano necessariamente delle somiglianze con le caratteristiche fisiche di un essere umano, eppure sono a tutti gli effetti dei robot. L'associazione di un robot all'immagine antropomorfa può essere fuorviante non solo da un punto di vista estetico, ma anche da quello cognitivo, poiché costituisce un paragone molto "impegnativo" nel tentativo di trasportare il concetto di intelligenza dall'individuo umano alla macchina. Svincolandoci da questo paragone, definiamo un robot come una creatura meccanica in grado di effettuare delle operazioni svolte tipicamente da un essere umano (pensiamo in particolare all'insieme di attività schematiche e ripetitive tipiche del lavoro in fabbrica, come la costruzione, la rifinitura e la verniciatura di pezzi meccanici). In base a questa definizione, non tutti i robot sono quindi "intelligenti". Definiamo infatti un *"robot intelligente" come una creatura meccanica in grado di svolgere attività prettamente umane e di funzionare autonomamente* (1), indipendentemente dall'intervento dell'essere umano. L'aggettivo "intelligente" implica che il robot non esegue delle azioni secondo lo schema ripetitivo e stupido che caratterizza i robot spesso impiegati nell'automazione industriale. Tale definizione enfatizza un'altra importante differenza: non solo un robot non deve essere fisicamente simile ad un essere umano per svolgere alcune delle sue mansioni, ma esso non è neanche equivalente ad un computer. Un robot può utilizzare un computer come parte costruttiva (una sorta di componente strutturale), esattamente come un essere umano utilizza il cervello oppure il sistema nervoso. A differenza dei computer, tuttavia, il robot è in grado di interagire con il mondo esterno. Per interazione intendiamo qualcosa di più del semplice disporre di una tastiera sulla quale un operatore può digitare dei comandi o acquisire dalla porta seriale dei segnali elettrici. Il concetto di interazione acquisisce nella robotica un'accezione particolare, se pensiamo al funzionamento autonomo che abbiamo reso parte della definizione di robot intelligente. *Interagire* col mondo esterno significa potersi *muovere autonomamente*, *percepire* l'ambiente riconoscendo ad esempio degli "ostacoli" (sensing), *reagire* ad essi, ad esempio evitandoli, e modificare l'ambiente stesso (act/react), ad esempio afferrando un pezzo meccanico e spostandolo.

### **Robotica e paradigmi di progettazione**

Un paradigma è una filosofia, uno stile di pensiero, un set di assunzioni e tecniche che caratterizzano l'intero approccio ad una classe di problemi (1). Robin Murphy descrive un paradigma sia come un "modo per osservare un problema", sia come un insieme di tool per risolvere lo stesso problema. In tal senso, nessun paradigma è corretto; piuttosto esistono problemi che sembrano più facilmente trattabili seguendo un particolare approccio. In altre parole, applicare il paradigma più adatto semplifica la soluzione del problema<sup>1</sup>. Prima di progettare una soluzione per la navigazione autonoma del nostro robot,

---

<sup>1</sup> Questo vale tanto per la robotica quanto per l'informatica in generale. Il paradigma object-oriented, ad esempio, non è il paradigma "corretto", mettendo in luce quello procedurale come il paradigma "sbagliato". Semplicemente esso risulta il paradigma più adatto a gestire la complessità di molti dei problemi che i moderni sistemi software devono risolvere.

quindi, diventa importante capire quali sono i paradigmi più importanti nella robotica, con particolare riferimento "all'organizzazione dell'intelligenza", ossia alla definizione di quella componente dell'intero "sistema robot" preposta a:

- percepire gli stimoli dall'esterno (SENSE);
- ragionare con diversi livelli di pianificazione del proprio comportamento al fine di prendere una decisione (PLAN);
- agire (ACT).

Identifichiamo questi tre compiti mediante altrettante "primitive"<sup>2</sup>: SENSE, PLAN e ACT<sup>3</sup>. Queste tre primitive svolgono un ruolo essenziale nella definizione dei diversi approcci alla risoluzione di un problema di robotica. In particolare, possiamo descrivere i paradigmi di progettazione in ambito robotico mediante due approcci ortogonali e complementari:

- *Approccio basato sulle primitive*: descrive le relazioni tra le tre primitive SENSE, PLAN e ACT;
- *Approccio basato sul controllo e la distribuzione delle informazioni*: descrive il modo in cui i dati acquisiti dai sensori vengono processati e distribuiti attraverso i vari componenti del sistema.

Per mettere in relazione le tre primitive dobbiamo prima capire quali funzionalità ciascuna di esse rappresenta. Nel loro insieme, tali primitive rappresentano una ragionevole partizione delle funzionalità di un robot<sup>4</sup>. Ricadono nella categoria SENSE tutte quelle funzioni che acquisiscono informazione dai sensori e producono un output, utilizzato solitamente da altre funzioni in una delle restanti due categorie. Appartengono invece alla categoria PLAN le funzionalità che acquisiscono informazione (sia dai sensori, sia dalla memoria, per mezzo della costruzione di modelli del mondo circostante) e producono uno o più task da eseguire. Un set di task che costituisce (parte della) la *pianificazione* di una missione può essere il seguente:

"muoversi dalla posizione attuale in linea retta per 3 metri, girare a sinistra, procedere per 3 metri, ruotare di 90°, scattare una foto, ritornare alla posizione iniziale, spegnere i motori"

Tale pianificazione può essere l'obiettivo dell'intera missione, oppure solo un insieme di goal necessari (ma non sufficienti, evidentemente) per completare la missione stessa. Chiamiamo con il termine di *piano di missione* l'insieme di task necessari e sufficienti per completare con successo la missione. E' importante notare che le funzionalità nella categoria PLAN non eseguono tali task: esse si limitano a pianificarli! In altre parole le funzionalità di tipo PLAN modificano costantemente il piano di missione in base alla percezione del mondo esterno e del raggiungimento degli obiettivi della missione.

Primitive del robot	Input	Output
SENSE	Dati provenienti dai sensori	Informazione processata (prodotta come output) dai sensori ("sensed information" o informazione di tipo sensoriale)
PLAN	Informazione processata dai sensori oppure informazione di tipo cognitivo contenuta in memoria (world models)	Direttive, insieme di task, piano di missione
ACT	Informazione processata dai sensori oppure direttive provenienti dalla pianificazione	Comandi agli attuatori

**Tabella 1 – Le primitive del robot in termini di input e di output**

La categoria ACT, infine, è costituita dalle funzionalità che producono come output i comandi che pilotano direttamente i motori (gli attuatori più in generale). Un attuatore è un qualsiasi dispositivo, non necessariamente elettrico, che consente di intervenire indirettamente sul funzionamento o sul controllo di

<sup>2</sup> Il termine primitiva in questo contesto non rappresenta una singola funzionalità, bensì una classe distinta di funzionalità.

<sup>3</sup> Le tre primitive SENSE, PLAN e ACT sono tradizionalmente accettate nella comunità scientifica che si occupa di robotica. Esiste tuttavia una suggestiva quarta primitiva, LEARN, maggiormente focalizzata sulle funzionalità di apprendimento (più o meno autonomo) di un robot. Seppure esistono diverse applicazioni che sfruttano questa funzionalità (mediante l'uso di reti neurali, di sistemi fuzzy, ecc.), non esiste al momento un'architettura formale che la includa, cosicché un vero paradigma deve essere ancora scoperto.

<sup>4</sup> Con l'eccezione della funzionalità di apprendimento descritta dalla primitiva LEARN.

meccanismi; per esempio, il motore che gira il timone di una nave funge da attuatore. Un esempio di funzionalità che fa parte della categoria ACT è la seguente:  
 “svolta di 98° in senso orario mantenendo una velocità di 0,2 mps”.

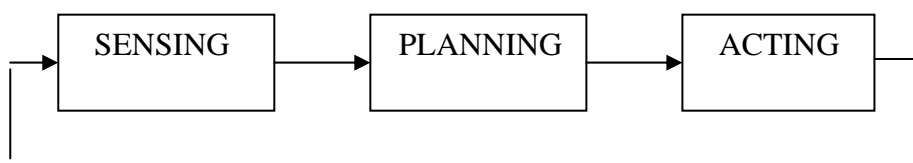
La **tabella 1** riassume le tre primitive in termini di input e di output.

Descrivere adeguatamente un paradigma limitandosi semplicemente a categorizzare una funzionalità all’interno di un insieme di primitive come SENSE-PLAN-ACT non è, tuttavia, sufficiente per un progettista, poiché non riusciamo ancora a descrivere in modo organico e dettagliato il modo in cui le informazioni vengono processate e come i compiti vengono distribuiti all’interno del sistema. Avere una visione chiara di come distribuire i compiti – e quindi la logica di controllo, l’intelligenza – attraverso i sottosistemi è un obiettivo fondamentale per il progettista. Questa considerazione ci porta a descrivere un paradigma *anche* mediante il secondo approccio precedentemente evidenziato, basato sul controllo e sulla distribuzione del trattamento delle informazioni. Da questo punto di vista, possiamo caratterizzare i paradigmi in base all’organizzazione sensoriale in essi attuata. Per *organizzazione sensoriale* intendiamo proprio l’organizzazione delle componenti e l’attribuzione delle responsabilità di trattamento delle informazioni provenienti dai sensori. In alcuni paradigmi le informazioni di tipo sensoriale sono utilizzate in modi dedicati, specifici, ristretti, da ciascuna funzionalità del robot. Spesso tali informazioni sono il risultato di elaborazioni successive, richieste solo in determinate situazioni. In questi casi il trattamento dell’informazione è detto di tipo *locale*. Il modello del mondo è costruito ad hoc per soddisfare le necessità correnti. In altri paradigmi, invece, tutte le informazioni sensoriali sono processate all’inizio per costruire un *modello globale* del mondo (global world model). Un sottoinsieme di tale modello viene di volta in volta fornito alle singole funzioni che ne hanno bisogno durante le attività del robot. Questi paradigmi sono quindi basati su un trattamento dell’informazione di tipo globale. Nei paragrafi successivi verranno descritti i paradigmi di tipo gerarchico, reattivo e ibrido “deliberativo-reattivo”. Ogni paradigma influenza fortemente le architetture software per l’implementazione della logica di controllo e “dell’intelligenza” di un robot. In base al tipo di robot che costruiremo e alle missioni di navigazione che esso dovrà svolgere, identificheremo in primo luogo un fondamento teorico per la scelta del paradigma più adatto ed, in base a tale scelta, progetteremo un’architettura adeguata allo scopo delle missioni del robot.

### Introduzione ai paradigmi gerarchico, reattivo e ibrido

Prima di descrivere nel dettaglio ciascun paradigma, può essere utile presentare una panoramica delle loro caratteristiche salienti e fornire alcuni criteri per scegliere un’architettura software, con particolare riferimento alle architetture utilizzate in robotica.

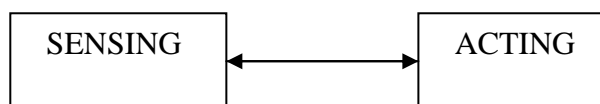
Il **paradigma gerarchico** è storicamente uno dei primi paradigmi concepiti per risolvere in modo top-down i compiti svolti dai robot negli anni ‘60-’70 dello scorso secolo [Murphy, 2000]. Esso si basa su un principio che possiamo chiamare “di osservazione introspettiva” che spesso condiziona anche il ragionamento – e il comportamento – umano. Un esempio chiarirà le cose. Immaginiamo di trovarci all’interno di una stanza con una finestra aperta. Ad un certo punto qualcosa all’esterno attira la nostra attenzione, al punto da indurci ad andare alla finestra per osservare meglio la situazione. Nella circostanza appena descritta il nostro apparato sensoriale (visivo e forse anche uditivo) “percepisce qualcosa” dall’esterno (SENSING), il nostro cervello elabora l’informazione, associando forse il suono o il movimento ad un oggetto. Interessati dal nuovo evento, decidiamo quindi di spostarci alla finestra, pianificando più o meno inconsciamente il percorso più breve per raggiungerla (PLANNING), facendo attenzione ad evitare eventuali ostacoli. Nel momento in cui ci ritroviamo davanti alla finestra, abbiamo tradotto in azione l’intento pianificato (ACTING). Nel paradigma gerarchico, il robot è governato dal medesimo principio: esso percepisce il mondo esterno, ossia l’ambiente su cui opera, pianifica la prossima azione da compiere ed, infine, esegue tale azione. Queste attività vengono svolte dal robot in modo strettamente sequenziale. Chiameremo tale sequenza di fasi *ciclo SENSING-PLANNING-ACTING*, illustrato in **figura 1**.



**Figura 1 – Fasi tipiche del paradigma gerarchico**

Al termine della fase di ACTING, il ciclo si ripete finché (ad esempio) la missione non sarà portata a termine. È facile intuire quindi che, in ogni architettura che implementa il paradigma gerarchico, esistono almeno tre componenti distinti: il modulo di acquisizione dei dati dai sensori, quello di pianificazione e quello di elaborazione dei comandi ai motori - attuatori. Un'altra caratteristica che contraddistingue il paradigma gerarchico è relativa al trattamento dei dati percepiti dai sensori, i quali vengono elaborati al fine di formare un *modello globale del mondo*, ossia una singola rappresentazione che viene poi utilizzata dal modulo di pianificazione per decidere quale sarà la prossima azione da compiere.

L'approccio suggerito dal paradigma gerarchico è stato negli anni successivi fortemente criticato, per diverse ragioni. Da un punto di vista psicologico, oggi sappiamo che l'introspezione non è sempre il migliore strumento per caratterizzare (o anche solo spiegare) in modo accurato un processo cognitivo. I processi cognitivi sono così complessi che la loro esecuzione non è sempre strettamente sequenziale. Da un punto di vista informatico ed ingegneristico, inoltre, la costruzione di modelli globali è costosa, difficile da assemblare e piuttosto fragile, necessitando di continue assunzioni sull'ambiente esterno e sulla sua rappresentazione, soggetta alle limitazioni di memoria tipiche degli apparecchi embedded come i robot. Da un punto di vista biologico, infine, basare il ragionamento e il comportamento di un robot esclusivamente sul ciclo SENSING-PLANNING-ACTING significa ignorare quella che viene chiamata "evidenza biologica". Se vogliamo prendere una tazza di caffè e, toccandola, ci scottiamo, è probabile che lasciamo istintivamente la presa, rischiando magari di far cadere la tazza a terra, perché è più importante preservare la nostra incolumità che bere il caffè. Questo meccanismo di difesa è tendenzialmente istintivo e viene applicato senza una pianificazione preordinata e speculativa, a differenza del gesto di bere il caffè che può essere speculativo in quanto determinato a soddisfare un piacere momentaneo (il gusto o la sete). Trasportando le stesse proprietà dal mondo umano al mondo dell'intelligenza artificiale, un robot che viene mandato all'interno di un edificio in fiamme dovrà localizzare la zona incendiata per poi cercare di estinguere l'incendio. Se però esso si avvicinasse troppo alle fiamme, rischierebbe di bruciare a sua volta, rendendo del tutto inutile la missione. Il robot, analogamente all'essere umano che si scotta con la tazza, deve cercare di evitare danneggiamenti strutturali, almeno finché gli è possibile completare in sicurezza la missione, preservando le proprie funzionalità. Questi due esempi ci fanno capire come l'emergere di situazioni particolari facciano scattare meccanismi biologici "di basso livello", soggetti più all'istinto di sopravvivenza che alla speculazione di un ragionamento opportunistico. Se alla base di un ragionamento ci può essere una fase di pianificazione frapposta tra percezione del mondo esterno e azione sul mondo stesso, appare invece corretto modellare l'emergenza biologica passando direttamente dalla fase di SENSING alla fase di ACTING, come illustrato in **figura 2**.



**Figura 2 – Nel paradigma reattivo la fase di pianificazione è del tutto assente**

Da tali osservazioni deriva il **paradigma reattivo**. In esso la fase di pianificazione viene completamente eliminata, da cui deriva un netto guadagno in efficienza rispetto al paradigma gerarchico. Come vedremo successivamente, l'esecuzione di più fasi SENSING-ACTING non è di tipo sequenziale, come avviene nel paradigma gerarchico, bensì di tipo concorrente. Ogni accoppiamento di primitive SENSE-ACT viene chiamato "behavior", ossia comportamento<sup>5</sup>. Ogni comportamento sfrutta una *rappresentazione locale del mondo* ed elabora la migliore azione da compiere indipendentemente da altri comportamenti preesistenti e attivi. Riprendendo il nostro esempio della tazza, possiamo immaginare due comportamenti latenti: "bere" ed "evitare ustioni di contatto". Tali comportamenti vengono elaborati dal nostro apparato cognitivo-sensoriale in modo indipendente uno dall'altro. Possiamo immaginare che ad ogni comportamento sia assegnata una priorità e una condizione di attivazione:

comportamento: "bere"

condizioni di attivazione:

avere sete; (dipende dallo stato interno dell'essere umano)

c'è una bevanda a disposizione; (dipende dall'ambiente esterno)

priorità (0-10): 7

---

<sup>5</sup> In questo contesto un comportamento è una mappa che mette in relazione un ben determinato input proveniente dai sensori con un insieme di azioni necessarie per eseguire un particolare task.

*comportamento*: "evitare ustioni di contatto"

condizioni di attivazione<sup>6</sup>:

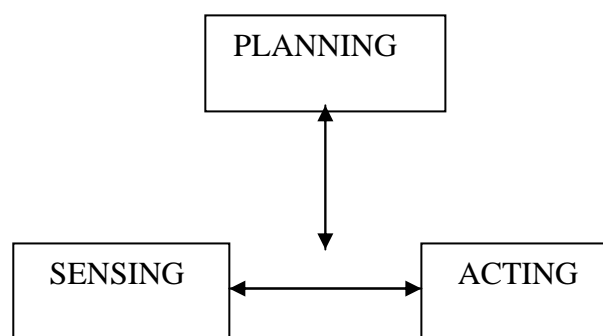
temperatura oggetto > 50°C; (dipende dallo stato dell'ambiente esterno)

contatto presente tra l'oggetto e una parte del corpo umano; (dipende sia dallo stato dell'ambiente esterno, sia da quello interno<sup>7</sup>)

*priorità* (0-10): 9

Ad ogni istante il comportamento con la priorità maggiore e le cui condizioni di attivazione sono soddisfatte assume il controllo e viene eseguito. Nel nostro esempio è probabile che all'inizio ci si trovi in uno stato in cui siano soddisfatte solo le condizioni di attivazione del comportamento "bere" (il caffè è pronto e ho sete). Diciamo allora che tale comportamento "emerge" da un punto di vista biologico rispetto agli altri. Per capire il paradigma reattivo, è importante rendersi conto che questa emergenza, tuttavia, non è una proprietà statica e globale del nostro modello. A seconda di come si evolve la situazione corrente, altri comportamenti – in contesti locali altrettanto prioritari – possono emergere. Nel momento in cui tocchiamo la tazza bollente e percepiamo una sensazione di calore bruciante localizzata nelle dita, diventano vere anche le condizioni di attivazione del comportamento "evitare ustioni di contatto" (ricordiamo che i due comportamenti, "bere" ed "evitare ustioni di contatto", sono in esecuzione parallela). In questa particolare situazione, il secondo comportamento emerge biologicamente rispetto al primo, avendo una più alta priorità. Questa emergenza biologica si concretizza senza alcuna pianificazione da parte nostra. Analogamente, se riconsideriamo l'esempio del robot che entra nell'edificio in fiamme per spegnere l'incendio, un comportamento sarà quello di "raggiungere l'incendio", mentre altri comportamenti plausibili, eseguiti concorrentemente, saranno quelli di "evitare gli ostacoli" e di "evitare danni termici alla propria struttura". Entrambi questi esempi mostrano la necessità di una rappresentazione locale del mondo che è un aspetto caratteristico del paradigma reattivo rispetto al paradigma gerarchico.

Se il paradigma reattivo presenta molte importanti migliorie secondo i punti di vista biologico, psicologico ed informatico-ingegneristico, ha ancora senso studiare il paradigma gerarchico? La risposta ci viene fornita dai risultati della ricerca degli ultimi anni. Nonostante il paradigma reattivo abbia diverse proprietà desiderabili, tra cui l'efficienza di esecuzione, la tendenza oggi è quella di preferire **architetture ibride di tipo deliberativo-reattivo**. In queste architetture, il robot per prima cosa pianifica (da cui il termine di deliberativo) il modo migliore di decomporre un task (una missione) in subtask (sotto-missioni). Dopo questa fase di pianificazione, il robot decide quali siano i comportamenti (behavior) da assumere per affrontare una missione e, di conseguenza, attiva una fase di tipo SENSING-ACTING. La fase di pianificazione viene eseguita in un singolo passo, mentre la fase di SENSING-ACTING combina in un solo passo primitive SENSE con primitive ACT, come illustrato in **figura 3**. L'organizzazione sensoriale in tali paradigmi, inoltre, sono una fusione degli stili gerarchico e reattivo.



**Figura 3 – Le tipiche fasi del paradigma ibrido**

Determinare il paradigma più adatto per realizzare il componente "intelligente" di un robot è un primo importante passo, ma da solo resterebbe un elemento teorico, quasi filosofico. Allo stesso modo in cui un algoritmo diventa eseguibile solo attraverso una sua implementazione in un linguaggio di programmazione, uno specifico paradigma si manifesta attraverso architetture software "rappresentative". Un'*architettura software* disciplina la distribuzione della logica di controllo all'interno di un sistema [Mataric, 1992], descrivendo un insieme di componenti architetturali e le loro relazioni, ossia il modo in cui essi interagiscono [Dean&Wellman, 1991]. Studiando delle architetture rappresentative nell'ambito dei paradigmi della robotica, possiamo imparare a progettare, modificare, ed

<sup>6</sup> In questo semplice esempio trascuriamo la durata del contatto

<sup>7</sup> Pensiamo ad esempio alla posizione fisica dell'essere umano e dell'oggetto

estendere i componenti architeturali per costruire robot artificiali autonomi ed intelligenti. Poiché uno degli obiettivi della robotica è proprio quello di imparare a *costruire* tali architetture, abbiamo bisogno di saper valutare una specifica architettura per poterla confrontare sia con architetture diverse, sia con problemi diversi, al fine di poter scegliere quella più adatta. Per valutare un'architettura software è necessario stabilire prima dei criteri. Nel caso della progettazione di un robot, stabiliamo a priori i seguenti criteri [Murphy, 2000][Arkin, 1998]:

**MODULARITÀ** – esprime il livello di autonomia di ciascun componente architeturale rispetto agli altri componenti. Un'architettura fortemente modulare è composta da moduli disaccoppiati tra loro, per cui la sostituzione di un modulo con un'implementazione diversa non influenza gli altri moduli: chiaramente un'ottima qualità per realizzare sistemi software flessibili.

**NICHE TARGETABILITY** – esprime il livello di adattamento che l'architettura software presenta rispetto alla specifica applicazione da costruire. Se un giorno decidessimo, ad esempio, che oltre alla navigazione autonoma, il nostro robot debba scattare delle foto e afferrare un particolare tipo di oggetto per spostarlo in uno speciale contenitore, quanto l'architettura inizialmente concepita esclusivamente per la navigazione si adatterebbe alla risoluzione di questi nuovi problemi?

**PORTABILITÀ** – esprime la facilità con la quale è possibile adottare l'architettura per applicazioni e robot concepiti in domini applicativi anche molto diversi tra loro. Se volessimo costruire un nuovo robot, ad esempio un braccio meccanico, quanto l'architettura originaria si adatterebbe al nuovo dominio?

**ROBUSTEZZA** – esprime una misura delle vulnerabilità (hardware e software) di un robot e delle soluzioni fornite dall'architettura per minimizzare o risolvere tali vulnerabilità.

I criteri descritti saranno un'utile guida nella successiva fase di design per scegliere un'architettura di base sulla quale iniziare a progettare il nostro sistema software per il controllo e la navigazione autonoma di un robot. Nei prossimi paragrafi descriveremo in dettaglio ciascun paradigma e le corrispondenti architetture rappresentative.

### Caratteristiche del paradigma gerarchico

Il paradigma gerarchico è caratterizzato dalla sequenza ordinata di fasi SENSING-PLANNING-ACTING descritta precedentemente. Possiamo immaginare il robot come una scatola nera (black-box) che, all'accensione, come prima cosa attiva tutti i sensori ed acquisisce da essi informazione (figura 4). La fase di SENSING nel paradigma gerarchico è monolitica: tutti i dati provenienti dai sensori vengono elaborati e fusi in un'unica rappresentazione interna globale che costituisce il modello del mondo (world model), ossia di quella porzione d'ambiente che consideriamo rilevante dal punto di vista del robot per l'adempimento della missione. Questo modello è tutto ciò che il robot conosce del mondo esterno (talvolta esso è riferito anche con il termine di "mappa" oppure di "universo").

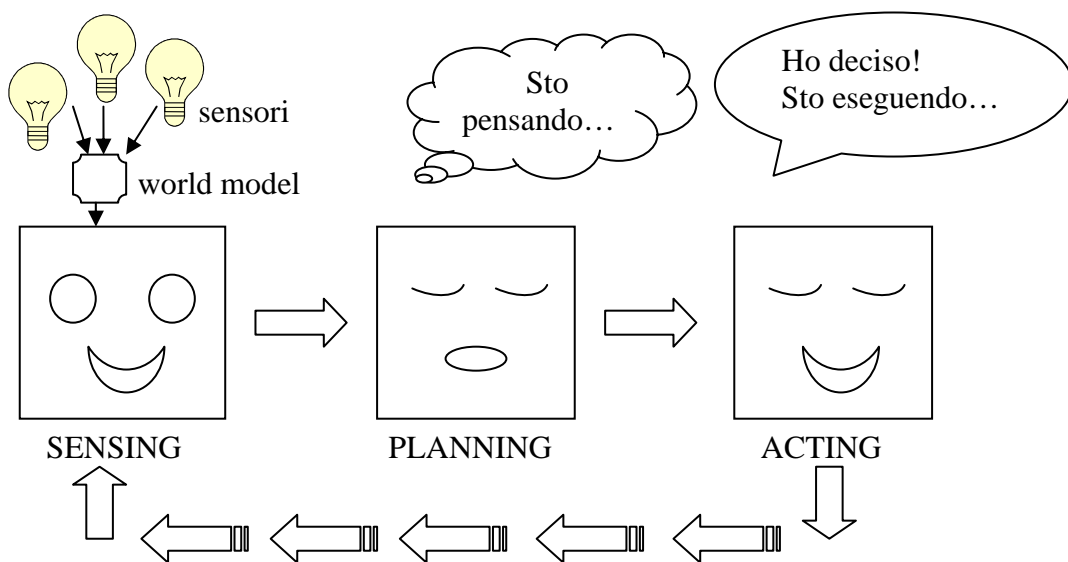


Figura 4 - Principali fasi del paradigma gerarchico

Una volta costruita la mappa, il robot passa dalla fase di SENSING a quella di PLANNING. Assumendo una visione antropomorfa del robot, possiamo immaginare che esso in questa fase chiuda gli occhi, smetta quindi di percepire dati dall'ambiente esterno, ed inizi invece a pianificare le direttive – i comandi da mandare agli attuatori - necessarie per raggiungere il goal, ossia l'obiettivo della missione (o di un suo specifico step). Per determinare quali direttive sia più conveniente eseguire viene sfruttato pesantemente il modello globale costruito nella fase di SENSING. Terminata la pianificazione, il robot esegue la direttiva

(o l'insieme di direttive) pianificata come miglior azione da compiere, attuando la fase di ACTING. Completata l'intera sequenza, il robot "riapre gli occhi" ed acquisisce nuovamente informazioni dai sensori. Ricordiamo che la fase di ACTING può indurre modifiche nell'ambiente per le quali è necessario aggiornare il modello del mondo – pensiamo ad esempio alla posizione relativa del robot che varia con il movimento dello stesso. Il ciclo descritto si ripete fino a completamento della missione.

Dato che il modello del mondo influenza la fase di pianificazione, dalla quale dipende il funzionamento del robot, cerchiamo di capire come esso si costruisce che tipo di informazioni contiene. Partendo da queste ultime, possiamo affermare che un modello del mondo nel paradigma gerarchico è in generale composto da:

1. Una *rappresentazione nota a priori dell'ambiente esterno* in cui il robot deve operare (ad esempio una mappa topografica delle strade e degli edifici, se il robot opera all'esterno; oppure una mappa dei corridoi e delle stanze, se esso opera in un ambiente chiuso);
2. I *dati acquisiti dai sensori* e le informazioni elaborate da essi (posizione corrente, orientamento, assunzione sulla posizione relativa rispetto ad altri punti di riferimento, temperatura, pressione, ecc.);
3. eventuale *conoscenza cognitiva aggiuntiva* che potrebbe essere necessaria per completare un task ("tutti i pezzi gialli che si trovano nella stanza di fronte devono essere ripartiti nei contenitori rossi per essere poi trasportati nella stanza ST13; i pezzi rossi sono invece degli scarti che vanno collocati nel bidone TRASH1").

Capiamo subito che creare un'unica rappresentazione globale in grado di memorizzare questa grande mole di informazioni è un compito non banale, senza considerare la quantità di risorse richieste, memoria e tempo di elaborazione su tutte. Nonostante il costante aumento della potenza di calcolo dei processori, un approccio gerarchico puro si è dimostrato abbastanza insoddisfacente per molti task che necessitano di tempi di risposta rapidi e di interazioni in un ambiente aperto, in costante cambiamento, come vedremo successivamente. Per comprendere meglio lo stile di risoluzione dei problemi e poter fare un adeguato confronto con il paradigma reattivo, tuttavia, può essere utile ragionare su un esempio concreto, attribuendo una missione di navigazione ad un ipotetico robot progettato secondo il paradigma gerarchico. Tenendo presente la necessità di costruire un modello globale del mondo, possiamo caratterizzare le missioni di navigazione sulla base dei seguenti aspetti:

- a. attribuzione di uno stato iniziale (*initial state*), ossia la posizione iniziale del robot nella mappa;
- b. attribuzione di uno stato finale o stato obiettivo (*goal state*), ossia la posizione finale desiderata che il robot deve raggiungere per considerare completata con successo la missione;
- c. valutazione della differenza corrente tra stato iniziale e stato finale, ossia "valutazione dell'incremento" effettuato dall'ultima azione nel tentativo di avvicinarsi alla soluzione finale.

Gli algoritmi di navigazione di questo tipo implementano una variante del metodo **General Problem Solver** chiamato **Strips [Barr&Feigenbaum, 1981]**. Strips è un algoritmo<sup>8</sup> di ricerca basato su una tecnica molto semplice detta *means-ends analysis*:

Se il robot non riesce a portare a termine il task (goal) in un'unica operazione (in questo caso, in un unico movimento o azione), allora è sufficiente esaminare tutte le possibili operazioni (movimenti o azioni), e scegliere quella che riduce maggiormente la distanza tra la posizione corrente e lo stato obiettivo finale che si vuole raggiungere. Minimizzare questa distanza significa proprio identificare l'incremento migliore che avvicina il robot alla soluzione desiderata. In altre parole, il ragionamento su cui si basa Strips è analogo alla strategia che normalmente utilizziamo quando cerchiamo di risolvere un puzzle (**figura 5**). Partiamo da una situazione iniziale totalmente disordinata, con i pezzi sparsi in ordine casuale sul tavolo. Non appena riconosciamo due pezzi che si incastrano perfettamente tra loro, iniziamo a costruire una parte della soluzione. Ad ogni passo intermedio, l'aggiunta di un ulteriore pezzo (o di un gruppo di pezzi) al puzzle costituisce un successivo incremento. La soluzione parziale, ottenuta mediante ogni incremento, è sempre "più vicina" alla configurazione di pezzi che costituisce la soluzione finale.

Da questa valutazione degli incrementi necessaria per scegliere di volta in volta quello più vantaggioso, scaturisce un processo di decisione che il robot deve compiere e che costituisce il cuore della fase di PLANNING. Caratterizziamo quindi il processo di decisione per mezzo di una particolare funzione che chiamiamo *operatore*. Diremo che il robot sceglie ad ogni passo l'operatore che riduce maggiormente la distanza rispetto alla soluzione finale. Esempi di operatori possono essere considerati comandi semplici o pattern di comandi complessi quali "vai alla porta", "attraversa la porta", "ruota di x gradi verso destra", "prosegui dritto", "afferra un oggetto".

Come possiamo capire, gli operatori dicono che cosa il robot può fare, ma dipendono spesso dal modello del mondo. L'operatore "attraversa la porta", ad esempio, potrebbe dipendere da un parametro "porta": infatti nella stanza in cui il nostro ipotetico robot si trova potrebbero esistere più porte, per cui diventa indispensabile poterne specificare una in particolare. Dobbiamo quindi creare questa *rappresentazione*

---

<sup>8</sup> Un algoritmo è una procedura per risolvere uno specifico problema contraddistinta da due proprietà molto importanti: deve essere corretta e deve terminare.

*interna* che possa essere *parametrica*. Ovviamente esistono diversi stili di rappresentazione. Strips utilizza la logica dei predicati e modella ogni elemento *rilevante* del mondo (rispetto al problema di navigazione) per mezzo di costrutti logici che chiamiamo fatti, o *assiomi*. Tali assiomi sono rappresentati appunto mediante predicati logici<sup>9</sup>. Un *predicato* è una funzione che fornisce in output un valore di verità (TRUE oppure FALSE).

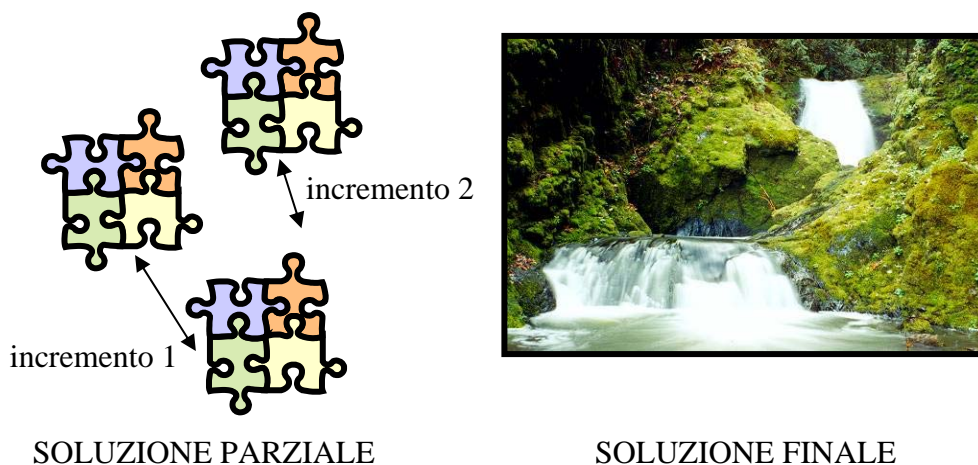


Figura 5 – Strategia di costruzione della soluzione finale per incrementi usata da Strips

Un predicato, essendo modellato come funzione, potrà prendere in input degli argomenti e questa caratteristica diventa molto comoda proprio per specificare i parametri quali il nome di una porta, quello di una stanza, il robot stesso e la sua posizione. Per evitare possibili confusioni di formalismo, nel resto del paragrafo adottiamo la convenzione di rappresentare i predicati con lettere maiuscole, mentre utilizziamo le lettere minuscole per le variabili che, in questo contesto, sono dei semplici contenitori (placeholder) ai quali non è stato assegnato ancora nessun valore<sup>10</sup>.

Partiamo ora dalla seguente situazione: abbiamo il nostro robot – che chiamiamo IT – il quale si trova nella stanza R1. L’obiettivo della missione di navigazione consiste nello spostare IT nella stanza R2. La **figura 6** illustra la mappa topografica dell’ambiente in cui IT opera.

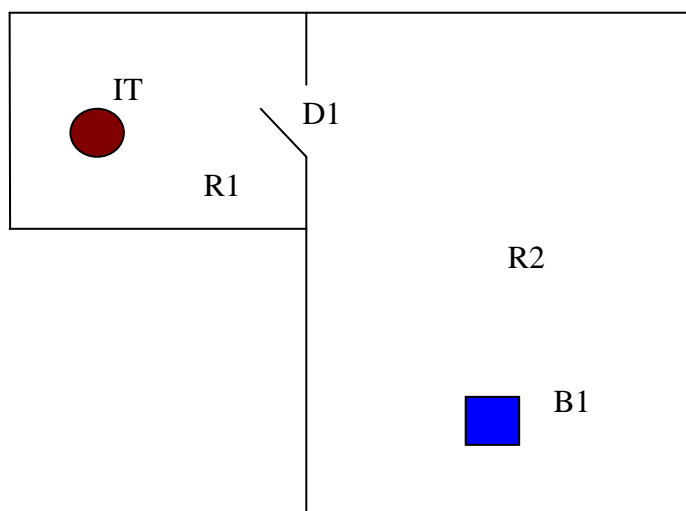


Figura 6 - Un esempio di rappresentazione del mondo (configurazione iniziale) in Strips

<sup>9</sup> Non è un caso che algoritmi come Strips fossero spesso codificati in linguaggi di programmazione logici come il PROLOG, i quali consentono di implementare in modo estremamente naturale fatti, regole, algoritmi di ricerca per individuare la regola “migliore” da utilizzare (matching tra regole), processi di induzione logica e di backtracking.

<sup>10</sup> Talvolta questo tipo di variabili vengono chiamate anche “variabili libere” o “variabili non vincolate”.

Il modello del mondo (world model) è, nel caso di Strips, una traduzione (se preferiamo, una codifica) in predicati logici della mappa di **figura 6**, dei suoi elementi e delle loro relazioni reciproche. Diremo, in particolare, che tale rappresentazione interna è costituita generalmente da conoscenza fattuale. La prima decisione da prendere, a questo punto, consiste nel definire l'insieme di fatti che il robot conosce e che può sfruttare per "ragionare" sul mondo. Supponiamo che tale conoscenza sia limitata nell'attestare se un oggetto mobile come IT (e che identifichiamo con la classe<sup>11</sup> *movable\_object*) è presente in una stanza, se esso è vicino ad una porta o ad un altro oggetto mobile, e se una porta è aperta o chiusa, unitamente all'informazione relativa alle due stanze collegate dalla porta stessa.

Traduciamo tale conoscenza fattuale in predicati logici:

```
INROOM(x, r) // x è di tipo movable_object, r è di tipo room
NEXTTO(x,t) // x è di tipo movable_object, t è di tipo movable_object oppure door
STATUS(d, s) // d è di tipo door, s è un tipo enumerato {OPEN,CLOSED}
CONNECTS(d, rx, ry) // d è di tipo door, rx e ry sono di tipo door
```

Con questo insieme di predicati possiamo costruire finalmente la nostra rappresentazione interna del mondo (world model) descritta in **figura 6**:

### initial state

```
INROOM(IT, R1)
INROOM(B1, R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1) // posso attraversare D1 in entrambe le direzioni!
STATUS(D1, OPEN)
```

Questi predicati descrivono la posizione iniziale del robot all'inizio della missione (ci sono due stanze R1 ed R2 separate da una porta D1, il robot IT si trova in R1 e D1 è aperta). Notiamo che il predicato NEXTTO non è stato utilizzato poiché non rappresenterebbe un fatto vero nella situazione di **figura 6**, in quanto non sarebbe possibile legare nessuna delle due variabili *x* e *t* con un qualche elemento del mondo nella configurazione iniziale.

Per valutare un incremento di soluzione, dobbiamo prima definire lo stato finale (goal), illustrato in **figura 7**:

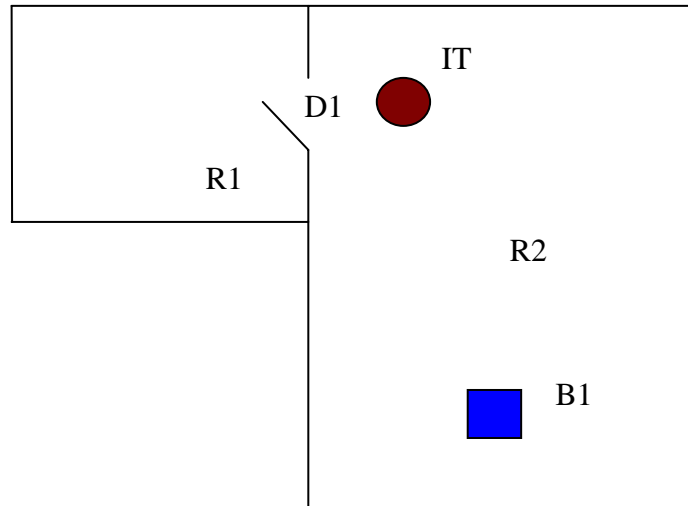
### goal state

```
INROOM(IT, R2)
INROOM(B1, R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1) // posso attraversare D1 in entrambe le direzioni!
STATUS(D1, OPEN)
```

Una volta costruita la mappa (world model), per poter valutare quale sarà la prossima mossa da compiere (ecco emergere la fase di PLANNING!) è necessario saper calcolare la differenza tra ogni possibile configurazione parziale e la configurazione finale. Per fare questo costruiamo una particolare tabella di supporto, riportata in **tabella 2**, e chiamata *tabella delle differenze*. In essa riportiamo tutte le operazioni che possono essere eseguite dal robot. Tali operazioni sono esempi di funzioni operatore precedentemente discusse. Ogni operatore è definito da una serie di condizioni che devono essere verificate prima della sua esecuzione. Tali condizioni, concettualmente analoghe alle condizioni di attivazione dei behavior nel paradigma reattivo, sono chiamate *precondizioni*. Gli ultimi due elementi che compongono la tabella delle differenze sono le cosiddette *add-list* e *delete-list*. Ogni volta che un operatore viene eseguito, ci potrebbe essere nuova conoscenza fattuale da aggiungere o da rimuovere (ricordiamo che il robot modifica l'ambiente anche con il suo semplice movimento!). Queste due liste vengono associate a ciascun operatore in modo che sia sempre semplice capirne gli effetti ed aggiornare conseguentemente il modello globale del mondo.

---

<sup>11</sup> Senza entrare ora nel dettaglio, una classe è un costrutto nei linguaggi orientati agli oggetti per definire tipi di dato astratto, ossia moduli che rappresentano entità astratte come un robot oppure una stanza, dotati di dati e di operazioni. Nel nostro esempio, IT e B1 sono due istanze di *movable\_object* che vengono sostituite alle variabili libere rappresentate dai parametri di un predicato. Tipizzare i parametri di un predicato risulta comodo per poter limitare le sostituzioni accettabili per ciascun parametro.



**Figura 7 - Un esempio di rappresentazione del mondo (configurazione finale) in Strips**

Come per i predicati, dobbiamo definire quali sono tutti gli operatori a disposizione nella programmazione del nostro robot. In questo esempio, tratto da [Murphy, 2000], immaginiamo che IT possa utilizzare solo i seguenti due operatori:

GOTODOOR(*x*, *d*) // sposta un oggetto *x* movable\_object in prossimità di una porta *d* door  
 GOTHRUDOOR(*x*,*t*) // un oggetto *x* movable\_object attraversa una porta *d* door

La tabella delle differenze a questo punto sarà la seguente:

Operatore	Precondizioni	Add-list	Delete-list
OP1: GOTODOOR(IT, dx)	INROOM(IT, rk) CONNECTS(dx, rk, rm)	NEXTTO(IT, dx)	
OP2: GOTHRUDOOR (IT, dx)	CONNECTS(dx, rk, rm) NEXTTO(IT, dx) STATUS(dx, OPEN) INROOM(IT, rk)	INROOM(IT, rm)	INROOM(IT, rk)

**Tabella 2 – Tabella delle differenze in Strips**

La tabella specifica che l'operatore GOTODOOR può essere eseguito solamente se valgono le seguenti precondizioni:

INROOM(IT, rk) – Il robot IT è in una stanza, la quale sarà poi assegnata alla variabile *rk*;  
 CONNECT(dx, rk, rm) – c'è una porta, che sarà assegnata alla variabile *dx* che mette in comunicazione due stanze, le quali saranno assegnate rispettivamente alle variabili *rk* e *rm*.

I predicati che esprimono le precondizioni sono stati vincolati solamente nella variabile *x* relativa a INROOM, la quale è stata legata mediante l'etichetta IT che rappresenta il nostro robot. Le altre variabili sono per il momento ancora libere.

Analogamente a quanto visto per GOTODOOR, l'operatore GOTHRUDOOR può essere eseguito solo se:

- i. il robot si trova in una stanza - INROOM(IT, *rk*);
- ii. il robot si trova in prossimità di una porta - NEXTTO(IT, *dx*);
- iii. la porta in questione è aperta - STATUS(*dx*, OPEN);
- iv. la porta in questione mette in comunicazione con un'altra stanza - CONNECTS(*dx*, *rk*, *rm*)

Dall'esame delle liste add-list e delete-list, inoltre, possiamo vedere come viene rappresentato il fatto che, una volta attraversata una porta, il robot si trova nella stanza successiva (inserimento di un predicato nella add-list di GOTHRUDOOR) e non è più presente nella stanza precedente (inserimento di un predicato nella delete-list di GOTHRUDOOR). Queste due liste riflettono quindi le conoscenze fattuali che vanno rispettivamente aggiunte ed eliminate dal modello del mondo per mantenerlo aggiornato. Definiti gli operatori, il modello del mondo e i predicati che rappresentano la conoscenza fattuale, possiamo finalmente esprimere ogni incremento di soluzione mediante una differenza. Ad esempio, la differenza tra lo stato iniziale e quello finale (goal) può essere scritta ora nel seguente modo:

```
INROOM(IT, R2) = FALSE
```

che può a sua volta essere espresso variando leggermente il formalismo così:

```
NOT(INROOM(IT, R2))
```

Se non ci fosse questa differenza, ossia se `INROOM(IT, R2)` fosse soddisfatta, il robot IT si troverebbe nella stanza R2 e la missione sarebbe conclusa. Così non è e allora Strips prova diverse sostituzioni di valori alle variabili di ciascun operatore per trovare quello che definisce l'incremento migliore, ossia quello che riduce la distanza tra la situazione iniziale e il goal. Vediamo come. Per prima viene consultata la tabella delle differenze. In essa vengono esaminati tutti gli operatori disponibili, partendo dall'alto verso il basso, secondo la colonna della add-list. Viene considerata questa colonna perché essa rappresenta l'effetto dell'esecuzione di ciascun operatore, ossia rappresenta proprio l'incremento da aggiungere allo stato precedente. Se Strips, esaminando la colonna add-list, trova un operatore che produce il goal state richiesto, allora tale operatore elimina del tutto la differenza tra stato iniziale e stato finale, che ricordiamo essere `INROOM(IT, R2)`. L'operatore OP2: GOTHRUDOOR sembra essere un buon candidato incremento poiché nella sua add-list troviamo proprio:

```
INROOM(IT, rm)
```

Se `rm = R2`, allora il risultato di OP2 sarà esattamente `INROOM(IT, R2)`, il che eliminerebbe del tutto la differenza e la missione sarebbe conclusa. Strips, quindi, verifica le precondizioni di OP2, esaminando la corrispondente colonna. Per fare questo, l'etichetta R2 deve essere sostituita ad ogni occorrenza della variabile `rm`. L'unica precondizione che è influenzata da `rm` è `CONNECTS(dx, rk, rm)` che diventa allora: `CONNECTS(dx, rk, R2)`. Finché le variabili `dx` ed `rk` non saranno legate (vincolate) a qualche altra etichetta, non sarà possibile attribuire alcun valore di verità al predicato `CONNECTS`. In altre parole, `dx` ed `rk` rappresentano al momento dei caratteri jolly:

```
CONNECTS(*, *, R2)
```

Per legare queste due variabili, Strips esamina il modello del mondo corrente e cerca delle sostituzioni plausibili. L'unico predicato della rappresentazione globale del mondo che può essere messo in corrispondenza con `CONNECTS(*, *, R2)` è il seguente:

```
CONNECTS(D1, R1, R2)
```

Per effetto di questo predicato, Strips lega `dx` a D1 e `rk` a R1. Più formalmente scriviamo:

```
dx → D1; rk → R1
```

Arrivato a tal punto, Strips propaga i legami appena stabiliti alla successiva precondizione dell'operatore OP2: `NEXTTO(IT, dx)` che diventa:

```
NEXTTO(IT, D1) // attenzione: precondizione falsa!!!
```

Quest'ultima precondizione è falsa poiché il predicato `NEXTTO(IT, D1)` non è contenuto nel modello corrente del mondo. Concettualmente possiamo interpretare questa situazione nel seguente modo: per completare la missione è necessario eseguire l'operatore OP2: GOTHRUDOOR, ma prima di fare questo bisogna spostare il robot IT nei pressi della porta D1. Per tenere traccia di questo fatto, Strips usa la tecnica della *ricorsione*. Come prima cosa Strips marca il goal state originale con l'etichetta G0, poi lo inserisce in una struttura dati stack (pila) e crea un nuovo sub-goal state che chiama G1. Tale nuovo goal G1 è proprio la precondizione fallita `NEXTTO(IT, D1)`, come illustrato in **figura 8**. In altre parole, Strips con lo stack tiene traccia del goal iniziale G0 che lo porterebbe a risolvere il problema. A questo punto il nuovo obiettivo diventa risolvere il sottoproblema G1. Una volta risolto questo task, G1 può essere rimosso dalla pila e Strips avrà tutti gli elementi per risolvere G0. Risolto G0, la missione di navigazione sarà terminata.

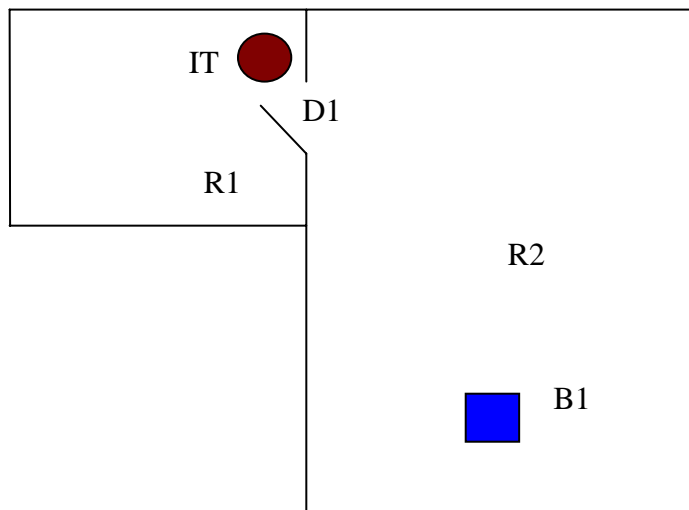
G1: NEXTTO(IT, D1)
G0: INROOM(IT, R2)

**Figura 8 - Stack dei sub-goal (o degli incrementi)**

Come fa Strips a risolvere G1? Semplice: ripete in modo ricorsivo esattamente la stessa procedura utilizzata per tentare di risolvere G0. Viene consultata la tabella delle differenze, partendo sempre dall'alto e guardando la colonna add-list. Questa volta, però, viene cercato un operatore che corrisponda al nuovo sotto-obiettivo G1. Il match viene trovato nell'operatore OP1: GOTODOOR(IT, dx). Sfruttando il meccanismo di sostituzione delle variabili, viene creato il legame  $dx=D1$ . Dopodiché bisogna verificare le precondizioni. In questo caso vengono assegnati i legami:  $rk \rightarrow R1$  e  $rm \rightarrow R2$  che soddisfano entrambe le precondizioni di OP1. Finalmente Strips può eseguire l'operatore OP1, la cui esecuzione produce un cambiamento nel modello corrente del mondo. Per effetto dell'esecuzione di GOTODOOR, infatti, il robot si trova ora nei pressi della porta D1, situazione descritta mediante l'aggiunta nella rappresentazione globale del mondo del predicato NEXTTO(IT, D1), in accordo alla tabella delle differenze. Lo stato dopo l'esecuzione di OP1 è ora il seguente, illustrato anche in **figura 9**:

**State after OP1**

```
INROOM(IT, R1)
INROOM(B1, R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1) // posso attraversare D1 in entrambe le direzioni!
STATUS(D1, OPEN)
NEXTTO(IT, D1) // incremento di soluzione per effetto dell'esecuzione di OP1
```



**Figura 9 – Rappresentazione del mondo dopo l'esecuzione dell'operatore GOTODOOR**

A questo punto il sottogoal G1 è stato soddisfatto e viene rimosso dallo stack. Il controllo passa nuovamente alla precedente chiamata in cui Strips cercava di verificare il goal G0, esattamente nello stesso punto in cui era stato sospeso per attivare G1, ossia nella valutazione della precondizione NEXTTO(IT, D1). Tale precondizione, per effetto dell'esecuzione di OP1, è adesso soddisfatta (era l'obiettivo G1). Essendo verificate tutte le precondizioni di OP2, Strips ne pianifica l'esecuzione, cambia la rappresentazione interna del mondo aggiungendo il predicato INROOM(IT, R2) (si veda la corrispondente colonna add-list) e rimuove il predicato INROOM(IT, R1) (si veda la colonna delete-list). Il nuovo stato è il seguente:

**State after OP2**

```
INROOM(IT, R2)
INROOM(B1, R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1) // posso attraversare D1 in entrambe le direzioni!
STATUS(D1, OPEN)
NEXTTO(IT, D1) // incremento di soluzione per effetto dell'esecuzione di OP1
```

Strips ricalcola la differenza tra stato corrente e stato finale, valuta tale differenza come nulla e quindi termina la propria esecuzione.

In conclusione, lo stile di risoluzione di Strips è marcatamente orientato verso la pianificazione, piuttosto che verso l'esecuzione. Independentemente dal linguaggio di programmazione usato, Strips richiede allo sviluppatore di costruire:

- i. una rappresentazione interna globale del mondo;
- ii. la tabella delle differenze con gli operatori, le precondizioni, le liste add-list e delete-list;
- iii. una funzione valutatore di differenze

L'algoritmo può essere schematizzato dai seguenti passi:

1. Calcola la differenza tra stato iniziale e stato finale utilizzando la funzione valutatore di differenze;
2. Se la differenza è nulla, termina;
3. Se la differenza è non nulla, riducila selezionando il primo operatore dalla tabella delle differenze nella cui add-list ci sia un predicato che neghi la differenza;
4. Per tale predicato, esamina le precondizioni per vedere se può essere identificato un insieme di sostituzioni plausibili per le variabili, tale che tutte le precondizioni siano soddisfatte. Altrimenti prendi la prima precondizione che fallisce e falla diventare un nuovo goal, salvando sullo stack il goal precedente. Applica quindi in modo ricorsivo i passi 3 e 4;
5. Quando tutte le precondizioni per un operatore sono soddisfatte, pianifica l'esecuzione di quell'operatore ed aggiorna la rappresentazione interna del mondo (attendendo che l'operatore venga poi effettivamente eseguito).
6. Ritorna al precedente operatore con almeno una precondizione fallita ed eseguillo o, se ci sono altre precondizioni non soddisfatte, fai ripartire il processo di ricorsione.

Come possiamo renderci conto, tutta questa enfasi nella pianificazione e nella costruzione ed aggiornamento del modello del mondo richiede memoria, memoria, memoria. Nonostante sia possibile concepire rappresentazioni diverse da quella usata nel nostro esempio, questa caratteristica rappresenta spesso una forte limitazione dell'approccio puramente gerarchico. La memoria, inoltre, non è il solo punto debole di tale paradigma. Molto spesso ipotizzare che il robot operi sempre in un mondo chiuso, fortemente prevedibile e preordinato, è un'assunzione troppo forte che origina il noto "frame problem", come vedremo nel prossimo paragrafo.

### Assunzione del mondo chiuso e "frame problem"

Nella robotica due concetti molto ricorrenti sono l'assunzione del mondo chiuso e il frame problem. Letteralmente "frame" significa telaio, ossatura, cornice, struttura. Nel nostro contesto, esso identifica il tentativo di rappresentare una situazione del mondo reale il più fedelmente possibile (ai fini della risoluzione di un problema), pur mantenendo computazionalmente trattabile il risultato finale. Tale rappresentazione interna è spesso influenzata da alcune assunzioni. Come abbiamo potuto vedere nell'esempio di Strips, il paradigma gerarchico è caratterizzato dall'**assunzione del mondo chiuso**:

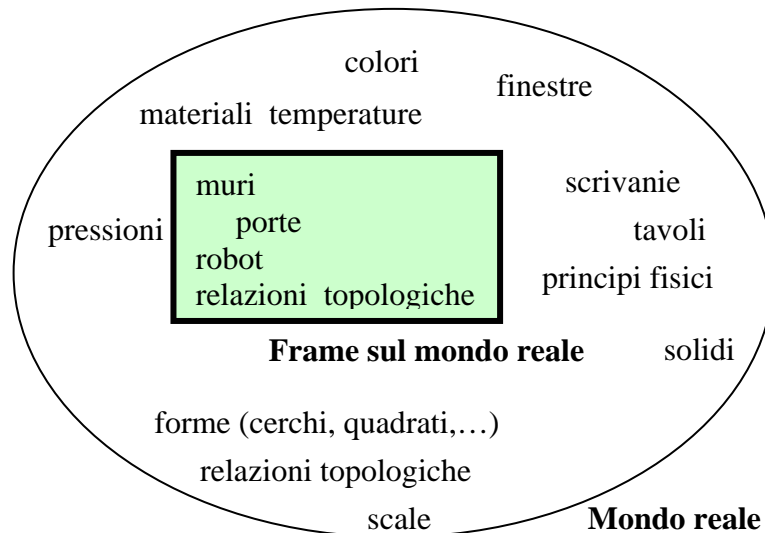
*Il modello del mondo (world model) contiene una descrizione completa ed esaustiva di tutti gli elementi del mondo reale di cui il robot ha bisogno per completare la sua missione.*

Il frame problem è all'assunzione di mondo chiuso possono essere esemplificati dalla **figura 10** nella quale tutto ciò che il robot conosce del mondo esterno è rappresentato all'interno del frame. Tutto ciò che sta al di fuori di questa "cornice" sul mondo reale è inaccessibile al robot e, quindi, è come se non esistesse.

Se l'assunzione di mondo chiuso viene violata, il robot potrebbe non funzionare più correttamente, ossia il suo comportamento potrebbe non essere più prevedibile. Si tratta indubbiamente di un grosso problema, soprattutto nel paradigma gerarchico poiché uno dei suoi innegabili punti di forza è costituito proprio dalla capacità di *spiegare* sempre perché il robot ha compiuto una certa azione e, al contempo, di *prevedere* la successiva azione che esso eseguirà<sup>12</sup>. Il comportamento deterministico fortemente prevedibile è stato una delle più importanti ragioni sia del successo del paradigma gerarchico, sia dell'iniziale scetticismo riservato da molti esperti di robotica nei confronti del paradigma reattivo. Come vedremo, quest'ultimo abbandona completamente l'assunzione che un robot, per funzionare correttamente, debba conoscere *tutti* gli elementi del mondo reale, soprattutto perché rappresentarli formalmente non è sempre intuitivo, oltre che economicamente vantaggioso. Dopotutto, in una situazione reale esistono moltissimi elementi.

---

<sup>12</sup> Non è detto, tuttavia, che questa capacità predittiva sia sempre facile da elaborare: essa dipende dalla complessità delle rappresentazioni interne del modello del mondo e dagli strumenti di manipolazione del modello stesso, come il calcolo dei predicati in Strips.



**Figura 10 - Assunzione di mondo chiuso e “frame problem” nel paradigma gerarchico**

L’ingenua nozione di “stanza” e di “oggetto” emersa nella discussione di Strips costituisce un evidente esemplificazione. In una situazione più realistica dovremmo distinguere oggetti statici da oggetti in movimento, materiali, superfici, stanze, temperature, pressioni, effetti di illuminazione e così via. Possiamo facilmente immaginare che, se per due stanze e due oggetti generici, la complessità della rappresentazione interna utilizzata in Strips è ancora trattabile, la situazione diventa ben presto critica qualora volessimo rappresentare fedelmente un ambiente reale come il mio studio, nel quale il nostro robot IT dovrebbe essere in grado di muoversi liberamente ed autonomamente.

### Architetture rappresentative del paradigma gerarchico

Le due più importanti architetture rappresentative del paradigma gerarchico sono, rispettivamente, il Nested Hierarchical Controller (NHC), sviluppata da Meystel [Meystel, 1990], e la più recente NIST Real-time Control System (RCS), sviluppata da Albus [Albus&Proctor, 1996]. In questo paragrafo ci concentreremo sulla prima architettura. Il lettore interessato ai lavori di Albus può consultare, oltre alla citata reference, anche il testo di Robin Murphy “Introduction to AI Robotics” riportato in bibliografia, nel quale viene fornita un’introduzione al NIST RCS.

Per rappresentare un’architettura software esistono diverse notazioni grafiche. In questo progetto utilizzerò UML [Booch et al., 1999] per illustrare aspetti di design logico quali la suddivisione in layer (o livelli) e la decomposizione di un sistema in componenti più elementari (package e classi). Indubbiamente la scelta di usare UML, seppure sia ormai uno standard de-facto nella progettazione dei moderni sistemi software, tradisce una forte inclinazione verso linguaggi di programmazione orientati ad oggetti come Java e C++. Considerati i sistemi reattivi e ibridi costruiti oggi in robotica, questa non è sicuramente un’assunzione restrittiva. Può invece essere considerata una forzatura se pensiamo ai primi sistemi gerarchici come Strips, scritti in linguaggi come Lisp, PROLOG, FORTRAN o, nella migliore delle ipotesi, C. In questi linguaggi non esiste il concetto di classe e, ad esclusione del C, neppure il costrutto della ricorsione. Con un po’ di abuso di notazione, comunque, il lettore può pensare ad una classe come ad un modulo, ovvero ad una struttura dati unita all’insieme di operazioni che accedono ad essa. Grazie a questa assunzione, possiamo utilizzare un’unica notazione grafica – UML – per descrivere aspetti logici di design per tutte le architetture di riferimento dei tre paradigmi esaminati nel presente lavoro.

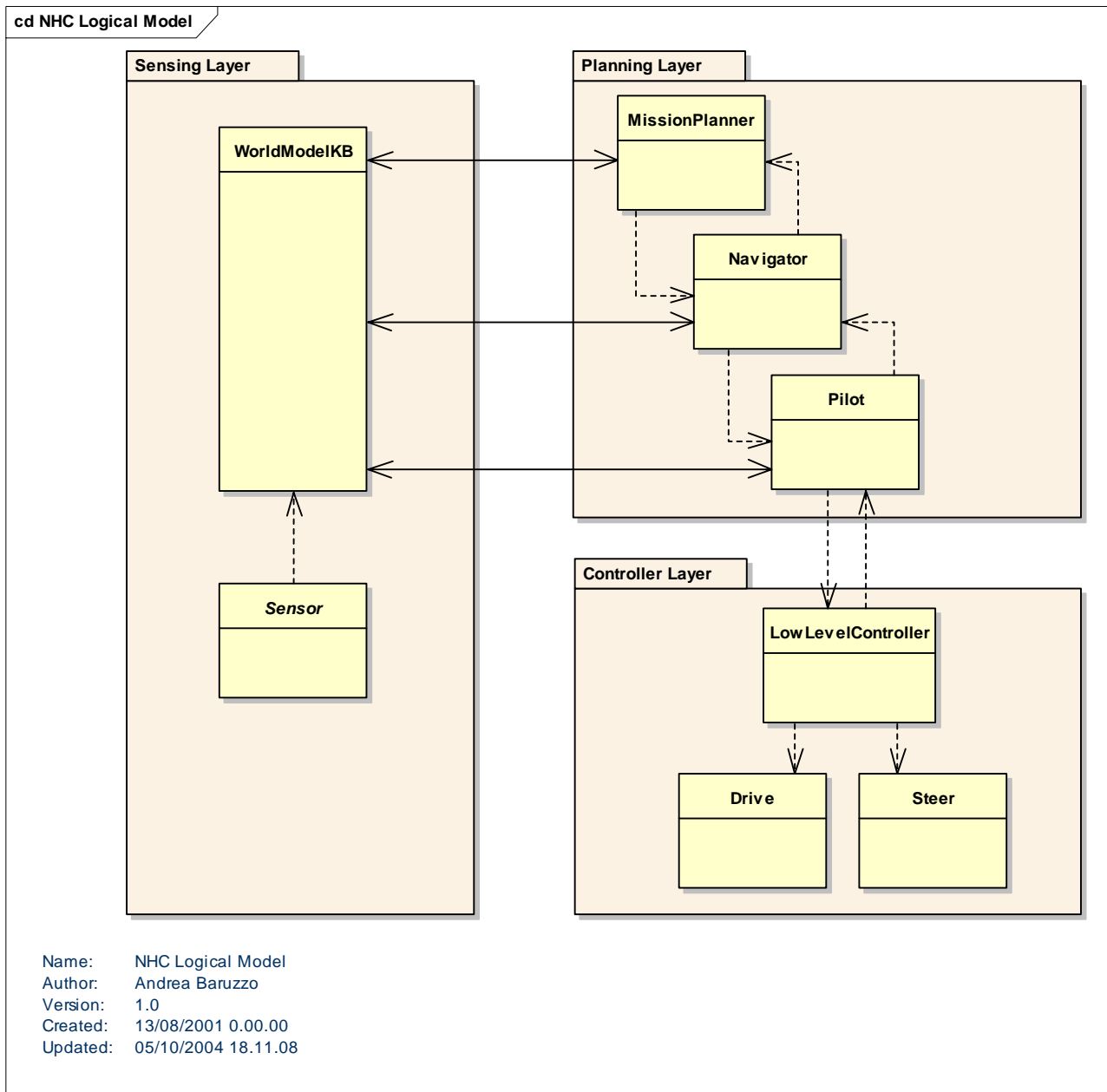


Figura 11 - Modello logico di un'architettura NHC

Fatta questa premessa, esaminiamo l'architettura dei sistemi NHC. Un sistema basato su NHC è costituito dai seguenti tre principali componenti (figura 11):

1. *Sensing Layer*, dedicato alla ricezione degli input dai sensori, alla rappresentazione interna del mondo (world model) e al mantenimento della Knowledge Base;
2. *Planning Layer*, dedicato alla gestione della missione e, più in particolare, alla pianificazione;
3. *Controller Layer*, dedicato alla gestione dei servo meccanismi, dei motori e degli attuatori.

Come possiamo intuire, a ciascuno dei precedenti componenti è possibile abbinare un tipo di primitive (SENSE, PLAN e ACT rispettivamente). Il robot inizia a raccogliere i dati provenienti dai sensori e li combina per formare il world model utilizzando primitive di tipo SENSE. Il modello interno del mondo contiene anche conoscenza a priori sulle entità dell'ambiente esterno, come descritto nei paragrafi precedenti. Dopo che tale modello è stato creato (o aggiornato), il robot pianifica quali azioni deve compiere. La pianificazione per la navigazione è realizzata mediante una procedura costituita da tre passi principali, eseguiti rispettivamente dai moduli *MissionPlanner*, *Navigator* e *Pilot*. Ognuno di questi moduli ha accesso alla knowledge base e al world model per completare le proprie attività, come illustrato dalle frecce di associazione di figura 11. Ovviamente un'attività di pianificazione richiede l'interazione delle tre componenti in una sequenza ben precisa, illustrata questa volta dalle frecce di dipendenza che legano *MissionPlanner* a *Navigator* e *Navigator* a *Pilot*. Il componente *Pilot*, infine, può essere visto come

interfaccia verso i controllori degli apparati fisici (motori, attuatori, servo meccanismi). Esso genera specifici comandi (primitive di tipo PLAN) da inviare al modulo *LowLevelController* che, a sua volta, traduce questi ultime in segnali di controllo diretti all'attuatore corrispondente. Tali segnali di controllo rappresentano primitive di tipo ACT.

Il maggiore contributo dell'architettura NHC è costituito da una netta decomposizione della pianificazione in tre differenti funzionalità (e relativi sottosistemi), concepiti allo scopo di supportare le attività di pianificazione. Tale decomposizione, inoltre, facilita la distribuzione del controllo sull'intera architettura. Il componente *MissionPlanner* riceve una missione dall'operatore umano oppure ne genera una per conto proprio. Il compito di questo componente è quello di tradurre la missione in rappresentazioni che siano eseguibili dagli altri componenti. Per far questo, esso accede alla mappa topologica, localizza la posizione attuale del robot e riformula il goal della missione in termini della mappa stessa. Tale informazione viene poi spedita al *Navigator* il quale genera un percorso dalla posizione corrente alla posizione obiettivo. Tale percorso è infine inviato al *Pilot*. Compito di quest'ultimo è esaminare ogni singolo tratto del percorso e, in particolare per il primo tratto, determinare quali comandi inviare al controller. Una volta che tali comandi sono stati eseguiti dal controller, il robot aggiorna il world model ed interroga nuovamente i sensori, iniziando un nuovo ciclo SENSING – PLANNING – ACTING.

Le architetture ispirate al sistema NHC presentano diversi vantaggi. La distribuzione dell'intelligenza è ripartita tra *MissionPlanner*, *Navigator* e *Pilot*. *MissionPlanner* è più intelligente di *Navigator* che, a sua volta, è più intelligente di *Pilot*. Esiste inoltre una continua interazione tra questi tre componenti, a differenza di Strips nel quale primitive PLAN e primitive ACT non erano mai intercalate le une alle altre. Nell'architettura NHC, invece, ogni percorso è caratterizzato da tratti e da punti intermedi, chiamati *waypoint*. Ogni tratto è costituito da due *waypoint*: il punto iniziale e il punto finale. Se il robot ha raggiunto un *waypoint*, il *Pilot* informa il *Navigator*. Se tale *waypoint* non corrisponde alla posizione obiettivo, il *Navigator* invia al *Pilot* il tratto di percorso successivo. In caso contrario, il *Navigator* informa il *MissionPlanner* che il robot ha raggiunto il goal. Il *MissionPlanner*, a sua volta, può considerare la missione completata, oppure può generare un nuovo goal. Se durante il movimento il robot incontra un ostacolo lungo il percorso, il *Pilot* aggiorna il world model ed informa dell'evento il *Navigator*, il quale ha il compito di generare un nuovo cammino per raggiungere la posizione obiettivo sulla base del world model aggiornato. Il nuovo cammino viene inviato quindi al *Pilot*. Queste interazioni spiegano le frecce di dipendenza dal *Pilot* al *Navigator* e dal *Navigator* al *MissionPlanner* (le frecce in senso inverso dovrebbero essere ovvie). Da tale distribuzione dell'intelligenza possiamo intuire che il *MissionPlanner* risolve i compiti ad un più alto livello d'astrazione, delegando al *Navigator* problemi di pianificazione del percorso che, a sua volta, delega al *Pilot* i compiti di invio dei comandi ai controllori.

Un chiaro svantaggio di questa architettura, invece, è che essa è espressamente concepita per missioni di navigazione. Attività di *picking* (afferrare e manipolare un oggetto), ad esempio, sono difficilmente mappabili facilmente nei componenti principali, limitandone la ricusabilità. Data la tipologia di missioni che il nostro robot dovrà svolgere, ossia principalmente missioni di navigazione, mi è sembrato opportuno esaminare questa architettura come architettura rappresentativa del paradigma gerarchico. Compresi ora i pregi e i difetti del paradigma gerarchico, possiamo ad esaminare più in dettaglio il paradigma reattivo.

## II paradigma reattivo e architetture rappresentative

#### da scrivere

## II paradigma ibrido "deliberativo-reattivo" e architetture rappresentative

#### da scrivere

## Comparazione e scelta del paradigma per il progetto di navigazione autonoma

#### da scrivere

## Bibliografia

- (1) [Albus&Proctor, 1996] Albus, J.; Proctor, F.G. – "A Reference Model Architecture for Intelligent Hybrid Control Systems", proceedings of the International Federation of Automatic Control, San Francisco, CA, June 30-July 5, 1996
- (2) [Arkin, 1998] Arkin, R. – "Behavior-Based Robotics", MIT Press, 1998
- (3) [Barr&Feigenbaum, 1981] Barr, A.; Feigenbaum, E. – "The Handbook of Artificial Intelligence", Vol 1, William Kaufmann, Inc., Los Altos, CA, 1981
- (4) [Booch et al., 1999] Booch, Grady; Rumbaugh, James; Ivacobson, Ivar – "The Unified Modeling Language User Guide", Addison Wesley, 1997

- (5) [Dean&Wellman, 1991] Dean, T.; Wellman, M. – “*Planning and Control*”, Morgan Kaufmann Publishers, San Mateo, CA, 1991
- (6) [Mataric, 1992] Mataric, M. – “*Behavior-Based Control: Main Properties and Implications*”, proceedings of Workshop on Intelligent Control Systems, IEEE International Conference on Robotics and Automation, Nice, France, 1992
- (7) [Meystel, 1990] Meystel, A. – “*Knowledge-Based Nested Hierarchical Control*”, in *Advances in Automation and Robotics*, vol. 2, Ed. G. Saridis, JAI Press, Greenwich, CT, 1990, pp. 63-152
- (8) [Murphy, 2000] Murphy, Robin R. – “*Introduction to AI Robotics*”, MIT Press, 2000