ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Sviluppo di un Framework per la Programmazione di Robot basati su Architettura di Controllo Behaviour-based

TESI IN
Programmazione Concorrente e Distribuita L-M

CANDIDATO
Matteo Bianchi

RELATORE
Prof. Alessandro Ricci

CORRELATORE
Prof. Andrea Roli

Terza Sessione
Anno Accademico 2011/2012

Indice

Introduzione

In merito al campo della robotica la situazione a cui stiamo assistendo oggigiorno presenta molte similitudini con la fase attraversata dal business dell'informatica e dei computer circa trent'anni fa. Lo stesso Bill Gates, uno dei principali protagonisti della rivoluzione del personal computer che ha visto la luce intorno alla metà degli anni '70, afferma di riconoscere nel periodo storico corrente parecchi dei tratti che hanno caratterizzato quegli anni. La situazione fino ad allora vedeva il mondo dell'informatica di consumo limitato a grossi e costosissimi mainframe che solamente grandi aziende, industrie ed università potevano permettersi. Col passare del tempo il progresso tecnologico ha velocemente portato a miniaturizzare e rendere sempre più accessibili e pervasive queste tecnologie fino a portare un computer sulla scrivania di ogni casa. E il processo non si è certamente fermato lì.

Quello che Bill Gates ricorda in un articolo su *Scientific American* [8] è il periodo in cui lui e Paul Allen diedero vita alla *Microsoft*, una delle realtà che ha certamente seguito, se non guidato, molto da vicino questi avvenimenti. Secondo lui ora il mercato della robotica è lentamente soggetto alla stessa convergenza tecnologica da cui l'informatica a tratto enormi vantaggi nei passati decenni e arriverà prima o poi anch'esso a portare un robot in ogni casa.

Discutendo con gli specialisti del settore ha riconosciuto lo stesso entusiasmo che si respirava “ai suoi tempi” ma contemporaneamente anche le stesse delusioni e difficoltà. Tutto ciò è fisiologico nel campo delle nuove tecnologie. Mentre nuove applicazioni si diffondono velocemente, non sono ancora delineate delle “best practice” né tantomeno degli standard sui quali appoggiarsi. Certamente per una scienza vasta, complessa e multidisciplinare come la robotica tale aspetto può addirittura farsi sentire con ancora più vigore. Ogni

azienda che operi in questo settore lo fa seguendo il proprio “expertise” e le proprie strategie dando vita ad una moltitudine di soluzioni spesso del tutto incompatibili le une con le altre. Ma come ogni processo innovativo insegna, è solo questione di tempo prima che si arrivi ad accettare comunemente delle linee guida e delle piattaforme tecnologiche che agevolino la diffusione e l'evoluzione di questo mercato ormai sulla via del decollo.

Dal canto suo la tecnologia dei dispositivi hardware compie continuamente grandi progressi portando le dimensioni e i costi di dispositivi quali servomotori, telecamere, sensori di prossimità, ricevitori GPS e così via, a livelli tali da permettere la costruzione di robot ben equipaggiati a costi relativamente accettabili. Ne è un chiaro esempio il numero di robot domestici che sono già presenti sul mercato: a prezzi non troppo elevati si possono infatti trovare robot giocattolo che simulano animali, robot di servizio per pulire e lavare i pavimenti, per tosare l'erba del giardino, per pulire i fondali delle piscine e per rimuovere le foglie dalle grondaie. Ovviamente quello dei robot domestici è solo l'applicazione più recente della robotica. Tante altre tipologie di sistemi robotici sono state sviluppate in precedenza, a partire dalle macchine automatiche usate nelle catene di montaggio che possono essere considerate i veri e propri progenitori di questi sistemi, al pari di quanto lo siano stati i mainframe per i personal computer.

Mentre le capacità dell'hardware aumentano, il software di controllo di questi robot invece progredisce più lentamente [8]. Nel corso degli anni diverse strategie sono state sviluppate dai ricercatori e tanti altri studi devono senz'altro essere ancora approfonditi ma riuscire a sviluppare sistemi di controllo che permettano ad un robot di esibire un comportamento “intelligente” è senza meno una delle sfide più competitive in questa materia, se non la più difficile. Uno dei grandi problemi che il software di controllo si è trovato a dover affrontare è la reattività ai continui cambiamenti del mondo reale, soprattuto quando il robot si trova a lavorare in ambienti sconosciuti e a stretto contatto con gli esseri umani, anche per evidenti motivi di sicurezza. Ovviamente questo problema deve essere affrontato con molta decisione perché la visione di una dif-

fusione capillare di queste tecnologie, soprattutto per uso domestico, possa divenire realtà. Il primo tentativo in questa direzione risale intorno alla metà degli anni '80 quando è stato introdotto un paradigma di progettazione dei sistemi di controllo denominato *behavior-based* proprio con lo scopo di cercare di affrontare questi problemi assumendo un'ottica più vicina al comportamento assunto dagli esseri viventi che alle classiche tecniche di intelligenza artificiale fino al quel momento adottate in robotica.

Lo scopo di questa tesi è effettuare uno studio approfondito di tale approccio alla programmazione di robot soprattutto come strategia per l'implementazione di sistemi di controllo per robot autonomi di uso quotidiano, perseguendo la visione appena discussa.

Successivamente viene introdotto un framework molto semplice come contributo in supporto alla formalizzazione e alla creazione di questo tipo di sistemi che possa essere usato efficacemente dagli sviluppatori per impostare lo scheletro, la base, su cui appoggiarsi per l'implementazione dei controllori per i loro specifici robot. Il suddetto framework viene progettato con un occhio di riguardo alla fase di simulazione dei questi sistemi che ricopre un'importanza sempre maggiore nel relativo processo di sviluppo. In questo caso ci si appoggia ad un simulatore commerciale che prende il nome di *Webots*.

Si prosegue poi con la risoluzione di un caso di studio notoriamente affrontato in letteratura per mettere alla prova la flessibilità e l'efficacia delle possibili soluzioni realizzabili sulla base del framework introdotto.

Infine vengono fatte delle considerazioni sugli esiti di questo esperimento e si riflette su eventuali sviluppi futuri considerando anche alcuni risultati ottenuti in ambito di ricerca e discussi in letteratura.

Capitolo 1
Robotica e architetture di controllo

Il focus principale attorno al quale si estende questa tesi riguarda la programmazione dei robot. Di conseguenza assumeranno una importanza centrale le varie strategie che possono essere impiegate per strutturare ed organizzare il software di controllo, nonché per indirizzarne le fasi di progettazione e di implementazione. Prima di addentrarsi in una impegnativa analisi di questo argomento si danno però alcune nozioni di base riguardanti la robotica nel suo complesso e alcuni importanti concetti che permeano l'intera trattazione. Dopo di che si analizzano i possibili approcci per lo sviluppo di sistemi di controllo, uno dei quali verrà poi dettagliatamente studiato nel capitolo successivo.

1.1 Introduzione alla robotica

Il termine “robot” è stato usato per la prima volta nel 1921 dallo scrittore ceco Karel Čapek nell'opera teatrale di fantascienza R.U.R. (*Rossum's Universal Robots*) in cui veniva usato per indicare delle creature antropomorfe artificiali (più simili a dei cloni che a dei robot) costruite con materiali organici sintetici con lo scopo di aiutare l'umanità a svolgere determinati compiti. Nel corso dell'opera teatrale però, come siamo spesso abituati a vedere sui grandi schermi, questi robot si ribellano portando all'estinzione della razza umana. Fatta questa premessa non c'è quindi da stupirsi che questo termine “robot” derivi da quello slavo “robota” che per l'appunto può essere tradotto con “lavori forzati”, “servitù” o più semplicemente “lavoro” [45].

Direttamente da “robot” deriva il vocabolo inglese “robotics” (poi tradotto in italiano con “robotica”) stante ad indicare quella branca della scienza e della tecnologia che si occupa dello studio, della progettazione e del controllo dei robot. Questo termine è invece comparso diverso tempo dopo, precisamente nel 1941 ad opera del famoso scrittore di fantascienza Isaac Asimov nelle cui invenzioni letterarie ricorrono spesso anche le note “tre leggi della robotica” che secondo l'autore dovrebbero essere rispettate da ciascun robot affinché la convivenza con l'uomo sia possibile [46]:

1. Un robot non può recar danno a un essere umano né può permettere che, a causa del proprio mancato intervento, un essere umano riceva danno.
2. Un robot deve obbedire agli ordini impartiti dagli esseri umani, purché tali ordini non contravvengano alla Prima Legge.
3. Un robot deve proteggere la propria esistenza, purché questa autodifesa non contrasti con la Prima o con la Seconda Legge.

Al giorno d'oggi tali leggi vengono spesso prese realmente in considerazione durante la pianificazione del comportamento desiderato dei robot [26].

A seguito di queste e delle tante altre opere simili concepite nel corso del '900 è evidente come le nozioni di robot e di robotica si siano diffuse fortemente nella cultura popolare.

Analizzando l'argomento da un punto di vista scientifico e tecnologico è possibile riscontrare che non esiste una definizione di “robot” chiara, concisa e comunemente accettata; in generale con questo termine è possibile riferirsi ad una grande varietà di macchine automatiche la cui unica caratteristica comune è quella di essere stata costruita dall'uomo con lo scopo di compiere certe attività (task) per conto suo. In questa definizione ricadono ad esempio i robot industriali adottati nelle catene di montaggio delle industrie manifatturiere, i robot di servizio usati quotidianamente in molte abitazioni per pulire il pavimento o per tagliare l'erba del giardino, fino ad arrivare ai sofisticati robot utilizzati nelle missioni spaziali per l'esplorazione planetaria.

Figura 1: Robot industriali *KUKA* al lavoro in una catena di montaggio.

Figura 2: *Roomba* – robot domestico aspirapolvere prodotto da *iRobot*.

Figura 3: *RM510* – robot domestico tosaerba prodotto da *Robomow*.

Solitamente i robot vengono sviluppati per aiutare o addirittura sostituire l'uomo in lavori relativamente semplici ma ripetitivi e noiosi, in cui spesso sono anche più efficienti di noi, oppure in compiti più complessi nei quali l'intervento umano non è possibile per via dell'entità del task, se ad esempio le capacità fisiche dell'uomo non sono adeguate, o per via delle condizioni ambientali di lavoro. Nel primo caso ricadono ad esempio i sistemi sfruttati nei processi produttivi automatizzati come le catene di montaggio. Nel secondo caso invece ricadono una serie di scenari, ancora poco diffusi, in cui i robot vengono adottati in ambienti avversi e pericolosi per l'uomo, ad esempio in presenza di sostanze tossiche, in presenza di radioattività, in assenza di ossigeno, in ambienti sottomarini, o addirittura nello spazio aperto o sulla superficie di pianeti esotici. A questa seconda situazione appartengono però anche scenari più quotidiani come quelli di robot utilizzati per aiutare i soccorritori a trovare superstiti e feriti in caso di disastri (terremoti, incidenti, attacchi terroristici, etc.) oppure quelli di robot in dotazione agli eserciti utilizzati in zone di guerra e di lotta al terrorismo per supportare la sorveglianza e l'esplorazione di territori ostili o per disinnescare ordigni esplosivi quali bombe e mine.

Figura 4: *PackBot* – robot militare telecomandato prodotto da *iRobot*.

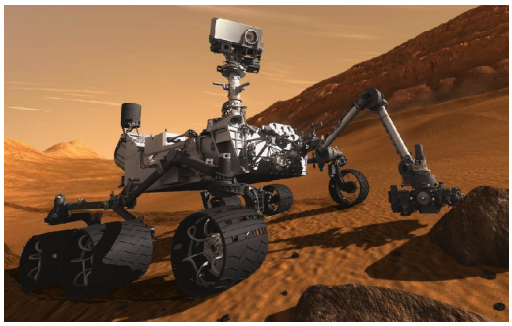


Figura 5: *Curiosity* – rover per l'esplorazione di Marte sviluppato dalla NASA.

Per quanto riguarda l'aspetto fisico vengono costruiti robot dalle forme più disparate: probabilmente quelle più conosciute sono le tipiche forme umanoidi raffigurate spesso nelle opere fantascientifiche ma in realtà la maggior parte dei robot sviluppati finora hanno sembianze molto più semplici che riprendono quelle di insetti ed animali oppure quelle di veri e propri veicoli, magari in scala ridotta, come gli UGV (*unmanned ground vehicle*), UAV (*unmanned aerial vehicle*) e UUV (*unmanned underwater vehicle*). In generale la specifica forma scelta durante la progettazione di un robot è direttamente influenzata dai compiti che questo dovrà svolgere e dalle caratteristiche dell'ambiente in cui si troverà ad operare.

Un importante connotato secondo il quale possono essere classificati i robot riguarda il livello di autonomia che questi devono dimostrare. Per autonomia si intende la capacità di funzionare in ambienti dinamici e non strutturati senza il bisogno di un continuo intervento umano [37]. In tali ambienti molte delle specifiche situazioni non sono note a priori obbligando il robot (o un qualsiasi sistema autonomo) ad essere in grado di rilevare le caratteristiche salienti della situazione corrente e a comportarsi di conseguenza decidendo quali azioni intraprendere. Inoltre la necessità di evitare l'intervento umano per lunghi periodi di tempo implica che si debba fare affidamento sulla piena capacità del robot di autogestirsi e di sopravvivere (ad esempio evitando di restare fisicamente bloccato oppure di esaurire completamente le riserve energetiche).

Figura 6: *ASIMO* – robot umanoide creato dalla *Honda.*

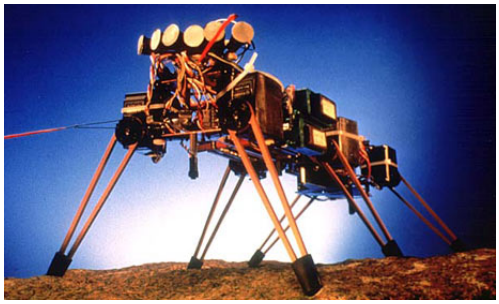


Figura 7: *Genghis* – robot creato al *Massachusetts Institute of Technology*.

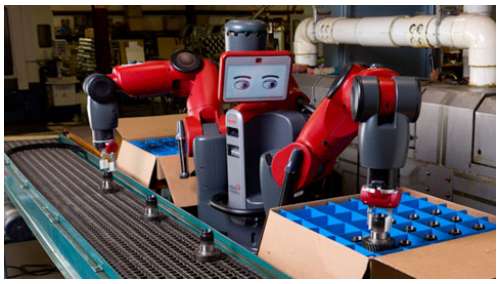


Figura 8: *Baxter* – robot manifatturiero multiuso prodotto da *Rethink Robotics*.

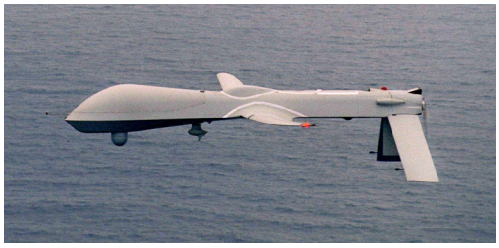


Figura 9: *MQ-1 Predator* – UAV creato da *General Atomics Aeronautical Systems*.

Figura 10: *MDARS* – UGV creato da *General Dynamics*.

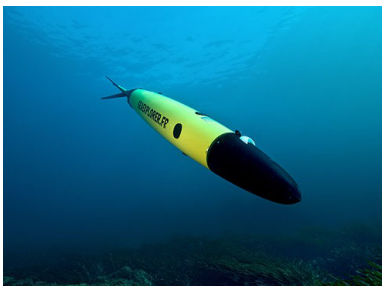


Figura 11: *SeaExplorer* – UUV creato da *ACSA*.

Se si considerano i robot industriali che eseguono compiti ripetitivi nel contesto di un processo produttivo è ovvio dedurre che macchine di tale fattura necessitano di poca autonomia. Infatti spesso queste macchine devo semplicemente eseguire una serie di azioni prefissate, arbitrariamente complesse, che viene definita una volta per tutte al momento della progettazione e soprattutto

della programmazione del robot. Di solito tali macchine svolgono i loro compiti all'interno di celle protette (*workcell*) appositamente modellate per facilitarne le operazioni, nelle quali, per motivi di sicurezza, è addirittura vietato l'accesso mentre queste sono in funzione [42]. Robot di questo tipo possono essere sviluppati dando per scontato molti aspetti riguardanti l'ambiente di lavoro e il più delle volte è sufficiente adottare i classici algoritmi di controllo automatico, al limite anche molto articolati, per fornire al robot la capacità di operare al meglio sfruttando le poche informazioni utili prelevate dall'ambiente tramite i consueti loop di feedback. In situazioni invece in cui l'ambiente non può essere modificato artificialmente, e non è quindi noto a priori, è evidente che il robot dovrà possedere una certa autonomia per poter fronteggiare le difficoltà e i pericoli che condizioni di questo tipo comportano.

Un tratto tipico dei sistemi autonomi è la mobilità. Un robot che per svolgere il task previsto ha la necessità di spostarsi fisicamente all'interno di un ambiente deve inevitabilmente incorporare una certa autonomia che gli permetta anche solo di spostarsi in maniera sicura evitando gli ostacoli ed evitando di costituire una minaccia per eventuali esseri viventi presenti nelle vicinanze.

Ovviamente si possono individuare diversi livelli di autonomia, da sistemi pienamente autonomi che non richiedono mai, o quasi mai, l'intervento umano[1] a sistemi telecomandati che possono essere più o meno autonomi a seconda che si affidino ai soli comandi che gli vengono impartiti o se parte delle operazioni vengono decise con indipendenza dal robot stesso.

Nel seguito si tratterà sostanzialmente solamente il caso di robot autonomi e tipicamente anche mobili.

La robotica è di per sé una scienza fortemente multidisciplinare che deve tenere in considerazione aspetti provenienti da diversi campi. Ad esempio la progettazione e la costruzione concreta di un robot coinvolge fondamenti di discipline quali la fisica, la meccanica, l'automazione, la teoria dei controlli;

1 Si considerino ad esempio i robot utilizzati nell'esplorazione spaziale, come quelli inviati su Marte, che una volta lanciati ricevono pochissimi comandi dall'uomo, anche per via del fatto che i ritardi di trasmissione iniziano ad essere molto poco trascurabili (fino a qualche decina di minuti in funzione della distanza corrente dalla Terra).

mentre la pianificazione del comportamento del robot sfrutta spesso concetti e tecniche inerenti all'informatica, all'intelligenza artificiale, alla biologia.

Tecnicamente parlando un robot può essere visto come un particolare tipo di controllo automatico [39], cioè un automa fisicamente situato in un ambiente del quale può percepire alcune caratteristiche tramite dei componenti denominati "sensori" e sul quale può compiere azioni con lo scopo di effettuarne delle modifiche per mezzo di dispositivi denominati "attuatori". Tutto ciò che è interposto tra le rilevazioni effettuate tramite i sensori e i comandi impartiti agli attuatori può essere definito come il "programma di controllo" o il "controllore" del robot. Questo è il componente in cui è codificata l'intelligenza del robot e, in un certo senso, ne costituisce quindi "il cervello" che ne deve guidare le azioni al fine di ottenere il comportamento desiderato. Un controllore può essere implementato in vari modi: in genere si tratta di software in esecuzione su di uno o più microcontrollori[2] fisicamente integrati nel sistema (on-board) però può anche essere ottenuto tramite circuiti elettronici (analogici o digitali) direttamente cablati nell'hardware del robot. La natura "astratta" del software lo rende ovviamente la soluzione più flessibile in quanto tale entità è di per sé facilmente trasferibile da un supporto di memorizzazione ad un altro: di conseguenza per cambiare il controllore, e con esso l'intero comportamento assunto dal robot, è sufficiente cambiarne il software di controllo mentre l'hardware resta inalterato permettendo dunque di effettuare modifiche a costo praticamente nullo.

Nel seguito verranno tralasciati i dettagli a proposito di sensori ed attuatori e ci si occuperà principalmente della parte riguardante i controllori ed in particolare si tratteranno solo sistemi in cui tali controllori sono costituiti da software. Due note possono essere fatte a riguardo. A seconda della particolare tecnologia software con cui il controllore viene sviluppato potrebbe esistere l'opportunità di tramutare automaticamente il programma di controllo in schemi circui-

2 Un microcontrollore è un componente elettronico completo che integra su di uno stesso chip un classico microprocessore ed una serie di dispositivi preconfigurati, quali una memoria volatile, una memoria permanente, interfacce di comunicazione di rete, porte di Input/Output programmabili e così via.

tali ottenendo un sistema certamente non più versatile ma decisamente più prestante: per di più in alcuni casi la soluzione hardware è addirittura l'unica soluzione perseguibile a fronte di vincoli temporali estremamente stretti che non siano affrontabili da un processore general-purpose. Infine si rende noto che a volte per superare i limiti computazionali di cui soffrono i microcontrollori si adottano soluzioni distribuite in cui il controllo viene suddiviso in due (o più) parti, comunicanti tra loro mediante una rete di telecomunicazione, una delle quali è effettivamente in esecuzione sul processore on-board del robot, mentre l'altra è affidata ad un supporto computazionale più potente [26].

1.2 Tipologie di architetture di controllo

Trattandosi di software, anche per quanto riguarda il programma di controllo di un robot è possibile applicare i principi classici dell'ingegneria del software. Primo fra tutti viene preso in considerazione quello riguardante la definizione dell'architettura software del controllore, cioè la struttura ed il paradigma di alto livello adottati in fase di analisi per inquadrare il problema del controllo, ed in fase di progetto per modellarne la soluzione. Nel caso specifico del software di controllo di un sistema robotico il termine comunemente adottato è "architettura di controllo".

L'architettura di controllo può essere definita come l'astrazione che fornisce i principi secondo i quali organizzare il sistema di controllo definendone la struttura e allo stesso tempo imponendo vincoli sul modo in cui il problema del controllo possa essere risolto [15][16].

Affinché un'architettura sia adatta allo sviluppo di un particolare sistema di controllo deve, per quanto possibile, rifletterne i requisiti e quindi le caratteristiche desiderate. Nel caso di controllori di sistemi autonomi le proprietà di cui tali architetture dovrebbero godere possono essere sintetizzate in [19]:

- Reattività all'ambiente – il robot dovrebbe essere reattivo a cambiamenti improvvisi nell'ambiente e dovrebbe essere in grado di prendere in considerazione eventi esterni con tempistiche compatibili con la corretta ed efficiente esecuzione del compito che sta eseguendo (limiti temporali *hard* o *soft real-time*).
- Comportamento intelligente – ciò richiede che il robot segua regole "di buon senso" in modo da manifestare comportamento intelligente e in modo che le reazioni agli stimoli esterni siano guidate dagli obiettivi del task.
- Integrazione di sensori multipli – i sensori presi singolarmente spesso soffrono di accuratezza, affidabilità ed applicabilità limitate, il che deve pertanto essere compensato con l'integrazione delle letture provenienti da più sensori complementari, possibilmente di natura diversa, che devono essere sfruttabili al meglio dal sistema di controllo (ridondanza dei sensori).

- Perseguimento di obiettivi multipli – il sistema di controllo dovrebbe essere in grado di ottenere o mantenere svariati obiettivi che in certe situazioni possono anche essere in contrasto tra loro.
- Robustezza – il robot dovrebbe avere la capacità di continuare a svolgere il proprio lavoro discretamente (ovviamente nel limite del possibile) anche a fronte di input imperfetti, eventi inattesi e malfunzionamenti improvvisi.
- Affidabilità – sarebbe opportuno che i compiti affidati al robot vengano portato avanti senza insuccessi e senza degradazione delle prestazioni.
- Programmabilità – caratteristica molto interessante riguarda la possibilità di sottoporre al robot dei task, descritti ad un certo livello di astrazione, anche a tempo di esecuzione invece che sviluppare sistemi di controllo capaci di gestire il solo task previsto in fase di progettazione.
- Modularità – il sistema di controllo di un robot dovrebbe essere suddiviso in sottosistemi (moduli, componenti) più semplici progettabili, sviluppabili e testabili separatamente, che siano poi integrati in maniera incrementale.
- Flessibilità – lo sviluppo di un sistema di controllo generalmente si affida a continue sperimentazioni i cui risultati vengono usati “per guidare” lo sviluppo e soprattutto la modifica della soluzione corrente, per cui tanto più un'architettura permette di creare controllori flessibili e facilmente alterabili, tanto più risulta possibile seguire questa strategia.
- Estensibilità – proprietà derivante dalla modularità e dalla flessibilità di un sistema che ne permette uno sviluppo incrementale e prolungato nel tempo attraverso l'integrazione, la verifica e il miglioramento dei suoi componenti.
- Scalabilità – il sistema dovrebbe poter scalare agilmente in relazione alle risorse computazionali disponibili, per esempio a fronte della sostituzione del processore con un modello più potente o della creazione di un'infrastruttura parallela composta da più microcontrollori interconnessi (in questo caso il controllo deve essere progettato come un sistema fisicamente distribuito).
- Adattabilità – dato che le caratteristiche dell'ambiente circostante possono modificarsi in maniera impredicibile, una caratteristica molto utile per un

sistema di controllo è l'abilità di adattarsi a questi cambiamenti e alterare di conseguenza le strategie di controllo (sono quindi auspicabili proprietà quali la capacità di apprendimento a tempo di esecuzione).

In genere ogni architettura si affida ad una determinata strategia di controllo: diverse strategie sono state sviluppate nel corso del tempo che spaziano da tecniche deliberative a tecniche puramente reattive. Queste sono considerate i due estremi che delimitano (grossolanamente) lo spazio concettuale all'interno del quale possono essere poi idealmente collocate le altre strategie. Spostandosi da un estremo all'altro cambia l'entità di alcuni tratti tipici dei sistemi di controllo come la dipendenza da una modellazione simbolica astratta dell'ambiente, le capacità cognitive manifestate, la quantità di informazioni elaborate ad ogni istante e di conseguenza il ritardo con cui il robot fornisce una risposta a fronte degli stimoli percepiti [1].

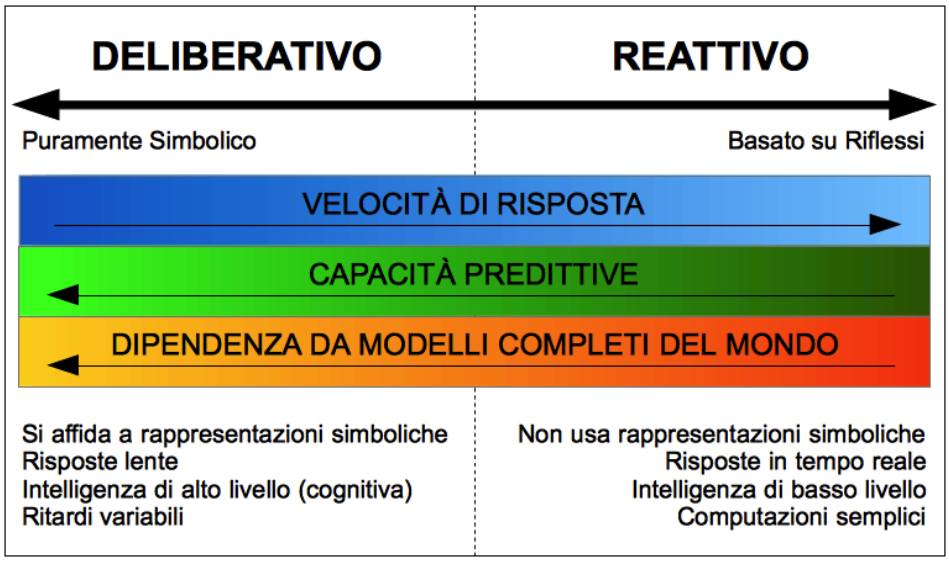


Figura 12: Spettro dei paradigmi di controllo.

In definitiva è possibile riscontrare il fatto che non esiste un'architettura migliore in assoluto, al contrario ciascuna di esse ha pro e contro. Di conseguenza la scelta sull'approccio da adottare dipende solitamente dal task che il robot dovrà svolgere e dalle condizioni di lavoro (l'ambiente): a volte potrebbe essere più agevole sviluppare il controllo come un semplice insieme di regole reattive, mentre altre volte può essere più comodo mantenere (aggiornato) un

modello interno del mondo invece che codificarlo direttamente in regole elementari [15].

1.2.1 Controllo deliberativo

A partire dalla nascita dell'intelligenza artificiale (ufficializzata con la *Dartmouth Summer Research Conference* nell'estate del 1956) è comunemente accettata l'idea che una macchina intelligente sviluppata per un certo scopo tenderebbe a costruire internamente un modello astratto dell'ambiente in cui si trova collocata sotto forma di rappresentazioni simboliche. Nel momento in cui vi venga sottoposto un problema da risolvere questa cercherebbe prima di tutto di esplorare le possibili soluzioni analizzando il modello interno e generando un piano da seguire composto da una sequenza di azioni; soltanto dopo questa fase di "deliberazione" tenterebbe di eseguire effettivamente il piano generato effettuando "esperimenti" sull'ambiente reale. Questo approccio dominò la ricerca nel campo dell'intelligenza artificiale per circa i trent'anni successivi. La robotica non ha fatto eccezioni, essendo in linea di principio strettamente legata a questa materia, e ciò portò allo sviluppo di strategie fortemente fondate sulla rappresentazione simbolica della conoscenza e sui metodi di pianificazione basati sul ragionamento deliberativo [1].

Un robot che impiega questo approccio richiede quindi un una conoscenza relativamente completa del mondo che lo circonda e usa questa conoscenza per prevedere gli effetti delle sue azioni, un'abilità che gli permette di ottimizzare le prestazioni del suo operato. Tutto ciò richiede forti assunzioni riguardo a tale conoscenza, prima fra tutte quella che il modello interno astratto analizzato in fase di pianificazione sia consistente, affidabile e sicuro. Se al contrario queste informazioni fossero inaccurate o se la loro controparte nel mondo reale fosse mutata dal momento in cui sono state reperite, l'esito della fase di esecuzione del piano potrebbe essere imprevedibile e terribilmente errato. In un ambiente fortemente dinamico, in cui ad esempio vi sono vari oggetti che si spostano arbitrariamente, è potenzialmente dannoso affidarsi ad informazioni obsolete e non più valide. La rappresentazione interna del mondo viene costruita in princi-

pio a partire dalla conoscenza a priori posseduta in fase di progettazione o di inizializzazione del sistema, ma, a seguito delle considerazioni appena fatte, perché costituisca un valido supporto alla deliberazione è evidente che deve soprattutto essere mantenuta costantemente aggiornata attraverso le informazioni provenienti dai sensori [1].

Questa idea fondante porta al tipico ciclo esecutivo *sense-model-plan-act* (SMPA) delle architetture che seguono l'approccio deliberativo. Tale ciclo è costituito da una fase in cui vengono effettuate le letture dei sensori (*sense*), una fase di modellazione in cui tali informazioni sono utilizzate per mantenere aggiornata la rappresentazione interna del mondo (*model*), una fase di pianificazione in cui il sistema di controllo "ragiona" sulla base del modello interno generando la sequenza di azioni da compiere (*plan*), ed infine una fase di esecuzione delle azioni previste (*act*).

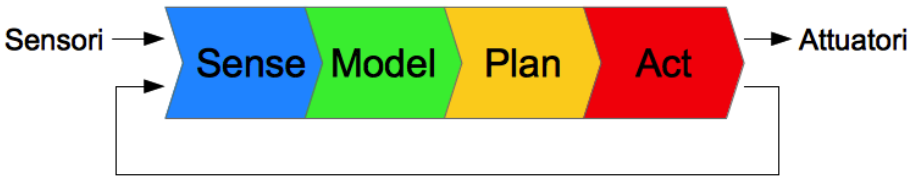


Figura 13: Ciclo di controllo *sense-model-plan-act*.

È quindi evidente che il paradigma SMPA si basa sull'assunzione che il mondo sia quasi-statico, o al limite predicibile, per tutto l'intervallo di tempo che intercorre tra le fasi di *sense* e *act*. Il fatto però è che nella maggior parte dei domini operativi questa assunzione non è valida (soprattuto all'aumentare della durata di tale lasso temporale).

Come conseguenza sono stati introdotti approcci leggermente più articolati per migliorare la reattività di questi sistemi mediante la decomposizione del singolo ciclo *sense-model-plan-act* in più cicli deliberativi paralleli ed interconnessi [19]. È infatti aspetto comune di queste architetture l'adozione di una suddivisione verticale gerarchica del sistema di controllo in livelli deliberativi orizzontali ciascuno dei quali segue un proprio ciclo *sense-model-plan-act*. L'idea alla base di questa suddivisione è quella che i diversi livelli della gerarchia elaborano piani che considerano scale spaziali e temporali diverse (quello

che viene spesso denominato *scope*) secondo un approccio top-down. In pratica l'assunzione di fondo è che le dinamiche di breve periodo dell'ambiente influiscano sempre meno nello svolgimento di un compito all'aumentare del livello di astrazione con cui lo si affronta [19]. Così i livelli inferiori tipicamente si preoccupano degli aspetti di basso livello (tra cui la gestione di sensori ed attuatori) con l'ausilio di modelli astratti molto ridotti a favore di una maggiore reattività. Mano a mano che si passa ai livelli superiori si eleva il livello di astrazione e si riscontrano modelli simbolici di alto livello e pianificatori che generano sequenze di azioni che si protraggono per lunghi periodi di tempo fornendo dei sotto-obiettivi ai livelli inferiori. È possibile delineare un flusso informativo che segue un percorso dal basso verso l'alto e poi dall'alto verso il basso attraversando tutta la gerarchia. Le percezioni generate dai sensori vengono catturate dai livelli inferiori, elaborate, e poi eventualmente trasmesse a quelli superiori. Lungo questo percorso dati provenienti da vari sensori possono essere aggregati in percezioni di più alto livello particolarmente adatte alle rappresentazioni richieste dalle pianificazioni eseguite ai livelli superiori. Nella fase di discesa invece questo flusso contiene i comandi e gli obiettivi parziali che i livelli superiori sottopongono a quelli subordinati. Essendo diverse le scale spaziali e temporali considerate dai vari livelli, sono diverse anche le tempistiche con cui i livelli eseguono i loro compiti: infatti i livelli inferiori, preoccupandosi degli aspetti "locali" (sia dal punto di vista spaziale che temporale), eseguono piani semplici e veloci con tempi di risposta brevi ma sono attivati più frequentemente in maniera da fornire continuamente una risposta consona alla situazione attuale; i livelli superiori invece si occupano di pianificazioni a lungo termine che si spera siano influenzate molto poco dalle variazioni locali dell'ambiente e quindi possono permettersi tempi di elaborazione più lunghi e soprattutto possono essere attivate molto meno di frequente[3] [1].

3 Si prenda ad esempio un caso in cui un robot debba raggiungere una destinazione a chilometri di distanza e durante il percorso incontri un ostacolo non previsto: i piani generati dai livelli inferiori dovranno quasi certamente essere rielaborati per permettere al robot di aggirare l'ostacolo mentre i piani di più alto livello possono restare inalterati in quanto la modifica indotta dalla presenza dell'ostacolo è solamente locale.

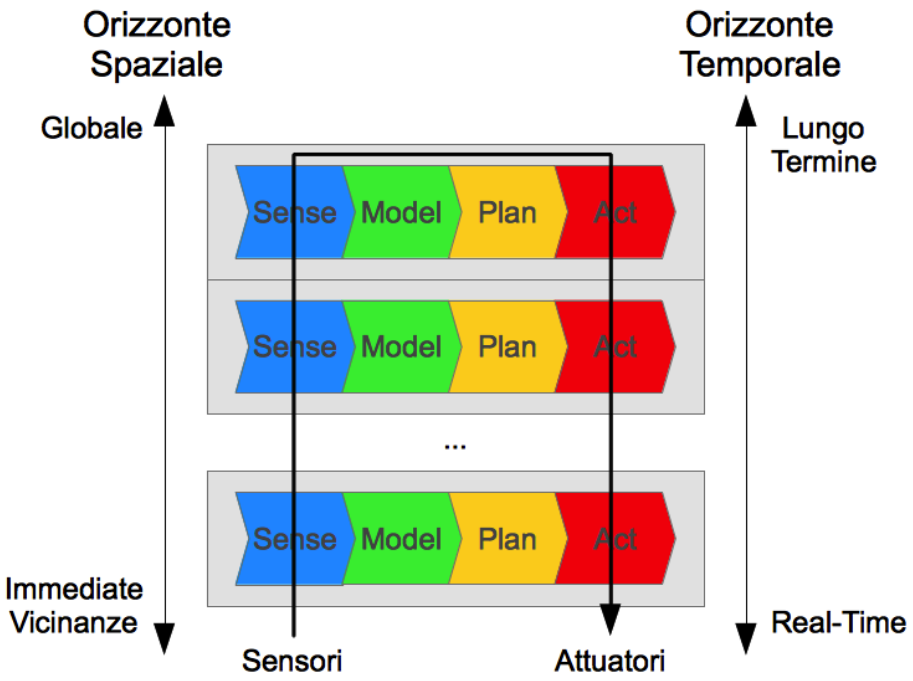


Figura 14: Architettura di controllo deliberativa gerarchica.

Un classico esempio di tali architetture è la NASREM *Telerobot Control System Architecture* sviluppata come modello di riferimento standard NASA/NIST(NBS) (*National Aeronautics and Space Administration / National Institute of Standards and Technology*, a quel tempo denominato *National Bureau of Standards*) voluto dal governo degli Stati Uniti d'America intorno alla metà degli anni 80.

L'ambito di applicazione per cui questi sistemi sono molto adatti riguarda tutti quegli ambienti relativamente prevedibili e strutturati in cui in ogni momento la fase di esecuzione, con buona probabilità, non incontrerà difficoltà né tantomeno fallimenti, mentre la fase di pianificazione può sensibilmente migliorare le prestazioni del robot [1].

In genere il modello interno utilizzato da queste architetture nella fase di pianificazione è mantenuto in una posizione in cui devono confluire le percezioni provenienti da tutti i sensori: non stupisce che la natura centralizzata di questo approccio venga spesso considerata un difetto [15][16]. Inoltre l'incertezza intrinseca di cui soffrono le fasi *sense* ed *act*, per via della naturale imprecisione di sensori ed attuatori, ed i continui cambiamenti dell'ambiente, spesso

richiedono frequenti ripianificazioni il cui costo diventa rapidamente proibitivo al crescere della complessità dei sistemi. Gli approcci deliberativi sono stati infatti criticati per scalare poco con la complessità dei problemi pratici rendendo impossibili reazioni in tempo reale ai cambiamenti improvvisi che avvengono nell'ambiente [16].

1.2.2 Controllo reattivo

In risposta ai problemi dimostrati dalle architetture deliberative classiche, in termini di scarsa capacità di risposta e scarse prestazioni in ambienti incerti, sono stati sviluppati degli approcci contrapposti denominati reattivi.

Il paradigma di controllo reattivo può essere definito semplicemente come una tecnica per accoppiare strettamente percezione ed azione per produrre risposte tempestive in ambienti dinamici e non strutturati [1]. L'appellativo "reattivo" si riferisce in genere sia alla prontezza di risposta di cui godono questi sistemi, sia alla caratteristica di evitare il più possibile il mantenimento di uno stato interno, entrambi aspetti peculiari di questo tipo di architetture [19]. Chiaramente evitare di appoggiarsi ad uno stato interno rimuove anche il vincolo di doverlo mantenere aggiornato e quindi aiuta ulteriormente a contenere i tempi di risposta. Ma quel che in realtà influisce per la maggiore su questa capacità è il fatto che la fase di pianificazione, che è quella più impegnativa in termini computazionali, viene direttamente evitata. Queste proprietà possono essere riscontrate nella loro manifestazione più forte nei sistemi cosiddetti "puramente reattivi" che sono solitamente costituiti da una collezione di coppie condizione-azione preprogrammate che sfruttano al limite uno stato interno di dimensioni minime.

Contrariamente a quanto avviene nei sistemi deliberativi non si adotta dunque un modello simbolico astratto del mondo e dato un obiettivo da raggiungere non viene effettuata alcun tipo di ricerca nello spazio delle possibili soluzioni. Al contrario, tutto si basa su di un mapping funzionale tra stimoli e risposte implementato solitamente come un mero "lookup"[4] delle azioni in base alle

4 Ricerca "istantanea", eseguita cioè con complessità temporale costante, effettuata

letture correnti dei sensori ottenendo un accoppiamento diretto tra sensori ed attuatori e un loop di feedback molto veloce [15][16].

Affinché un sistema puramente reattivo sia adatto ad un particolare contesto l'ambiente deve essere sufficientemente noto, il task deve essere ben definito e il robot deve avere a disposizione tutti i sensori necessari. Ciò è essenziale perché il progettista deve essere in grado di prevedere tutte le situazioni rilevamenti in cui il robot potrà ritrovarsi, dopo di che deve determinare e predisporre le risposte appropriate effettuando, per così dire, la fase di pianificazione in anticipo, cioè in fase di sviluppo [15]. Negli ambiti in cui questo è fattibile i sistemi reattivi ottengono alti livelli di efficienza dovuti soprattutto alla semplicità delle soluzioni e alle ridotte computazioni occorrenti; la loro applicabilità è però molto limitata in quanto l'incapacità di memorizzare ed elaborare dinamicamente una base di conoscenza ne impedisce notevolmente la flessibilità a tempo di esecuzione [16]. Inoltre il non utilizzo di un modello interno tende anche a generare comportamenti che seguono poco gli obiettivi desiderati in quanto il robot è per l'appunto incapace di "guardare avanti" e pianificare il corso delle azioni e può solamente reagire localmente agli stimoli che riceve: negli ambienti fortemente dinamici un sistema di questo tipo può essere adottato solamente per compiti di base quali assicurare la sopravvivenza del robot (ad esempio evitando gli ostacoli) [19].

1.2.3 Controllo behavior-based

Il paradigma di controllo *behavior-based* si interpone tra i due estremi, deliberativo e reattivo, riprendendo molti dei concetti introdotti con l'approccio reattivo ma cambiandone sostanzialmente la strategia[5]. Le architetture che seguono questo paradigma possono infatti essere considerate un'estensione di quelle

direttamente sulle strutture dati del programma di controllo: ad esempio data la situazione corrente si cerca all'interno di un elenco prefissato di regole quella più adatta da seguire.

5 Va fatto notare che i sistemi *behavior-based* vengono considerati da alcuni autori (tra cui Ronald Arkin [1]) come un caso particolare dei sistemi reattivi mentre i sostenitori principali di questo paradigma (a partire da Rodney Brooks e Maja Matarić [15][16]) asseriscono che esistono importanti differenze tra i due approcci. In quest'opera i due approcci vengono considerati differenti.

reattive: anche se condividono molti aspetti con le architetture reattive le loro capacità non si limitano alla semplice determinazione della prossima azione da eseguire ma possono invece sfruttare qualche forma di rappresentazione interna e sfruttarla per ottenere capacità e prestazioni superiori [15][16].

Questa tipologia di controllo prende ispirazione dalla biologia, principalmente dallo studio del comportamento esibito da semplici animali a partire da insetti e forme di vita ancora più elementari, fino ad arrivare a comprendere anche animali relativamente complessi. L'obiettivo è quello di tentare di riprodurre le "tecniche di controllo" adottate da queste creature, frutto più dell'evoluzione naturale, manifestata sotto forma di "istinto", che delle loro abilità deliberative, trattandosi evidentemente di esseri con capacità cognitive molto limitate [26][17]. Tant'è che spesso queste tecniche di controllo sono state addirittura impiegate in simulazioni per lo studio di sistemi biologici presenti in natura [17][18]. Tutto ciò affonda le proprie origini in una forte avversione al classico paradigma di controllo deliberativo che si è dimostrato poco reattivo e poco robusto per lo sviluppo di robot autonomi in grado di operare in ambienti fortemente dinamici, come quelli reali [3].

Le architetture *behavior-based* sono per loro natura distribuite: sono costituite infatti da una collezione di moduli operativi denominati *behavior* che computano parallelamente ed in assenza di un sistema di coordinazione centralizzato. Ogni *behavior*[6] può essere visto come un processo indipendente che persegue un certo obiettivo, il quale, in alcune situazioni, potrebbe anche essere contrastante con quelli associati agli altri *behavior*. Per svolgere il proprio compito ogni *behavior* ha accesso ai sensori e può generare comandi da sottoporre agli attuatori: in pratica un *behavior* è come se fosse un piccolo sistema di controllo a sé stante. Dal momento in cui *behavior* diversi possono richiedere azioni diverse per gli stessi attuatori sorge il problema della selezione dell'azione da compiere che deve essere risolto coordinando in qualche modo il funzionamento dei vari *behavior*. La necessità di permettere la convivenza di

6 Il termine *behavior* è traducibile in italiano con "comportamento", tuttavia nel seguito si preferisce mantenere il vocabolo anglosassone originale perché rappresenta un concetto più tecnico rispetto all'equivalente italiano.

una moltitudine di *bahavior* rende il problema della loro coordinazione una delle sfide progettuali più impegnative e più importanti nell'ambito dello sviluppo di questo tipo di sistemi [16].

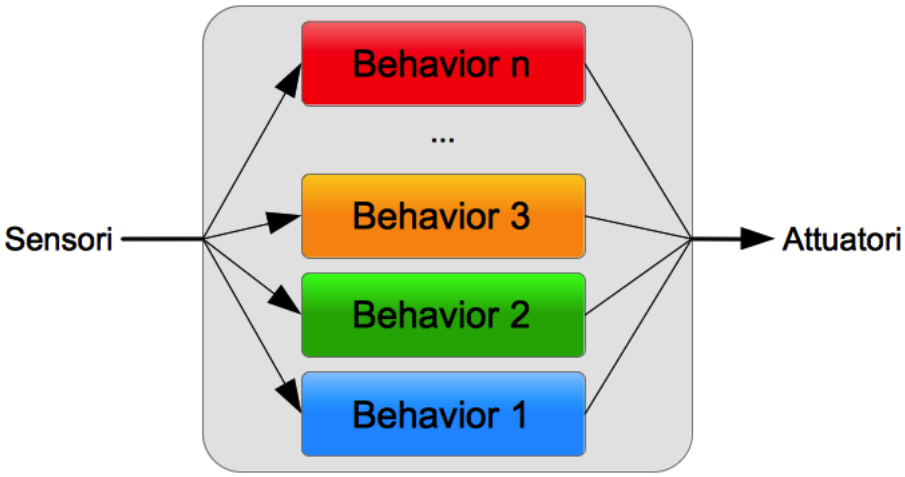


Figura 15: Architettura di controllo *behavior-based*.

La filosofia dei sistemi *behavior-based* richiede che l'esecuzione dei *behavior* non sia semplicemente serializzata ma che i *behavior* siano concettualmente concorrenti ed asincroni, con il conseguente vantaggio di dare vita a sistemi di controllo con la naturale propensione a sfruttare l'eventuale parallelismo offerto dall'hardware [16].

A volte certi sistemi *behavior-based* vengono progettati incrementalmente sviluppando una successione di comportamenti con livelli di astrazione crescenti (vedi l'architettura a sussunzione descritta più avanti) rassomigliando un po' ai sistemi gerarchici tipici delle architetture deliberative. In realtà anche in casi come questi c'è una forte discordanza tra i due approcci in quanto nell'approccio *behavior-based* ad ogni livello non viene comunque effettuata alcuna fase impegnativa di modellazione e pianificazione (ricalcando i sistemi reattivi) e soprattutto il flusso informativo è completamente diverso: nel caso deliberativo attraversa tutti i livelli passando da un livello a quello superiore/inferiore sotto forma di rappresentazioni simboliche astratte, nel caso *behavior-based* al contrario esistono più flussi paralleli che attraversano i singoli *behavior* separa-

tamente (anche se occasionalmente possono esistere scambi di informazioni *inter-behavior*).

I *behavior* sono entità ben più potenti delle semplici regole impiegate nel contesto di un sistema di controllo reattivo in quanto permettono di mantenere uno stato interno sulla base del quale effettuare semplici ragionamenti per determinare l'azione da compiere [15][16]. Inoltre l'utilizzo di informazioni di stato permette ai *behavior* di incorporare comportamenti estesi nel tempo differentemente da quanto possono fare delle banali regole reattive che normalmente si limitano a richiamare le azioni atomiche messe a disposizione dagli attuatori del robot. Nello stesso tempo però è buona norma che restino componenti possibilmente semplici. Tutti questi aspetti sono ciò che rende gli approcci *behavior-based* più espressivi e più potenti di quelli puramente reattivi, ma sono anche ciò che rende difficoltoso dare una definizione unica e precisa di "comportamento". Spesso infatti una nozione leggermente diversa di *behavior* può essere scelta di volta in volta che si affronta un nuovo problema. Di fatto le architetture *behavior-based* sono governate da un insieme di vincoli teorici che permettono una certa libertà di interpretazione; come conseguenza alcuni autori hanno criticato questa cosa interpretandola come una mancanza di struttura [18].

Come già detto di norma i sistemi *behavior-based* non utilizzano rappresentazioni centralizzate del mondo gestite da uno o più motori di reasoning al fine di effettuare della pianificazione; al contrario è possibile che si affidino a varie forme di rappresentazione distribuita sui *behavior*, sulle quali le elaborazioni che avvengono sono anch'esse distribuite. In alcuni casi le rappresentazioni non sono neanche strutture dati statiche ma processi attivi che influenzano direttamente il comportamento del robot e che possono essere avviati, fermati e modificati dinamicamente nel corso dell'esecuzione [18].

Spesso sistemi puramente reattivi composti da regole anche molto elementari producono comportamenti globali coerenti grazie all'interazione che avviene tra tali regole ed un particolare ambiente; in questo caso si parla comunemente di comportamenti "emergenti" in quanto non vengono specificati

esplicitamente in nessuna parte del sistema. Nei sistemi *behavior-based*, al contrario, questi comportamenti sono specificati internamente, anche se magari decomposti in varie parti; in questo caso le proprietà "emergenti" risultano dall'interazione tra i *behavior* e l'ambiente e costituiscono tipicamente comportamenti di più alto livello [16].

I *behavior* che compongono un sistema di controllo solitamente interagiscono sia direttamente, attraverso lo scambio interno di informazioni, sia indirettamente, per mezzo dell'ambiente. Le interazioni privilegiate sono però quelle del secondo tipo in cui i *behavior* interagiscono attraverso azioni e percezioni [16]: ad esempio due *behavior* possono interagire indirettamente dal momento che uno di essi esegue un'azione che modifica lo stato del mondo e l'altro rileva tale modifica a partire da successive percezioni reagendo in coerenza con il comportamento globale desiderato. È bene infatti ricordare che il comportamento complessivo esibito da un robot emerge dall'interazione dei tre elementi in gioco: il controllore che ne comanda gli attuatori, la sua struttura corporale che lo rende fisicamente situato nell'ambiente che lo circonda ed infine le proprietà e le dinamiche dell'ambiente stesso.

La tipologia di architettura *behavior-based* è quella più diffusa nell'ambito della robotica, soprattutto quando il problema da risolvere non richiede capacità deliberative particolarmente rilevanti, e costituisce l'argomento centrale di quest'opera; verrà pertanto approfondita nei capitoli successivi.

Alcune critiche sono state mosse nei confronti delle potenzialità di questi sistemi. Mentre i principali sostenitori di tale paradigma (tra cui Rodney Brooks, il pioniere e il più convinto fautore di questo approccio) affermano che con questo tipo di modello è "teoricamente" possibile riprodurre ogni forma di intelligenza, molti altri autori riconoscono la capacità di riprodurre comportamenti che richiedano competenze proprie di semplici animali, ma dubitano fortemente della capacità che questo ha di scalare al livello di intelligenze superiori quali quella umana, o comunque quella di animali particolarmente dotati [1]. Inoltre è stato criticato anche il fatto che è molto difficile riuscire a sviluppare correttamente i *behavior* fin dall'inizio e che spesso risulta necessario tor-

nare sui propri passi ed effettuare modifiche a *behavior* precedentemente sviluppati: nel caso in cui il sistema venga creato aggiungendovi progressivamente dei *behavior* che si appoggino in qualche modo a quelli già integrati nel sistema (vedi architettura a sussunzione descritta più avanti) una modifica di questo tipo ad un *behavior* primitivo richiederebbe di riassestare anche i successivi [19]. Infine molte delle architetture *behavior-based* non sono adatte allo sviluppo di sistemi "programmabili" il cui obiettivo venga deciso a tempo di esecuzione, ma sono in buona parte poco flessibili e capaci di ottemperare solamente agli obiettivi considerati durante la progettazione.

Come soluzione a questi problemi alcuni asseriscono che gli approcci reattivi e *behavior-based* possono in realtà essere compatibili con quelli deliberativi ed integrati con essi ottenendo architetture ibride (descritte nella prossima sezione) che possiedano contemporaneamente i vantaggi di entrambi.

1.2.4 Controllo ibrido

I sistemi reattivi e *behavior-based* non sono appropriati per tutte le applicazioni di robotica. In situazioni dove il mondo può essere modellato accuratamente e l'incertezza è limitata, i sistemi deliberativi sono preferibili siccome un piano completo può essere portato a completamento efficacemente. Nel mondo reale però le condizioni che favoriscono pianificatori puramente deliberativi sono molto rare. Inoltre i sistemi reattivi e *behavior-based* spesso si limitano a coprire requisiti di basso livello, come istinti di base per permettere al robot di sopravvivere e poco altro, mentre capacità di ragionamento e di deliberazione sono auspicabili al fine di essere in grado di risolvere problemi complessi. Molti pensano quindi che sistemi ibridi capaci di incorporare sia i vantaggi degli approcci deliberativi che quelli dei sistemi reattivi/*behavior-based* possano essere in grado di fornire le prestazioni migliori.

I sistemi deliberativi permettono di usare la rappresentazione della conoscenza per pianificare anticipatamente l'esecuzione delle azioni. Questa conoscenza può essere usata ad esempio per configurare o regolare la caratteristiche di uno strato di più basso livello sviluppato secondo l'approccio reattivo/*beha-*

vior-based in modo da perseguire determinati obiettivi e adattare meglio il comportamento reattivo all'ambiente di lavoro e agli scopi previsti [1]. Le architetture ibride si appoggiano dunque sui metodi deliberativi simbolici tradizionali dell'intelligenza artificiale e sul loro uso della rappresentazione astratta della conoscenza, ma mantengono contemporaneamente l'obiettivo di fornire la prontezza di risposta e la robustezza tipiche dei sistemi reattivi. Permettono di riconfigurare i sistemi di controllo reattivi sulla base della conoscenza del mondo disponibile attraverso la loro abilità di ragionare sui componenti reattivi sottostanti. La capacità di riconfigurare il sistema dinamicamente è un'aggiunta di competenza particolarmente significativa [1].

Questo tipo di approccio al problema del controllo è supportato anche da prove in campo biologico in cui, ad esempio, è noto che il comportamento umano può essere modellato come la coesistenza di due sistemi distinti, entrambi interessati al suo controllo. Un sistema modella il comportamento "automatico" (reattivo) e gestisce l'esecuzione di quelle azioni che vengono svolte senza attenzione né consapevolezza ed è costituito da una moltitudine di attività parallele (denominate *schemi*). Il secondo sistema invece controlla il comportamento "volontario" (deliberativo) che non avviene in automatico ma, contrariamente, richiede particolari attenzioni da parte dell'individuo e si interfaccia col precedente [1].

Nei sistemi reattivi e *behavior-based* la robustezza è ottenuta spesso a discapito di flessibilità ed adattabilità; azioni e percezioni vengono elaborate correttamente ed efficacemente ma le capacità cognitive sono spesso ignorate limitando i robot programmati secondo questi approcci ad imitare forme di vita molto primitive. In tanti sostengono che la rappresentazione della conoscenza sia necessaria per aumentare ed estendere i comportamenti di questi sistemi per risolvere problemi più complessi inerenti a domini più significativi. Questo include l'incorporazione di memoria e di rappresentazioni simboliche dinamiche dell'ambiente [1].

L'aspetto chiave su cui si differenziano le varie architetture ibride riguarda le strategie con cui la parte deliberativa e quella reattiva possono essere legate. Sono distinguibili tre approcci principali [1]:

- Integrazione gerarchica di pianificazione e reattività – la pianificazione deliberativa e l'esecuzione reattiva sono impegnate su attività diverse, su diverse scale temporali e spaziali (similmente a quanto avviene nei sistemi deliberativi gerarchici classici visti nella sezione 1.2.1).
- La pianificazione guida la reattività – alla pianificazione è permesso riconfigurare il livello reattivo, ad esempio modificandone dei parametri, all'interno del quale avviene effettivamente tutto il controllo.
- Accoppiamento pianificazione-reattività – pianificazione e reattività sono attività concorrenti che si guidano a vicenda.

In generale nelle architetture ibride sono necessari come minimo due livelli: uno deliberativo e uno reattivo. Sono comuni architetture che introducono esplicitamente un terzo livello intermedio che funge da interfaccia tra i due precedenti con lo scopo di coordinarli[7]. In sistemi di questo tipo i livelli sono posti in una gerarchia in cui il livello reattivo/*behavior-based* si trova nella posizione inferiore e si occupa dell'immediata sicurezza del robot e della transizione dal dominio della rappresentazione simbolica al dominio del controllo numerico non-simbolico, mentre il livello deliberativo si trova nella posizione superiore ed è costituito da un pianificatore che prende decisioni strategiche di alto livello e di lungo periodo affidandosi a conoscenza astratta [15][16][19]. Quando è esplicitamente presente anche un terzo livello interposto tra i due appena descritti questo ha lo scopo di riconciliare le diverse rappresentazioni adottate da quest'ultimi e risolvere eventuali conflitti, portando alle cosiddette architetture "a tre livelli" [18]: tale livello prende decisioni tattiche (ad esempio per mezzo di un pianificatore reattivo [43]) i cui effetti influiscono sul livello reattivo [19].

7 Un esempio relativamente semplice di sistema di questo tipo è mostrato in [9].

Per quanto concerne le modalità di progettare l'interfaccia tra la parte deliberativa e quella reattiva, quindi il livello intermedio, si possono individuare quattro tecniche principali [1]:

- Selezione – il pianificatore è adottato per determinare e configurare la composizione e i parametri di un livello inferiore sviluppato secondo l'approccio *behavior-based*. A questa categoria appartiene l'*Autonomous Robot Architecture* (AuRA), sviluppata da Ronald Arkin, che è di fatto una delle prime architetture ibride introdotte: questa architettura integra un pianificatore deliberativo gerarchico basato sulle tecniche tradizionali dell'intelligenza artificiale ed un controllore *behavior-based* flessibile e modulare basato sulla teoria degli schemi che viene inizializzato dal pianificatore ed eventualmente modificato durante l'esecuzione nel momento in cui si verificano situazioni anomale per le quali il solo sistema reattivo non è in grado di trovare soluzioni soddisfacenti.
- Consulenza – il pianificatore suggerisce cambiamenti che il sistema di controllo reattivo sottostante può effettuare o meno. A questa categoria appartiene l'architettura *Atlantis*, sviluppata al *Jet Propulsion Laboratory* (centro di ricerca e sviluppo della NASA), che integra un livello deliberativo che si occupa della pianificazione e della modellazione del mondo, un sequenzializzatore che gestisce l'inizializzazione e la terminazione delle attività di basso livello seguendo o meno i consigli forniti dal pianificatore, ed infine un controllore reattivo incaricato di eseguire collezioni di attività primitive.
- Adattamento – il pianificatore altera continuamente la componente reattiva alla luce dei cambiamenti delle condizioni del task che il robot sta eseguendo e dell'ambiente circostante. A questa categoria appartiene l'architettura *Planner-Reactor*, sviluppata da Lyons e Hendriks, composta da un livello *behavior-based* ed un livello deliberativo che ne monitora continuamente l'esecuzione. La componente reattiva è sviluppata usando il modello *Robot Schema* mentre il pianificatore è di tipo “anytime planning” il cui scopo è tentare di migliorare ininterrottamente la qualità di un piano che si

suppone essere una soluzione sub-ottima sempre disponibile ed in esecuzione da parte del livello reattivo.

- Rinvio – il pianificatore pospone le decisioni sulle effettive azioni da compiere fino a quando non è strettamente necessario; questo permette anche di utilizzare i valori più recenti provenienti dai sensori nel corso dell'esecuzione delle azioni diversamente da quanto sarebbe accaduto se si fosse creato subito un piano completo (come avviene negli approcci deliberativi classici). A questa categoria appartiene il *Procedural Reasoning System*, sviluppato da Georgeff e Lansky, che fornisce una strategia alternativa in cui il concetto di reattività può essere ricondotto al fatto che l'elaborazione dei piani (delle azioni da svolgere) viene ritardata il più possibile: quest'architettura si basa sul modello *belief-desire-intention*[8] e prevede che più piani possano essere in esecuzione concorrentemente al fine di perseguire i desideri del robot e che questi possano essere modificati, sospesi o abbandonati in qualsiasi momento a fronte di nuove percezioni che denotino cambiamenti significativi dell'ambiente. Tali piani non sono l'output di un pianificatore classico come avviene nei sistemi puramente deliberativi ma sono preconfigurati dal progettista e codificano i comportamenti desiderati; per questo motivo vi si fa spesso riferimento col nome di "conoscenza procedurale".

8 *Belief-desire-intention* è un modello usato per lo sviluppo di agenti intelligenti basato sul concetto di *belief* (credenza), che rappresenta lo stato informativo dell'agente, *desire* (desiderio), che rappresenta gli obiettivi che l'agente è interessato a perseguire, *intention* (intenzione), che rappresenta ciò che l'agente ha deciso di fare in un determinato momento e determina l'esecuzione di un piano [38].

1.3 Valutare sistemi di controllo tramite simulazione

La simulazione ricopre un ruolo molto significativo nel processo di sviluppo di un robot autonomo. Tramite simulazione è infatti possibile metterne alla prova la struttura fisica e soprattutto il sistema di controllo al fine di sperimentare varie alternative ed individuare l'architettura più indicata per il caso in esame.

Spesso la costruzione fisica di un robot può essere molto dispendiosa quindi è certamente vantaggioso poterne testare le capacità in simulazione prima di intraprendere investimenti importanti. Nello stesso tempo è anche possibile valutare diversi progetti del robot, sia della sua forma fisica che del software di controllo, a costo praticamente nullo ed è possibile evitare che comportamenti erronei e catastrofici del robot rechino danni "reali" a sé stesso o a chi o cosa vi sia vicino. La possibilità di testare in simulazione un programma di controllo porta anche a ridurre notevolmente i tempi di sviluppo in quanto si possono evitare operazioni, quali il trasferimento del software sull'hardware del robot e soprattuto la manipolazione fisica del robot, che possono prolungare le fasi sperimentali. In genere i simulatori permettono di simulare lo trascorrere di un "tempo virtuale" non per forza equivalente al "tempo reale", anzi, è solitamente possibile simulare la dinamica dell'ambiente e del robot ad una velocità molto superiore a quella reale riproducendo, ad esempio, una simulazione di qualche ora in pochi minuti (dipendentemente dalle capacità computazionali della macchina su cui viene effettuata la simulazione). Questa caratteristica è essenziale nel caso in cui in fase di sviluppo si vogliano adottare particolari algoritmi di intelligenza artificiale, ad esempio algoritmi euristici evolutivi [41], che richiedano di eseguire diverse prove al fine di ottimizzare certi parametri del sistema di controllo o addirittura di sintetizzare parti del sistema di controllo stesso. In situazioni come queste affidarsi a simulazioni eseguite a velocità anche molto superiori a quella reale è praticamente un'esigenza. Spesso infatti, affinché questi algoritmi generino i risultati sperati, sono necessarie anche molte decine di prove che, compiute con il robot reale e con le tempistiche reali, richiederebbe un lavoro molto tedioso e certamente molto impegnativo in termini di tempo. Infine c'è da ricordare quanto detto nelle sezioni

introduttive, e cioè che alcuni robot dovranno operare in ambienti in cui gli essere umani non possono introdursi o che non sono facilmente riproducibili "in laboratorio" per i quali la simulazione è evidentemente la via più praticabile se non l'unica[9].

La simulazione di un singolo robot di norma è costituita da un processo ciclico in cui ad ogni iterazione si considera essere trascorso un certo intervallo di tempo piccolo a piacere (solitamente dell'ordine di qualche decina di millisecondi) [26]: il simulatore si preoccupa di stimare i valori rilevati dai sensori, trasmetterli al programma di controllo, recuperare i comandi che questo ha inviato agli attuatori, determinare gli effetti che questi attuatori hanno sull'ambiente simulato, ed infine simulare l'applicazione delle forze in gioco agli oggetti presenti nell'ambiente, robot compreso. Tutto questo viene ripetuto finché non si verifica una certa condizione di terminazione, come una richiesta esplicita dell'utente del simulatore oppure un timeout.

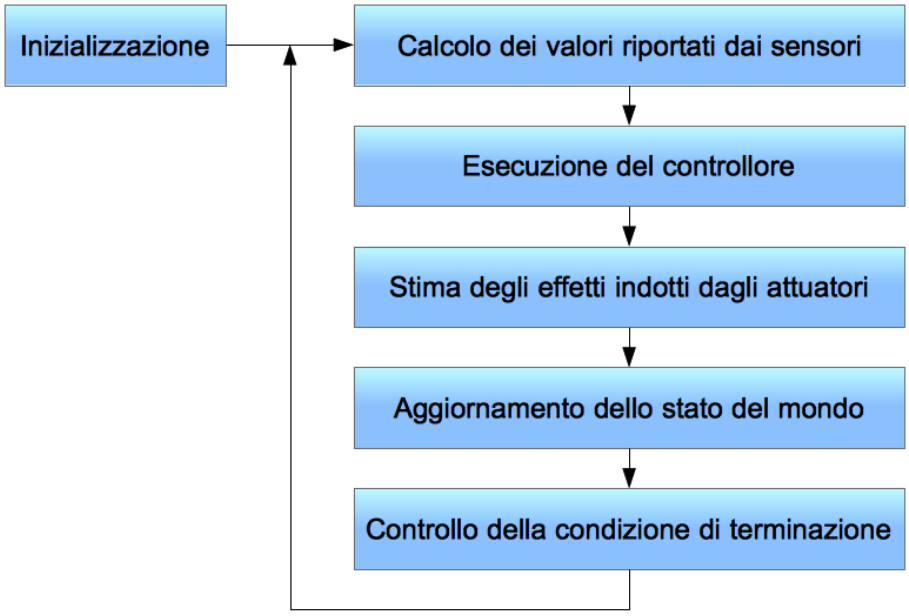


Figura 16: Tipico flusso esecutivo di una simulazione.

Le simulazioni di base prevedono normalmente di riprodurre gli effetti dovuti alla dinamica dei corpi rigidi e le forze applicate agli oggetti vengono

9 Si pensi ad esempio che con un simulatore si possono implementare leggi fisiche con parametri diversi da quelli presenti sulla Terra o, nel caso estremo, si possono perfino emulare leggi fisiche completamente diverse da quelle reali.

considerate costanti per tutto l'intervallo temporale di una iterazione: in funzione di queste forze il simulatore calcola le accelerazioni subite dagli oggetti e ne effettua lo spostamento conseguente. È evidente che una simulazione risulta tanto più accurata ed affidabile quanto minore è questo intervallo.

Perché i risultati di una simulazione siano coerenti con la realtà vanno fatte importanti considerazioni riguardo alle tempistiche delle fasi di lettura dei sensori, elaborazione da parte del sistema di controllo, ed azione degli attuatori. Nello specifico è necessario assicurarsi questi passi siano effettivamente eseguibili sul robot reale entro un intervallo di tempo compatibile (non superiore) a quello adottato per la simulazione fisica. A proposito di questo aspetto è possibile distinguere due tipi di eventi: eventi che per essere valutati richiedo più tempo nella simulazione ma meno tempo nel mondo reale, ed eventi che invece richiedono meno tempo nella simulazione ma più tempo nel caso reale [26]. Un esempio di eventi del primo tipo è la rilevazione delle collisioni che nel caso reale avviene molto velocemente e molto semplicemente attraverso la lettura di un sensore (come un sensore ad ultrasuoni o un sensore ad infrarossi attivi) mentre nella simulazione può richiedere anche diverso tempo in funzione della complessità della scena: eventi di questo tipo non alterano la validità della simulazione ma semplicemente ne rallentano l'esecuzione che nel caso estremo può diventare più lenta del tempo reale. Un esempio di eventi del secondo tipo è invece l'aggiornamento dei sensori che nella simulazione può essere effettuato con tempi anche molto minori rispetto a quanto avviene nel caso reale in cui le frequenze di aggiornamento dei sensori risentono di limiti inferiori invalicabili: in situazioni di questo genere è quindi necessario prestare attenzione al fatto che il simulatore rispetti i tempi previsti dai dispositivi reali per non rischiare di sviluppare un sistema di controllo che, una volta trasferito sul robot fisico, non ottenga le prestazioni attese. Infine stessa cosa vale per la parte di elaborazione del sistema di controllo che deve rispettare le tempistiche richieste anche nel caso del robot reale e quindi bisognerà fare delle considerazioni sulla potenza computazionale effettivamente disponibile sul robot.

Un aspetto importantissimo da tenere a mente quando si utilizza un simulatore è il fatto che il mondo reale, e con esso ciò che lo compone, è per sua natura complesso, imperfetto e non deterministico, impossibile quindi da modellare precisamente con semplici formule matematiche. Come conseguenza anche sensori ed attuatori non possono essere modellati e simulati con semplici formule matematiche. Infatti è sempre presente una componente di "rumore" che altera leggermente il valore di una lettura o l'esito di un'azione. Un simulatore deve quindi avere anche il compito di imitare questo rumore [26]. In genere vengono usate tecniche molto semplici come l'aggiunta di un valore casuale, alla lettura di un sensore o all'azione di un attuatore, generato con una certa distribuzione di probabilità (ad esempio una distribuzione uniforme o normale). Nel caso dei sensori è possibile anche usare una tabella di "lookup" (di ricerca) all'interno della quale viene inserito un insieme di letture effettuate con un sensore reale, dopo di che il simulatore si preoccupa di effettuare una interpolazione tra queste letture al fine di ottenere un valore plausibile per la situazione corrente della simulazione.

In generale è buona norma non affidarsi ciecamente alla simulazione ma farne un utilizzo moderato preferibilmente nelle sole prime fasi di sviluppo o per effettuare semplici test. La simulazione infatti è considerata avere certi tratti in comune agli approcci di controllo deliberativi in cui il robot agisce su di una base di conoscenza simbolica "virtuale" che astrae dalla situazione reale. Ugualmente un simulatore costituisce di sua natura un'astrazione della realtà (alcuni la definiscono "un'allucinazione artificiosa") che spinge un sistema di controllo a credere di agire in un ambiente reale mentre in realtà sta solamente elaborando dei simboli che non hanno alcuna correlazione ad eventi reali [1]. Pertanto è di vitale importanza dare priorità alle prove effettuate con robot reali in ambienti reali in quanto questo spesso rende note situazioni non contemplate dal sistema di controllo sviluppato basandosi esclusivamente sul comportamento del robot in simulazione, e potrebbe mettere alla luce problematiche che un simulatore non può essere in grado di catturare.

1.3.1 Webots

Il simulatore utilizzato nel seguito di quest'opera è *Webots*, uno strumento professionale sviluppato da *Cyberbotics* in collaborazione con lo *Swiss Federal Institute of Technology* (EPFL) di Losanna [28]. *Webots* è usato in più di mille università e centri di ricerche di tutto il mondo e viene promosso come una tecnologia collaudata, approfonditamente testata, ben documentata e in sviluppo continuo da quindici anni a questa parte con l'obiettivo di ridurre notevolmente i tempi di progettazione, implementazione e testing, a vantaggio di un processo di sviluppo agile. Si tratta di un ambiente di sviluppo completo creato con lo scopo di guidare la modellazione, la programmazione e la simulazione di robot principalmente mobili ed autonomi. Con *Webots* è possibile progettare scenari arbitrariamente complessi che includano eventualmente anche più di un solo robot. Offre inoltre la possibilità di riprodurre simulazioni di elevata qualità, con un aspetto grafico accattivante, nonché di scattare istantanee della situazione corrente o registrare video durante l'esecuzione di una simulazione.

Webots si appoggia sul concetto di "tempo virtuale" (introdotto poco fa) ed è quindi possibile riprodurre una simulazione a velocità molto maggiore rispetto a quella reale fino ad arrivare, nel caso limite, a sfruttare completamente le capacità dell'hardware su cui è in esecuzione. L'intervallo temporale elementare usato nel processo di simulazione prende il nome di *simulation step* e può essere configurato a piacimento a seconda che si preferisca ottenere una maggiore precisione od una maggiore velocità di completamento della simulazione[10].

Per quanto riguarda la simulazione delle leggi fisiche e delle interazioni tra gli oggetti presenti in scena *Webots* si affida al motore fisico *Open Dynamics Engine*. Questo motore sposa il modello dello *scene graph* (o *scene tree*) [47] secondo il quale ogni entità coinvolta nella simulazione, robot compreso, è rappresentata come un nodo di una struttura a grafo (o ad albero) ed è in una qualche relazione con l'entità padre. Questo approccio dà vita ad una struttura ricorsiva che permette ad esempio di creare robot liberamente complessi innestando

10 Viene consigliato un valore di default di 16 millisecondi.

tra loro diversi attuatori e fornendo così al robot molti gradi di libertà: si pensi ad esempio ad un braccio meccanico composto da più parti connesse da giunzioni regolabili. Un'ampia tipologia di oggetti possono comporre la scena da simulare comprendendo robot, sorgenti luminose di vario tipo, pavimenti, muri, ostacoli, etc. Le proprietà di ciascun oggetto, come forma, aspetto (colore e texture), massa e coefficienti d'attrito, sono configurabili individualmente e vengono poi sfruttate durante la simulazione per riprodurre le caratteriste desiderate dei corrispettivi oggetti reali. Se il funzionamento standard del simulatore fisico non dovesse essere sufficiente o se fosse necessario introdurre particolari leggi fisiche (ad esempio l'attrito viscoso all'interno di fluidi) è possibile estenderlo per mezzo di plug-in programmabili dallo sviluppatore.

Webots dispone di molti modelli di robot preconfezionati ma permette di progettare nuovi robot liberamente componendo tra loro oggetti elementari. Un robot può essere equipaggiato con una vasta gamma di sensori ed attuatori che possono essere programmati a piacimento. Tra i sensori sono presenti sensori di prossimità (ultrasuoni, infrarossi attivi, laser), sensori di tatto, sensori di luminosità, sensori di posizione e di forza sugli attuatori, telecamere, accelerometri, giroscopi, localizzatoti GPS, bussole digitali, ricevitori radio (per la comunicazione inter-robot), e altri. Tra gli attuatori invece sono disponibili servomotori rotazionali o lineari[11] (che possono essere usati per svariati scopi come mettere in rotazione delle ruote o regolare delle articolazioni meccaniche), LED, display LCD, emettitori radio (per la comunicazione inter-robot), e altri. Infine un particolare tipo di attuatore disponibile è la base con due (o più) motori per ruote differenziali che permette di costruire robot a guida differenziale[12].

11 Un servomotore è essenzialmente un motore elettrico a corrente continua equipaggiato con alcuni sensori, tra cui sensori di posizione e sensori di forza (*force feedback*), ed un circuito elettronico di controllo: generalmente può essere comandato richiedendo che l'attuatore raggiunga una determinata posizione (angolare o lineare) rispettando una certa velocità ed accelerazione massima [26][29].

12 Un robot a guida differenziale è un robot che possiede due (o più) ruote collegate a motori differenti che possono essere quindi pilotate indipendentemente le une dalle altre: il controllore del robot deve perciò preoccuparsi di impostare la velocità desiderata per ogni singola ruota affinché l'effetto complessivo indotto dalle diverse rotazioni produca lo spostamento desiderato del robot [26].

Il controllore del robot può essere scritto in uno dei diversi linguaggi disponibili: le API per eccellenza fornite da *Webots* sono quelle pensate per l'utilizzo del linguaggio *C* ma, sulla base di queste, sono poi state fornite API per *C++*, *Java*, *Python*, *Matlab*, ed infine è possibile accedere alle funzionalità del robot anche attraverso un'interfaccia TCP/IP standard. Il controllore può essere sviluppato usando l'ambiente di sviluppo integrato in *Webots* oppure un qualsiasi altro strumento di terze parti.

Infine l'ultimo support che *Webots* offre riguarda la possibilità di trasferire il controllore direttamente sul robot reale concludendo così il processo di sviluppo. Questa operazione è disponibile per un set di robot commerciali (*e-puck*, *Nao*, *DARwIn-OP*, e altri) mentre per un robot progettato dall'utente l'operazione può essere effettuata manualmente seguendo un guida.

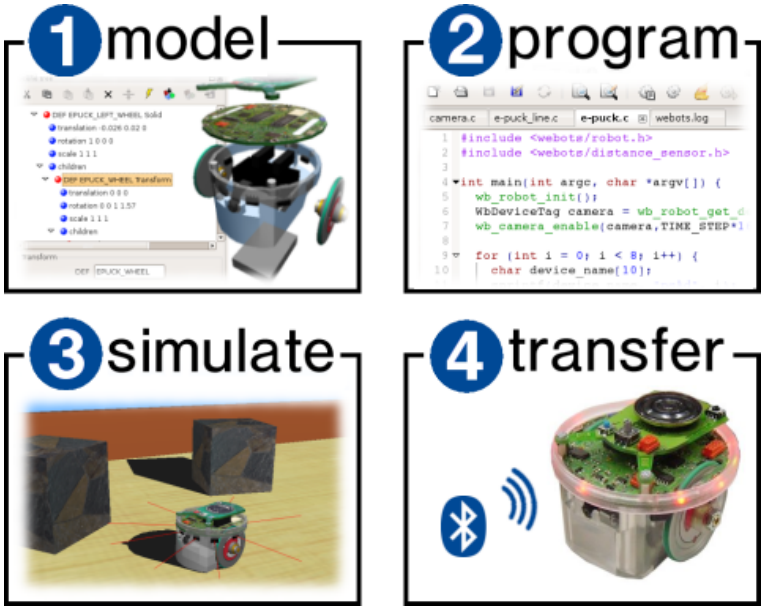


Figura 17: Le quattro fasi di sviluppo secondo la filosofia di *Webots*: modellazione, programmazione, simulazione e trasferimento opzionale sul robot reale.

L'idea di controllore che *Webots* prevede è quella di un processo ciclico in cui ad ogni iterazione vengono prelevati gli input forniti dai sensori, i quali sono poi elaborati generando gli output diretti agli attuatori. Per quanto riguarda la simulazione quello che avviene è che controllore e simulatore sono

eseguiti come due processi[13] indipendenti che comunicano tra loro. Ad ogni iterazione il controllore deve richiedere "attivamente"[14] l'intervento di *Webots* al fine di effettuare la simulazione di un intervallo di tempo, denominato *control step*, che, in pratica, rappresenta la durata temporale di una iterazione del controllore[15] [30]. Nel corso di questa operazione *Webots* metterà effettivamente in pratica il controllo aggiornando lo stato degli attuatori (oltre a quello del resto del mondo) e valutandone le interazioni con gli altri oggetti. La durata effettiva di questa operazione dipende dalla velocità a cui *Webots* sta effettuando la simulazione: può essere molto maggiore, (circa) uguale o molto minore al *control step*, a seconda che la simulazione dell'intervallo richiesto sia particolarmente onerosa, oppure, se ciò non si verifica, che la simulazione sia eseguita in tempo reale o in modalità accelerata. Al termine di questa operazione viene aggiornato lo stato della simulazione e viene trasferito al processo che esegue il controllo il valore aggiornato assunto dai sensori.

Perché la simulazione sia coerente con la situazione reale la parte di controllo che si interpone tra due richieste di simulazione successive, una volta trasportata sul robot reale, deve essere computabile in un tempo non superiore al *control step* usato per le richieste di simulazione.

Prima che il controllore possa iniziare a svolgere il proprio lavoro è necessario recuperare i riferimenti ai dispositivi utilizzati, cioè sensori ed attuatori, sulla base del nome simbolico che gli è stato assegnato. I sensori inoltre devono anche essere espressamente attivati selezionando il periodo di aggiornamento desiderato così da poter simulare sensori con tempi di ritardo differenti dagli altri. Infine per ogni sensore è disponibile un'operazione con cui recuperare il valore più recente calcolato durante l'ultimo passo di simulazione richiesto a *Webots*, mentre per ogni attuatore sono disponibili delle operazioni per assegnarvi comandi che verranno eseguiti a partire dal prossimo passo di

13 In questo caso con "processo" ci si riferisce all'accezione tipica dei sistemi operativi, cioè all'esecuzione di una istanza di una particolare applicazione.

14 Mediante la funzione `wb_robot_step` delle API *C* oppure il metodo `step` della classe `Robot` per le API *C++*, *Java* e *Python*.

15 L'intervallo *control step* deve essere un multiplo intero dell'intervallo elementare di simulazione *simulation step*.

simulazione richiesto. Presupposto questo funzionamento ne deriva che due letture successive di uno stesso sensore che non siano separate da una richiesta di simulazione restituiranno esattamente lo stesso valore; dualmente, se due comandi vengono sottoposti ad uno stesso attuatore prima che venga richiesta l'esecuzione di un nuovo passo di simulazione verrà considerato solamente il secondo.

Le potenzialità di *Webots* non si fermano qui, c'è infatti un'ulteriore entità particolarmente importante che prende il nome di *supervisor* (supervisore). Un *supervisor* in *Webots* è un elemento che è visto dal simulatore come un particolare tipo di robot e che ricopre per l'appunto il ruolo di "supervisore" della simulazione. In genere non è un robot fisicamente presente nella scena ma è solamente un processo astratto che ha la possibilità di intervenire nella simulazione proprio come potrebbe fare un supervisore umano nel corso di una prova reale. Questo tipo di processo ha infatti accesso a delle API più estese di quelle a cui hanno accesso i controllori dei normali robot. In queste API sono disponibili operazioni per ottenere informazioni riguardanti qualsiasi oggetto posto in scena e per modificare tali oggetti in maniera automatizzata. Un supervisore può compiere operazioni quali spostare istantaneamente un robot, o un qualsiasi altro oggetto, in una posizione qualsiasi, oppure fermare e riavviare la simulazione, catturare istantanee, registrare video della simulazione, e così via. Il compito principale affidato ad un supervisore è quindi quello di gestire la simulazione e di elaborarne particolari informazioni a vantaggio dell'intero processo di sviluppo: spesso infatti un supervisore viene adottato per recuperare statistiche sulle prestazioni del controllore da utilizzare poi in algoritmi di ottimizzazione. Ad esempio, nel caso di algoritmi evolutivi [41] un supervisore potrebbe avere il compito di generare automaticamente i controllori iniziali (trasmettendone le informazioni ai robot simulati), valutarne statisticamente le prestazioni in termini di fitness, e proseguire con l'iterazione successiva ripristinando lo stato iniziale della simulazione e raffinando la ricerca di una soluzione pseudo-ottima nei dintorni di quelle che all'iterazione corrente sono risultate migliori delle altre.

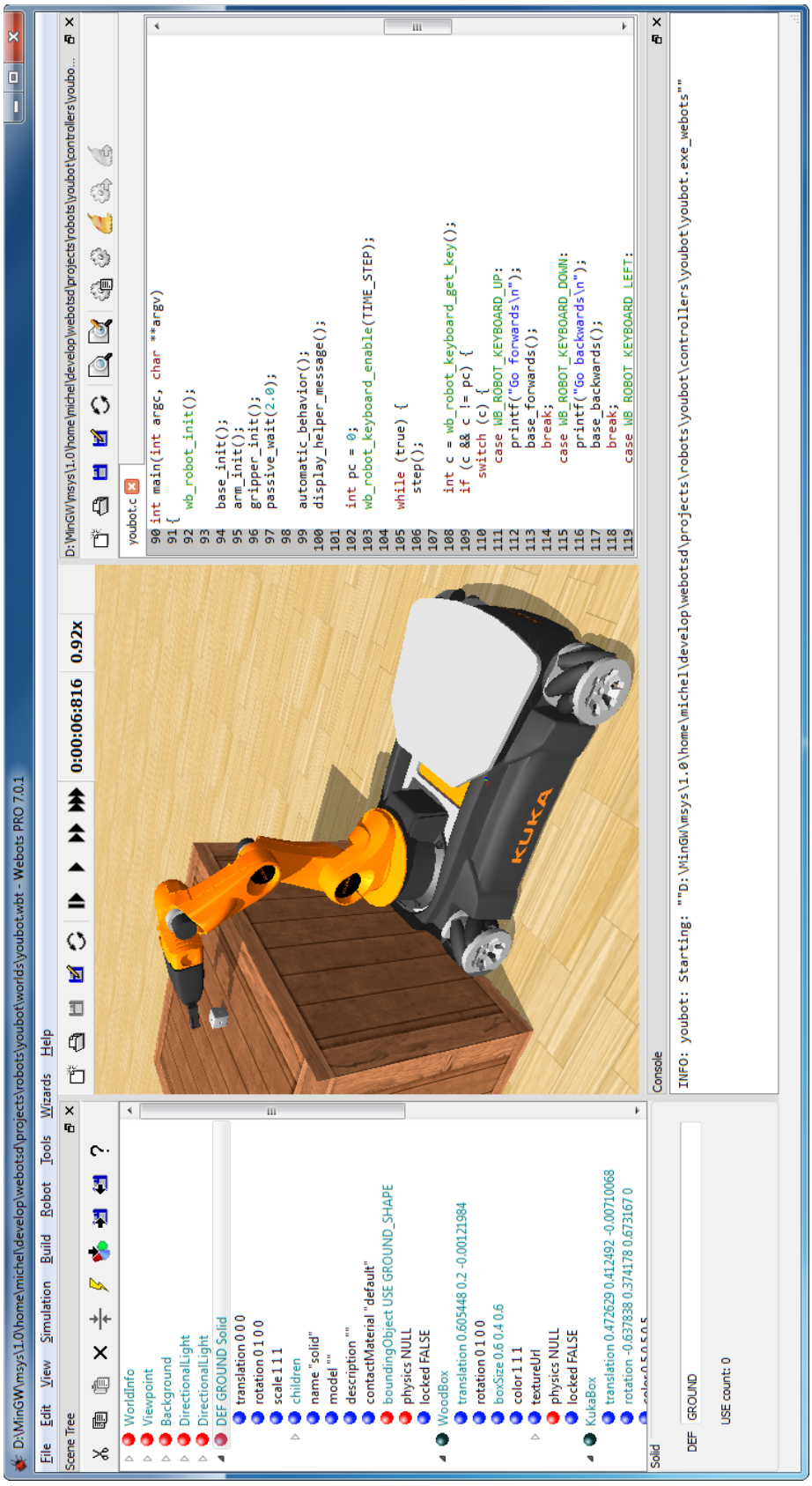


Figura 18: Interfaccia grafica di *Webots*: si notano lo *scene tree* a sinistra, la vista 3D al centro, l'IDE a destra e la console per lo standard output in basso.

Capitolo 2
Robotica comportamentale

La robotica comportamentale (dall'inglese *behavior-based robotics*) è quel particolare ramo della robotica che si occupa dello studio dei sistemi di controllo progettati secondo i principi del paradigma *behavior-based* introdotto nel capitolo precedente (sezione 1.2.3). La robotica comportamentale mette a contatto i campi dell'intelligenza artificiale, dell'ingegneria e delle scienze cognitive [18]. Lo scopo di questa disciplina è duplice: da un lato si concentra sullo sviluppo di metodi per controllare sistemi artificiali autonomi, solitamente robot, dall'altro si interessa di esplorare l'utilizzo della robotica per modellare, studiare ed aiutare la comprensione dei sistemi biologici, spaziando da semplici insetti fino ad animali più complessi (arrivando all'uomo nel caso limite) [17]. Come visto in precedenza la metodologia *behavior-based* impone una filosofia ispirata alla biologia ed intrinsecamente distribuita, che promuove un processo di sviluppo bottom-up e permette una certa libertà interpretativa. Lo studio portato avanti dalla robotica comportamentale trae le sue origini dall'architettura a sussunzione (descritta più avanti) e dal controllo reattivo.

Come si è visto nella sezione 1.2.3 un controllore *behavior-based* consiste in una collezione di moduli, denominati *behavior*, che hanno lo scopo di raggiungere o mantenere un certo obiettivo. Similmente a quanto accade per i sistemi puramente reattivi, progettare i moduli di un sistema *behavior-based* richiede di individuare le sequenze di azioni rilevanti in fase di sviluppo, in opposizione al determinarle a run-time come avviene invece nei sistemi deliberativi [15]. Questi moduli sono costruiti con leggi di controllo (a volte anche molto simili a quelle studiate nella teoria dei controlli automatici [39]) implementate in software o in hardware. Ogni *behavior* può ricevere degli input pro-

venienti dai sensori del robot e/o da altri *behavior*, e può trasmettere output agli attuatori e/o ad altri *behavior*. In questo modo un controllore *behavior-based* è sostanzialmente costituito da una rete strutturata di *behavior* interagenti.

Ovviamente, dato che diversi *behavior* possono contemporaneamente perseguire obiettivi differenti, in un dato momento un sistema di controllo *behavior-based* si troverebbe a dover eseguire le azioni preposte dai vari *behavior* che lo compongono. È facile dedurre che una situazione del genere non è sostenibile in quanto alcuni *behavior* potrebbero interferire tra loro in determinate situazioni. Ciò avviene quando l'effetto dell'azione voluta da un *behavior* è in conflitto con un secondo *behavior* concorrente: il caso lampante è quello in cui più *behavior* desiderino fornire comandi differenti ad uno stesso attuatore. La scelta dell'azione da compiere a fronte di diverse alternative possibili è infatti tra le operazioni più significative in sistemi di questo tipo. La soluzione ovviamente richiede una qualche forma di coordinazione dei *behavior*. Nella maggioranza dei sistemi *behavior-based* ci si appoggia su di una gerarchia di controllo cablata nel controllore che impone un'ordine fisso di priorità dei *behavior*; ciò assicura che un solo *behavior* possieda il controllo degli attuatori e che se più *behavior* sono in competizione la precedenza venga data a quello con la più alta priorità [15]. In contrasto a questo approccio, considerato spesso molto restrittivo, sono state suggerite soluzione più flessibili, alcune delle quali descritte in seguito.

È importante ricordare che i *behavior* hanno l'abilità di appoggiarsi ad uno stato interno e che, quando connessi tra loro, possono mantenere rappresentazioni astratte dell'informazione in maniera attiva e distribuita. Di conseguenza, diversamente da quanto accade per i sistemi reattivi, quelli *behavior-based* non sono poi così limitati nell'espressività e nelle capacità di adattamento ed apprendimento [18]. Resta vero però che la filosofia alla base di questo tipo di controllo (e dei sistemi reattivi in generale) prevede di evitare il più possibile l'adozione di modelli simbolici del mondo e di limitarsi al loro utilizzo solo se strettamente necessario per compiti di alto livello che richiedano una certa capacità cognitiva. Il non determinismo intrinseco nell'ambiente reale è stato

infatti la forza motrice che ha spinto al concepimento di questo approccio vedendo nell'utilizzo di rappresentazioni astratte un potenziale pericolo per la robustezza dei sistemi robotici: ideologia che può essere efficacemente sintetizzata con l'aforisma "the world is its own best model" di Rodney Brooks [1].

È noto che gli approcci classici deliberativi, descritti nella sezione 1.2.1, si affidano da una decomposizione top-down e gerarchica della soluzione. Al contrario i sistemi *behavior-based* progrediscono bottom-up partendo da regole reattive con lo scopo di implementare i comportamenti che assicurano la sopravvivenza del robot nell'ambiente (ad esempio la capacità di evitare gli ostacoli). Nuove capacità più complesse sono poi introdotte aggiungendo moduli specifici correlati al compito che il robot deve svolgere, i quali, combinati tra loro, generano il comportamento globale desiderato. Questo approccio sfrutta evidentemente il vantaggio offerto dalle proprietà emergenti dall'interazione tra i vari componenti del sistema, già discusse nella sezione 1.2.3. Infatti è usuale che il comportamento globale desiderato non sia direttamente codificato all'interno del sistema ma sia ottenuto dall'interazione dei *behavior* che lo compongono. Si prenda ad esempio una situazione in cui vi siano più robot presenti in uno stesso ambiente e che a tali robot sia richiesto di assumere il comportamento di alto livello di *flocking*, cioè riprodurre lo spostamento in gruppo tipico degli stormi di uccelli: tale comportamento globale può essere ottenuto dalla combinazione di comportamenti più semplici quali "evita gli ostacoli", "resta vicino al gruppo" e "prosegui in avanti" [18]. I singoli *behavior* sono comunemente progettati in modo che, per produrre il comportamento complessivo desiderato, i loro effetti interagiscano prevalentemente nell'ambiente invece che internamente al sistema [15][18].

I controllori *behavior-based* godono per loro natura di un certo livello di modularità messa in luce anche da questo approccio di sviluppo bottom-up. Il concetto classico di "modularità" nell'ambito dei sistemi software prevede che sia favorito lo sviluppo di ogni modulo indipendentemente dagli altri; purtroppo nel caso dei sistemi *behavior-based* i nuovi moduli aggiunti durante il processo di sviluppo spesso non possono essere progettati in maniera piena-

mente indipendente dagli altri, ma devono invece essere sviluppati ragionando sul sistema nella sua interezza e quindi tenendo a mente anche i *behavior* già presenti [15].

Per quanto riguarda i rapporti tra il paradigma *behavior-based* e l'approccio ibrido è possibile notare che la differenza chiave sta nel modo in cui le rappresentazioni simboliche e le relative scale temporali sono gestite [18]. Le architetture ibride si affidano a sistemi reattivi di basso livello che gestiscono informazioni e decisioni correlate al breve periodo e a pianificatori di alto livello per la manipolazione di modelli astratti e per la deliberazione sul lungo periodo. Diversamente i sitemi *behavior-based* tentano di rendere le rappresentazioni simboliche del mondo, e le relative scale temporali, uniformi in tutto il sistema [18]. Spesso infatti i modelli astratti del mondo, quando presenti, sono distribuiti sui *behavior* e magari gestiti sotto forma di processi attivi, al fine di soddisfare le richieste real-time delle altre parti del sistema[16]: questo è relativamente inevitabile dal momento che il controllore per sua natura non è centralizzato e quindi sarebbe incoerente prevedere di fondere le percezioni provenienti da tutti i sensori in un'unica posizione in cui mantenere un modello completo (al contrario degli approcci deliberativi) [15]. Per giunta i sistemi *behavior-based* tipicamente non impiegano una divisione gerarchica come quelli ibridi (e deliberativi) ma invece sono integrati in una struttura distribuita ed omogenea. Come i sistemi ibridi possono fornire sia controllo di basso livello che deliberazione di alto livello: il secondo è effettuato proprio dalle rappresentazioni simboliche distribuite ed attive appena descritte, a volte sfruttando direttamente i *behavior* di basso livello e i loro output [18].

La potenza, l'eleganza e la complessità di questi sistemi deriva dal modo in cui i suoi *behavior* costituenti sono definiti ed utilizzati. Questi possono essere progettati a vari livelli di astrazione. In generale sono più astratti delle azioni atomiche eseguibili dal robot e si estendono nel tempo e nello spazio ma dovrebbero comunque restare relativamente semplici. Per questi motivi è diffi-

16 La descrizione di un esempio che segue questo approccio è data in [16].

cile definirli precisamente, ma allo stesso tempo costituiscono anche un ricco mezzo per fornire interpretazioni innovative [18].

Come nota conclusiva si può menzionare il fatto che l'approccio *behavior-based* si è dimostrato adatto anche allo sviluppo di sistemi multi-agente in cui un gruppo di robot collaborano (o competono) per svolgere un task. In circostanze di questo tipo il comportamento globale dimostrato dal gruppo è implicitamente derivato dalle interazioni locali che si verificano tra i singoli robot. Questo argomento non verrà trattato, ma chiunque fosse interessato può trovare dei riferimenti in [16].

2.1 I precursori dei sistemi behavior-based

Fino ad ora è stato affermato che le strategie di controllo reattive e *behavior-based* sono state sviluppate in reazione all'incapacità delle strategie deliberative di operare efficacemente in ambienti incerti e non strutturati. In realtà questi paradigmi trovano precursori in tempi più antichi ed in ambiti diversi.

Verso la fine degli anni '40 è nata una scienza denominata “cibernetica” [40] che consiste sostanzialmente in un connubio tra la teoria dei controlli automatici, la scienza dell'informazione e la biologia, con lo scopo di studiare i principi di controllo e comunicazione comuni sia agli animali che alle macchine. Alcuni ricercatori di questo campo asserivano che un organismo vivente potesse essere paragonato ad una macchina che utilizzi la matematica sviluppata per i sistemi di controllo con feedback per esprimere un comportamento naturale [1]; questo portò anche all'introduzione della nozione di “situatedness” (che può essere tradotto con “essere situati”) avente la semantica di un forte accoppiamento tra l'organismo (la macchina) e l'ambiente in cui questo vive.

Tra la fine degli anni '40 e l'inizio degli anni '50 William Grey Walter applicò questi principi nella creazione di un semplice robot denominato simpaticamente “tartaruga” per via del suo aspetto fisico [1]. Questo robot era comandato da un circuito elettronico analogico collegato ad una fotocellula direzionale per rilevare sorgenti luminose, un sensore di contatto per rilevare collisioni e due “cellule nervose” costituite da due tubi a vuoto (valvole termoioniche, predecessori dei transistor). Gli attuatori comandati dal circuito erano un motore per sterzare la ruota anteriore, a cui era fissata anche la fotocellula[17], e un motore per mettere effettivamente in movimento la ruota.

Il comportamento assunto dal robot gli permetteva di: esplorare l'ambiente allontanandosi dalle forti sorgenti luminose (*antiphototaxis*), effettuare uno spostamento a fronte di una collisione in modo da evitare gli ostacoli (questo comportamento prevale su quello dettato dalla fotocellula), invertire il compor-

17 Con la possibilità quindi di scansionare l'ambiente circostante durante la rotazione della ruota/fotocellula attorno all'asse di sterzo.

tamento di repulsione dalle sorgenti luminose quando il livello della batteria scende sotto una certa soglia dirigendosi verso la stazione di ricarica sulla quale era applicata una forte sorgente luminosa (*phototaxis*), ripristinare infine il comportamento di *antiphototaxis* allontanandosi dalla stazione di ricarica una volta che la batteria è stata completamente ricaricata.

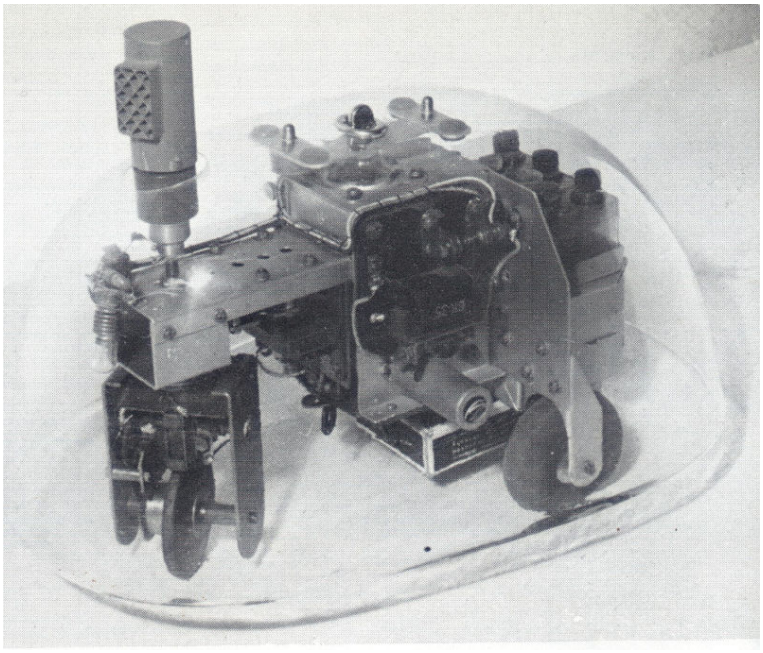


Figura 19: Tartaruga di Grey Walter.

Questo semplice esperimento ha introdotto molti dei concetti tipici dei sistemi *behavior-based* tra cui [1]:

- i *behavior* dovrebbero essere modellati ed implementati nella maniera più semplice possibile;
- risulta spesso necessario scegliere l'azione da compiere a fronte di richieste diverse generate da *behavior* diversi riconoscendo il *behavior* più significativo nella situazione corrente;
- il robot deve possibilmente essere sempre in moto diminuendo le probabilità di rimanere fisicamente bloccato;
- spesso ci si trova ad interagire con entità che costituiscono attrattori o repulsori per il robot (come ostacoli o forti sorgenti luminose).

Circa trent'anni dopo Valentino Braitenberg riprese queste idee e le estese conducendo una serie di esperimenti mentali riguardanti la progettazione di una collezione di quattordici veicoli [1]. Per mezzo di accoppiamenti elementari i sensori di questi sistemi influenzano direttamente gli attuatori, tramite inibizione o eccitazione, dando alla luce comportamenti globali anche molto complessi. Per esempio, semplicemente accoppiando in varie modalità dei sensori di luminosità ai motori di un veicolo a guida differenziale è possibile riprodurre una vasta serie di comportamenti che, dal punto di vista dell'osservatore, possono essere interpretati come paura, aggressione, amore, etc. nei confronti delle sorgenti luminose [1]. Nelle figure 20, 21 e 22 sono descritti i primi tre veicoli.

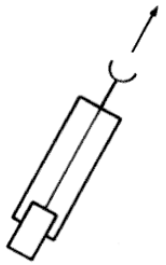


Figura 20: *Veicolo di Braitenberg 1*
Un sensore di una determinata grandezza fisica (luminosità, temperatura, etc.) direttamente connesso ad un motore: il robot si muove più o meno velocemente in funzione dell'intensità dello stimolo rilevato dal sensore (è attirato o respinto).

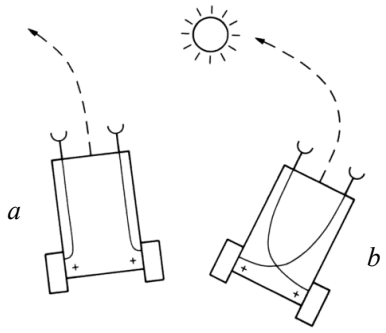


Figura 21: *Veicolo di Braitenberg 2*
Due sensori eccitano direttamente due motori a guida differenziale; a seconda di come vengono impostate le connessioni si possono ottenere comportamenti differenti: nel caso *a* il veicolo è respinto dalla sorgente della grandezza fisica (paura), nel caso *b* invece ne è vigorosamente attratto (aggressione).

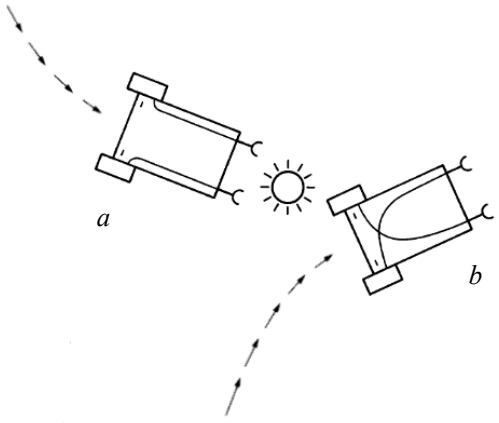


Figura 22: *Veicolo di Braitenberg 3*
Due sensori inibiscono direttamente due motori a guida differenziale; a seconda di come vengono impostate le connessioni si possono ottenere comportamenti differenti: nel caso *a* il veicolo si avvicina lentamente alla sorgente luminosa (amore), nel caso *b* invece si gira lentamente allontanandosi come fosse in cerca di altro (esplorazione).

2.2 Esprimere i behavior

Come è stato più volte ripetuto nei capitoli precedenti, non esiste una definizione precisa e formale di *behavior* comunemente adottata, né tanto meno è stato definito un approccio universale per esprimere i sistemi *behavior-based*: ne sono stati infatti introdotti diversi durante il progredire della ricerca in questo campo e in genere la scelta di quello più adatto può essere fatta di volta in volta sulla base dello specifico task e ambiente di lavoro.

Nel libro *Behavior-Based Robotics* [1] Ronald Arkin raccoglie e descrive alcune delle nozioni più importanti riguardanti i sistemi *behavior-based* tra cui alcune modalità di rappresentazione dei *behavior* sintetizzate di seguito.

Stimulus-Response Diagrams

Questa è la modalità più intuitiva e meno formale per rappresentare graficamente un sistema *behavior-based* in cui ogni *behavior* è rappresentato semplicemente come una risposta generata ad un dato stimolo: infine l'output di ogni *behavior* è incanalato in un meccanismo di coordinazione che determina l'azione complessiva intrapresa dal robot.

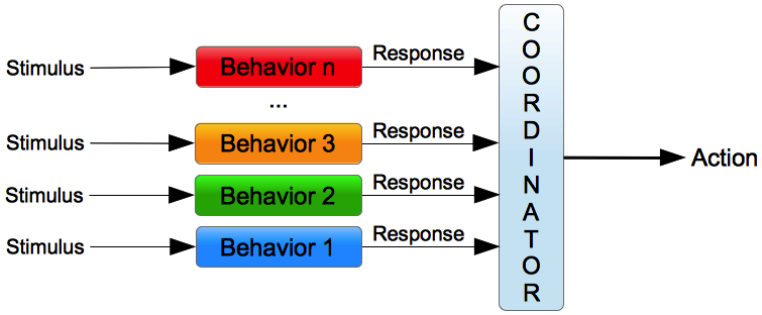


Figura 23: Esempio di diagramma *stimulus-response*.

Functional Notation

Seconda la notazione funzionale è possibile descrivere le stesse relazioni in maniera più formale. Ogni *behavior* è rappresentato da una funzione matematica $b(s)=r$ stante a significare che il *behavior* b produce la risposta r allo stimolo s (in un sistema puramente reattivo il tempo non è un parametro di b). Una funzione coordinatrice può poi essere rappresentata formalmente come:

```
coordinate-behaviors(
  behavior_1(stimulus),
  behavior_2(stimulus),
  behavior_3(stimulus),
  ...
  behavior_n(stimulus)
)
```

e può essere riutilizzata a sua volta in ingresso ad un secondo coordinatore di più alto livello generando formulazioni ricorsive del tipo:

```
coordinate-behaviors(
  coordinate-behaviors(behavioral-set_1),
  coordinate-behaviors(behavioral-set_2),
  ...
)
```

in cui i `behavioral-set` sono insiemi di *behavior* primitivi.

Finite State Acceptor (FSA) Diagrams

Questo tipo di diagrammi è utilizzato in genere per rappresentare le aggregazioni e le sequenze di *behavior* coinvolte durante l'esecuzione del task assegnato al robot. Si tratta di un automa a stati finiti $M=\{Q,\delta,q_0,F\}$ con Q insieme degli stati, $\delta=Q\times input\rightarrow Q$ funzione di transizione, q_0 stato iniziale, F insieme di stati accettori (stati finali che indicano il completamento del task). Ogni stato rappresenta una fase significativa del task e a ciascuno di essi è associato l'insieme dei *behavior* che devono essere necessariamente attivi durante tale fase.

Metodi formali

Usare modelli formali per esprimere i *behavior* fornirebbe ai sistemi di controllo una serie di proprietà molto interessanti tra cui:

- facilitare la generazione automatica di sistemi di controllo;
- un linguaggio completo e comune per esprimere i *behavior*;
- un framework per condurre analisi rigorose del controllore;
- supporto per la progettazione di linguaggi di programmazione di alto livello.

Diversi modelli formali sono stati introdotti tra cui vengono ricordati:

- **Robot Schema** – Metodo formale per esprimere programmi di controllo distribuiti: utilizza un'algebra di processo per comporre una rete di processi (*behavior*) denominati *schemi*. Gli *schemi* comunicano tra loro con scambio di messaggi sincroni attraverso porte di input/output predefinite. Ogni *schema* può essere definito ricorsivamente come composizione di altri *schemi* eseguiti in modalità parallela, sequenziale, condizionale o iterativa.
- **Situated Automata** – Sostanzialmente è un'estensione del modello di automa a stati finiti che riconosce la relazione fondamentale che sussiste tra il robot e l'ambiente in cui è immerso. Questo modello adotta una definizione in stile dichiarativo del programma di controllo tramite formalismi logici che permettono in seguito di generare automaticamente i circuiti digitali (hardware) per il controllo del robot. L'utilizzo di logiche formali permette infine di effettuare delle analisi sul sistema verificandone le proprietà.

2.2.1 Mapping comportamentale

Per codificare la risposta agli stimoli si deve creare un "mapping funzionale" tra il dominio degli stimoli e quello motorio [1]. Un *behavior* può essere espresso come un'associazione $\beta : S \rightarrow R$ con S dominio degli stimoli interpretabili e R insieme delle possibili risposte.

Ogni stimolo $s \in S$ è rappresentabile da un tupla $s = (p, \lambda)$ in cui p rappresenta la classe (il tipo) a cui lo stimolo appartiene e λ l'intensità dello stimolo; alcuni stimoli possono determinare anche solo un cambiamento interno dello stato del sistema in alternativa ad una vera e propria risposta motoria. Ad un *behavior* può essere associato anche un guadagno scalare g usato per modificare la risposta $r \in R$ in $r' = g \cdot r$[18].

L'insieme delle risposte R generate da un *behavior* può essere un insieme discreto o continuo. Nel caso discreto R è composto da un insieme numerabile di tipi di risposta predefiniti tra cui il *behavior* è tenuto a scegliere l'azione da

18 Ovviamente la natura dell'operazione indicata col simbolo "$\cdot$" deve essere definita in maniera coerente con R: se ad esempio le risposte r sono dei vettori numerici l'operazione "$\cdot$" può indicare un prodotto vettore per scalare.

compiere (azioni del tipo “turn-right”, “go-straight”, “travel-at-speed-5”, “stop”, etc.). Nel caso continuo le risposte fornite dal *behavior* possono spaziare con continuità su tutto il dominio di *R*.

Normalmente i *behavior* le cui risposte seguono una codifica discreta sono costituiti da regole del tipo *IF perceptual-condition THEN motor-output* (a volte sfruttando logiche *fuzzy*).

Al contrario i *behavior* che adoperano una codifica continua normalmente calcolano le risposte sul momento per mezzo di funzioni matematiche con codominio coincidente col dominio di *R*. Un esempio classico di codifica continua è quella usata per la navigazione basata su campi potenziali in cui i goal costituiscono degli attrattori, gli ostacoli costituiscono dei repulsori e l'output dei *behavior* segue il gradiente del campo potenziale risultante con l'obiettivo di raggiungere un attrattore evitando i repulsori.

2.3 Assemblare i behavior

Nei capitoli precedenti è già stata trattata quella particolare proprietà che prende il nome di "emergenza" e che si verifica quando diversi *behavior* vengono integrati in un sistema *behavior-based*. Spesso questo termine viene usato per indicare un comportamento complessivo che magari è inatteso o difficilmente deducibile a partire dai *behavior* componenti. È importante sottolineare che tutto sommato però i *behavior* e le funzioni coordinatrici sono algoritmi deterministici e computabili, quindi questo non determinismo e queste proprietà emergenti sono in effetti da ricercare nell'interazione tra il robot e l'ambiente. Quest'ultimo infatti è di per sé imprevedibile e impossibile da modellare in maniera completa e soddisfacente, ed è proprio ciò che porta al "comportamento emergente" [1].

Nel seguito si analizzano le possibili tecniche di coordinazione che si rendono necessarie per permettere ai *behavior* di un sistema di controllo di cooperare al fine di ottenere il risultato sperato. La notazione usata comprende [1]:

- S – denota il vettore degli stimoli s_i rilevanti per i *behavior* β_i percepiti in un dato istante t (con la forma vista nella sezione 2.2.1);
- B – denota il vettore dei *behavior* (funzioni di mapping) β_i attivi all'istante t;
- R – denota il vettore delle risposte r_i generate dai *behavior* β_i (con la forma vista nella sezione 2.2.1);
- G – denota un vettore che codifica l'importanza relativa (guadagno scalare) g_i associata ad ogni *behavior* β_i.

Riprendendo i concetti espressi nella sezione 2.2.1, dato un istante t la risposta fornita da un *behavior* sarà $r_i = \beta_i(s_i)$ e il vettore completo delle risposte R è dunque ottenuto come $R = B(S)$. In definitiva, applicando anche il vettore dei guadagni G si ha $R' = G \cdot R$ in cui il simbolo "·" indica un'operazione di scala per cui ogni risposta r_i viene modulata dal corrispondente guadagno g_i: $r_i' = g_i \cdot r_i$. Una funziona coordinatrice C è definita come $\rho = C(R')$ in cui ρ rappresenta la risposta globale del sistema di controllo agli stimoli S (nell'i-

stante *t*) diretta agli attuatori (normalmente dello stesso tipo delle risposte r_i). Infine le funzioni coordinatrici possono essere definite in maniera ricorsiva per operare non solo sugli output (di basso livello) dei *behavior*, ma anche sugli output provenienti da altre funzioni di coordinazione:

$$\rho' = C'([C_1(R_1'), C_2(R_2'), \ldots, C_n(R_n')])$$

Le funzioni coordinatrici possono essere implementate in qualsiasi modo ma generalmente si suddividono in due macro-classi che, se necessario, potrebbero essere composte fra loro [1].

Metodi competitivi

I metodi competitivi sono metodi di tipo "winner-takes-all" in cui, a fronte di conflitti, viene scelto un singolo *behavior* vincitore la cui risposta viene trasmessa in output:

- **Arbitration** – il vincitore viene scelto da un arbitro normalmente sulla base di una "rete di priorità" fissa in cui, tramite meccanismi di soppressione ed inibizione, la dominanza di un *behavior* su di un altro è imposta in fase di progetto. L'esempio per eccellenza di architettura che usa tecnica è l'architettura a sussunzione.[19]
- **Action-Selection** – in ogni dato istante ciascun *behavior* indica il proprio "livello di attivazione", cioè il desiderio (la necessità) di controllare gli attuatori secondo le intenzioni (gli obiettivi) del *behavior*, dopo di che il coordinatore seleziona a run-time il *behavior* con il livello di attivazione massimo; nessun ordinamento viene quindi predefinito in fase di progetto:

  $$C: R' = R[MAX(activation(\beta_1), activation(\beta_2), \ldots)]$$

  Le architetture che adottano questa tecnica spesso si affidano anche a meccanismi di "trasferimento dell'attivazione" tra i *behavior* che hanno perciò la capacità di influenzare direttamente i livelli di attivazione altrui tramite segnalazioni di inibizione o stimolazione.

19 Ulteriori esempi di sistemi di controllo relativamente semplici che si affidano alla tecnica di arbitraggio a priorità fissa sono mostrati [27].

- **Voting** – viene progettato un insieme predefinito di risposte r_i e ogni *behavior* ha a disposizione un certo numero di voti g_i che può attribuire a quest'ultime; la risposta che riceve il massimo numero di voti viene trasferita in output: $C: R' = MAX\left(votes(r_1), votes(r_2), ...\right)$

  Un esempio di architettura che si appoggia a questo schema di controllo è la *Distributed Architecture for Mobile Navigation* (DAMN).

Metodi cooperativi

Nei metodi cooperativi tutti gli output generati dai *behavior* vengono utilizzati e fusi insieme generando la risposta complessiva; il metodo più utilizzato prevede di effettuare una combinazione lineare (somma pesata) delle risposte fornite dai *behavior*: $C: R' = \sum\left(g_i \cdot r_i\right)$

L'esempio per eccellenza di architettura che impiega questo metodo è l'architettura *Motor Schemas* (descritta in seguito). Ad esempio nel caso della navigazione basata su campi di potenziale l'output di un *behavior* che attrae il robot verso un goal può essere sommato a quello del *behavior* che allontana il robot dagli ostacoli ottenendo traiettorie più o meno dirette al variare dei guadagni g_i.

Assemblaggi di behavior

Per favorire la modularità dei sistemi *behavior-based* e il loro processo di sviluppo può essere introdotto il concetto di "assemblaggio di *behavior*" [1]. Un assemblaggio è un gruppo di *behavior* regolati internamente da un coordinatore che implementa complessivamente un comportamento di base, fornendo pertanto una certa nozione di astrazione e di modulo da poter riutilizzare come "building block" per la costruzione di sistemi *behavior-based*.

2.3.1 Architetture behavior-based

Diverse architetture *behavior-based* sono state sviluppate nel tempo e tutte sono accomunate dai seguenti fattori [1]:

- enfasi sull'importanza di accoppiare strettamente percezione e azione (sensori e attuatori);
- evitare la rappresentazione simbolica della conoscenza e dell'ambiente;
- decomporre il problema in unità basilari significative (*behavior*).

Fattori profondamente distintivi delle varie architetture sono invece [1]:

- granularità della decomposizione in *behavior*;
- metodo di codifica delle risposte (discreto o continuo);
- metodo di coordinazione (competitivo o cooperativo);
- metodo di programmazione, disponibilità di linguaggi di supporto, riusabilità del software di controllo.

Facendo un paragone tra le possibili architetture adottate in robotica e i disparati linguaggi di programmazione che si sono sviluppati nel tempo si può giungere alla conclusione che, come accade ai linguaggi di programmazione, tutte le architetture sviluppate hanno stessa espressività computazionale e differiscono solo nei principi organizzativi e nelle astrazioni adottate [1]. Come per i linguaggi di programmazione ogni stile architetturale ha mantenuto il proprio utilizzo in un particolare dominio (o nicchia).

Per quanto riguarda in generale le architetture puramente reattive e quelle *behavior-based* tale dominio è quello degli ambienti di lavoro incerti ed imprevedibili, per i quali quindi è molto difficile e molto rischioso costruire un modello simbolico sul quale affidarsi per la pianificazione delle azioni da intraprendere [1]. Benché queste architetture non utilizzino quindi un'esplicita rappresentazione simbolica del mondo, come più volte detto, possono tuttavia mantenere implicitamente una rappresentazione ed uno stato interno e pertanto sono in grado di eseguire anche compiti sequenziali ed estesi nel tempo, che richiedono perciò il continuo mantenimento ed aggiornamento di uno stato.

Le architetture *behavior-based*, grazie alle differenze che presentano nel modo in cui sono espressi i *behavior* e nel modo in cui questi sono coordinati, offrono agli sviluppatori una discreta varietà di approcci ciascuno dei quali è più o meno adatto alle particolari situazioni in esame. Per dare un'idea di cosa significhi in pratica adottare una specifica architettura *behavior-based* e di come risultino strutturati i sistemi di controllo sviluppati con essa, si procede con un'esplorazione dettagliata di alcune fra le più importanti architetture a cominciare dall'innovativa architettura a sussunzione.[20]

20 Alcuni esempi di sviluppo di sistemi di controllo sono mostrati in [1], [3], [4], [16] e [27].

2.4 Subsumption Architecture

L'architettura a sussunzione è la prima vera architettura *behavior-based* considerata talvolta proprio il punto di partenza di tutta la robotica comportamentale. Venne introdotta intorno alla metà degli anni '80 da Rodney Brooks nei laboratori del *Massachusetts Institute of Technology*. La scarsa reattività ed efficacia degli approcci deliberativi fu la motivazione principale che portò allo sviluppo di questa filosofia ricalcando il comportamento degli insetti e degli animali.

L'architettura a sussunzione, nella sua formulazione originale, si basa su nove principi dogmatici [3]:

- Un comportamento complesso non deve necessariamente essere il prodotto di un sistema di controllo estremamente complesso.
- I componenti e le interfacce in un sistema dovrebbero essere semplici: se questo non si verifica forse è il caso di rianalizzarli e riprogettarli.
- L'idea è quella di favorire lo sviluppo di robot economici di utilizzo quotidiano che devono pertanto essere adatti a convivere con gli umani.
- La realtà che circonda questi robot è tridimensionale quindi anche gli algoritmi di controllo dovranno considerare tutte le tre dimensioni.
- I sistemi di coordinate assoluti sono fonte di grossi errori cumulativi per i robot ed è quindi preferibile usare mappe relazionali.
- Il mondo non è composto da semplici poliedri quindi non è buona prassi effettuare le fasi di sviluppo e di testing in ambienti artificiali (sia virtuali che fisici) appositamente costruiti per questo scopo.
- Sensori di basso livello, come sensori laser o ad ultrasuoni, forniscono informazioni molto elementari e sono quindi adatti ad interazioni di basso livello (come evitare gli ostacoli vicini), mentre per ottenere informazioni di più alto livello l'opzione migliore è l'utilizzo di dati visuali ottenuti ad esempio tramite telecamere elaborati poi da algoritmi di visione artificiale.
- Perché un robot sia robusto dovrebbe essere in grado di continuare a svolgere i suoi compiti anche a fronte di guasti inerenti ad alcuni suoi sensori.

- Si è interessati a costruire “esseri artificiali” autonomi capaci di sopravvivere per lunghi periodi di tempo senza alcuna assistenza umana.

La soluzione proposta prevede di suddividere il problema di controllo sulla base delle manifestazioni esterne desiderate, ciascuna delle quali costituisce un “livello di competenza” che il robot deve possedere. Un livello di competenza è una specifica informale di una classe di comportamenti (un *behavior*) che un robot deve mostrare in tutti gli possibili scenari operativi [3]. Il processo di sviluppo da adottare è tipicamente un processo iterativo a spirale e fortemente incrementale in cui ogni nuovo livello di competenza è aggiunto sui precedenti ed integrato direttamente su di questi includendone le capacità. Per esempio, nel caso di studio trattato nella proposta originale i livelli suggeriti sono [3]:

0. Evitare il contatto con gli ostacoli.
1. Vagare casualmente per l'ambiente evitando gli ostacoli.
2. Esplorare l'ambiente cercando di raggiungere posizioni lontane che sembrano raggiungibili.
3. Costruire una mappa dell'ambiente e pianificare percorsi da una locazione ad un'altra.
4. Notare cambiamenti nelle “parti statiche” dell'ambiente.
5. Ragionare sugli oggetti presenti nell'ambiente e compiere specifici task correlati ad essi.
6. Formulare ed eseguire piani che prevedono di modificare lo stato del mondo nelle modalità desiderate.
7. Ragionare sul comportamento adottato dagli oggetti presenti nell'ambiente e modificare i piani di conseguenza.

L'idea chiave dell'approccio è che ad ogni livello di competenza può essere associato uno strato di controllo aggiunto di volta in volta al sistema e costruito sui precedenti in maniera bottom-up. Al termine dello sviluppo di uno strato di controllo questo viene testato e perfezionato prima di passare a quello successivo; ciò comporta inoltre il vantaggio di avere un sistema di controllo piena-

mente operativo, seppur molto semplice, anche subito dopo lo sviluppo dei primi livelli di competenza.

L'integrazione tra i livelli è permessa dal fatto che i livelli superiori possono esaminare i dati elaborati dai livelli inferiori e, quando desiderano controllare il robot, possono influire sul loro normale flusso di controllo sopprimendo gli output di tali livelli o iniettando informazioni al loro interno. Questo vale solamente in relazione ai livelli superiori nei confronti di quelli inferiori: i livelli inferiori infatti non hanno alcuna conoscenza dei livelli superiori (conseguenza diretta del fatto che questi vengono sviluppati prima e si occupano di competenze di basso livello). Questo è il meccanismo di base secondo il quale i *behavior* vengono integrati e coordinati: ogni qualvolta un livello di competenza influenzi un livello inferiore si dice che il primo "sussume"[21] il ruolo del secondo; da ciò deriva il nome dell'architettura [3].

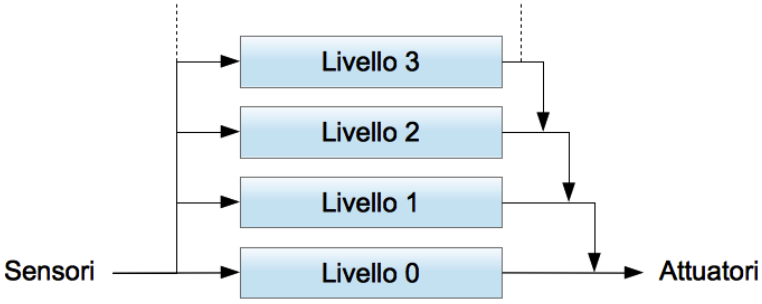


Figura 24: Schema di controllo a livelli in cui i livelli superiori sussumono i ruoli dei livelli inferiori quando desiderano prendere il controllo del robot.

Questa architettura gode di molte delle proprietà richieste per i sistemi di controllo (viste nella sezione 1.2) tra cui: perseguimento contemporaneo di obiettivi multipli (ciascuno dei quali è affidato ad un livello di competenza), capacità di gestione di svariati sensori (da parte dei vari livelli), estensibilità (aggiungendo nuovi livelli di competenze), robustezza (se un livello fallisce o non reagisce in tempo i livelli inferiori possono continuare a controllare il robot in maniera corretta).

21 Il verbo "sussumere" significa inquadrare un caso particolare nell'ambito di un concetto più generale che lo comprende, al fine di analizzarlo sulla base di un contesto più ampio.

Dal punto di vista pratico ogni *behavior* può teoricamente essere implementato in qualsiasi modalità, volendo anche riprendendo i paradigmi classici. Nella proposta dell'architettura a sussunzione ogni livello è composto da un insieme di moduli attivi e asincroni (denominati “processori”) che si scambiano messaggi elementari tramite collegamenti diretti e che non condividono alcuna memoria globale: questo modello è stato scelto anche perché scala molto facilmente in relazione alle capacità computazionali disponibili, ad esempio in base al numero di processori su cui il sistema può essere distribuito [3]. Ogni modulo possiede un certo numero di input e di output ed è modellato come una *Augmented Finite State Machine* (AFSM), cioè un'automa a stati finiti con l'aggiunta di timer e di registri di memoria per mantenere variabili interne (variabili d'istanza). Questi moduli computano in maniera completamente asincrona tra loro monitorando i messaggi ricevuti in input, inviando messaggi sui collegamenti di output, e mantenendo un eventuale stato interno. I messaggi possono essere perduti in quanto ad ogni input di un modulo è associato un registro che contiene solamente l'ultimo messaggio ricevuto. All'avvio del sistema ogni AFSM si trova nello stato di default *nil* e vi ritorna ogni qualvolta riceva un qualsiasi messaggio sulla speciale linea di input *reset*. Ogni stato di una AFSM possiede un nome identificativo e può essere di uno specifico tipo scelto tra i quattro disponibili [3]:

- *output* – un messaggio di output viene calcolato ed inviato tramite una linea di uscita, dopo di che si transita in uno stato futuro.
- *side effect* – il valore di una variabile interna viene aggiornato, dopo di che si transita in uno stato futuro.
- *conditional dispatch* – viene computato un predicato booleano e, a seconda del risultato, si transita in uno di due possibili stati futuri.
- *event dispatch* – viene continuamente monitorata una sequenza di coppie `<condizione, stato futuro>` finché non se ne verifica una e si transita nello stato corrispondente; queste condizioni sono combinazioni booleane di ricezione di messaggi di input o dello scadere di timeout misurati, tramite timer, a partire dall'istante in cui inizia l'azione. I timer di tutte le AFSM

hanno stesso identico periodo, denominato "tempo caratteristico" del sistema di controllo, ma non sono sincronizzati tra loro [5].

Per quanto riguarda invece i collegamenti tra i vari moduli questi sono modellati come "fili" virtuali che mettono in comunicazione un output di un modulo con un input di uno o più altri moduli. In un periodo di tempo pari a quello caratteristico su ogni canale può essere trasmesso un solo messaggio [4].

Al fine di supportare il meccanismo della sussunzione in realtà un output proveniente da un modulo può anche terminare direttamente su di un altro collegamento in corrispondenza dell'output o dell'input di un secondo modulo. Nel primo caso si tratta di *inibizione* e ciò implica che se il primo modulo invia un messaggio su tale collegamento, per un certo periodo di tempo[22], tutti i messaggi generati dal secondo modulo vengono bloccati e non raggiungono quindi le loro destinazioni. Nel secondo caso invece sono possibili due opzioni: *soppressione* e *default*[23]. L'opzione di *soppressione* è equivalente alla *inibizione* con la differenza che i messaggi provenienti dal primo modulo vengono trasmessi in input al secondo modulo, sostituendo quindi i messaggi originali diretti a tale input. L'opzione *default* invece costituisce la modalità duale permettendo ad eventuali messaggi originali diretti al secondo modulo di sostituire quelli provenienti dal primo [4]; in pratica con questo tipo di collegamento è possibile connettere un output di un modulo ad un input di un secondo modulo a cui sia già connesso un output di un terzo modulo dando priorità ai messaggi originali provenienti da quest'ultimo. Com'è facile dedurre, l'integrazione di un livello di competenza sugli altri avviene proprio attraverso questi meccanismi di *inibizione*, *soppressione* e *default* con i quali gli output dei moduli di un livello superiore influenzano o dominano (sussumono) il normale flusso dei livelli inferiori.

22 Nella prima versione descritta in [3] il tempo di inibizione/soppressione è una costante temporale (di qualche secondo) definita in fase di progettazione, mentre nella più moderna versione descritta in [4] tale intervallo di tempo è pari al doppio del tempo caratteristico, cosicché un modulo che desidera inibire/sopprimere un certo canale può ottenere questo effetto inviando continuamente messaggi su tale canale.

23 L'opzione *default* inizialmente non era disponibile nella prima versione dell'architettura a sussunzione descritta in [3].

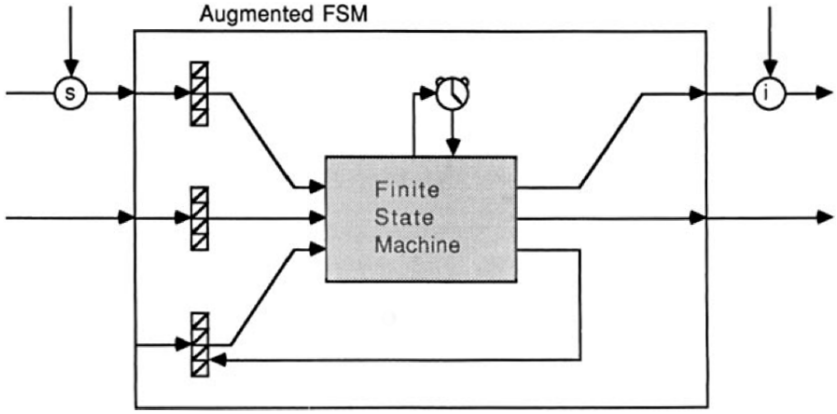


Figura 25: Una AFSM è composta da linee di output (che possono essere inibite), registri associati a variabili d'istanza o a porte di input (a cui è possibile connettere altri segnali con *soppressione* o *default*), timer, ed infine una macchina a stati finiti classica.

Un sistema di controllo a sussunzione viene generalmente rappresentato con uno schema grafico in cui i moduli sono rappresentati da blocchi, i collegamenti da linee e i meccanismi di inibizione, soppressione e default da circonferenze con inscritta una lettera identificativa: “i” in caso di inibizione, “s” in caso di soppressione e “d” in caso di default[24].

L'interazione diretta con le componenti hardware del robot, quali sensori ed attuatori, sono gestite internamente da moduli specifici e spesso non vengono indicate formalmente nello schema grafico. Di norma si affida la gestione di un attuatore da un solo modulo, assicurandosi così di non aver interferenze sugli output del sistema di controllo, mentre le letture dei sensori possono essere effettuate liberamente da parte di qualsiasi modulo [3].

Per avere un'idea di come appare uno schema di controllo a sussunzione in figura 26 è illustrato il sistema di controllo sviluppato per *Genghis* [4], un robot dotato di sei zampe indipendenti ciascuna delle quali possiede due gradi di libertà (mostrato in figura 7): ogni zampa è collegata al corpo tramite un'articolazione (“spalla”) che le permette di ruotare attorno a due assi; uno orizzontale che porta la zampa ad alzarsi e/o abbassarsi di un angolo *beta*, e uno verticale che invece ne determina il movimento in avanti e/o indietro di un angolo *alpha*. L'obiettivo di questo progetto era permettere al robot di camminare efficace-

24 Nella versione iniziale dell'architettura a sussunzione descritta in [3] all'interno della circonferenza è indicato anche la durata temporale dell'effetto.

mente e mantenere una buona andatura anche a fronte di terreni accidentati; come ultimo requisito viene aggiunta la capacità di seguire un essere umano che gli passi davanti rilevandone il movimento tramite l'utilizzo di sensori ad infrarossi passivi.

Per approfondire i dettagli di questo esempio si consiglia di leggere [4].

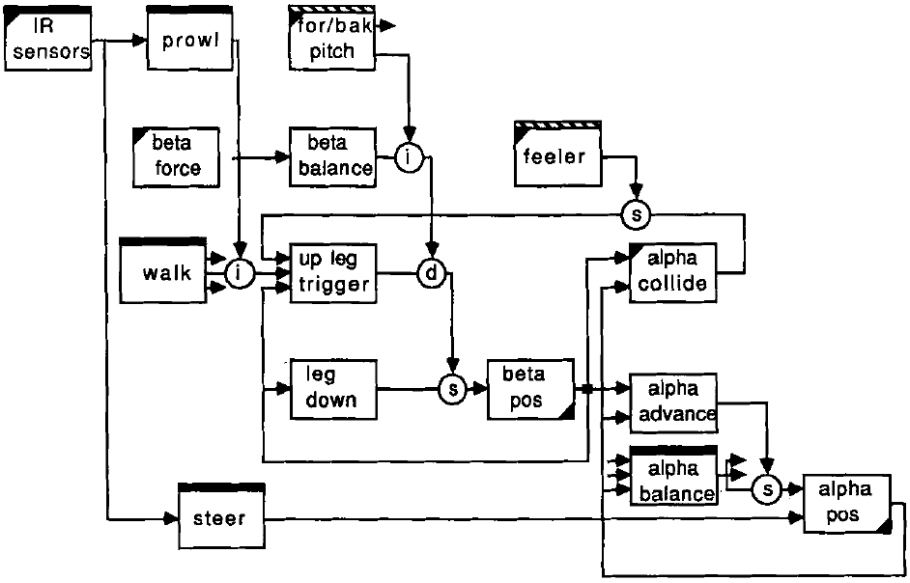


Figura 26: Controllore di *Genghis*: le AFSM con il bordo superiore sottile sono ripetute sei volte, una per zampa; le AFSM con la banda nera sul bordo superiore sono uniche; le AFSM con la banda striata sul bordo superiore sono ripetute due volte, per due gruppi diversi di zampe; le AFSM con un triangolo pieno in alto a sinistra ricevono input da alcuni sensori; le AFSM con un triangolo pieno in basso a destra controllano alcuni attuatori.

A livello implementativo i moduli e i collegamenti sono descritti con il *Subsumption Language*, un linguaggio di specifica *Lisp*-like. I moduli sono specificati definendone gli input, gli output, le variabili d'istanza e l'elenco degli stati come descritti poco prima. I collegamenti vengono invece definiti specificandone la sorgente ed un elenco di destinazioni: la sorgente è sempre un output di un modulo mentre una destinazione può essere un input di un modulo (compreso il *reset*), eventualmente corredato da un'azione di *soppressione* o *default*, oppure un output di un modulo corredato da un'azione di *inibizione*.

I dettagli di questo linguaggio possono essere trovati in [2], [3] e [4] insieme ad alcuni esempi di sistemi di controllo a sussunzione.

Seppure l'architettura a sussunzione è considerata particolarmente importante e significativa in quanto costituisce la prima vera e propria “rottura” con gli approcci classici e la prima architettura di controllo “alternativa”, è stata comunque soggetta a diverse critiche [1]. Prima fra tutte viene criticata la poca flessibilità posseduta delle soluzioni generabili con questo approccio soprattuto nei confronti delle dinamiche che si presentano a tempo di esecuzione. Infatti il meccanismo di coordinazione adottato dalla sussunzione è a priorità fissa e quindi il sistema non può adattarsi dinamicamente durante il suo funzionamento né tantomeno può variare la rilevanza dei *behavior* in funzione della situazione corrente. Una seconda importante critica mossa nei confronti di questa architettura riguarda la sua struttura a livelli che, per i livelli superiori, porta allo sviluppo di *behavior* non completamente indipendenti ma, al contrario, fortemente legati al funzionamento dei livelli inferiori, minando così gravemente la modularità e le possibilità di riuso dei componenti del sistema.

Successivamente all'introduzione dell'architettura a sussunzione, anche con lo scopo di semplificarne l'utilizzo, Rodney Brooks ha sviluppato il *Behavior Language*, un linguaggio che fornisce astrazioni di più alto livello rispetto all'originale *Subsumption Language* [5]. Un sistema di controllo definito usando il *Behavior Language* viene poi tradotto da un compilatore in un sistema di controllo in vecchio stile costituito da un insieme di AFSM interconnesse, il quale, a sua volta, può essere compilato per l'esecuzione su processori di varie tipologie grazie ad uno schedulatore di basso livello che simula il parallelismo delle AFSM. Con il *Behavior Language* è possibile raggruppare delle AFSM in unità operative gestibili ad un più alto livello di astrazione; inoltre le AFSM vengono sintetizzate automaticamente a partire da regole real-time con un mapping uno-a-uno. Queste regole real-time sono di due tipi: *whenever* ed *exclusive*.

Whenever

Una regola *whenever* è utilizzata per reagire al verificarsi di un determinato evento e presenta la sintassi:

```
(whenever condition &rest body-forms)
```

La semantica del linguaggio prevede che la condizione di una regola *whenever* venga continuamente controllata da un "processo" dedicato: ogniqualvolta la condizione risulti vera il corpo `body-forms` viene eseguito sequenzialmente dopo di che il processo torna a controllare la condizione[25]. Una condizione può essere costituita da: la ricezione di un messaggio in uno dei registri di input, lo scadere di un timeout, una combinazione in AND o OR di condizioni dei due tipi appena citati, la valutazione di un arbitrario predicato *Lisp*.

Caratteristica particolarmente importante è che una regola *whenever* può essere innestata a sua volta nel corpo `body-forms` di un'altra regola di livello superiore[26].

Exclusive

Una regola *exclusive* è utilizzata per monitorare in contemporanea un insieme di regole *whenever* e presenta la sintassi:

```
(exclusive &rest whenever-forms)
```

La semantica del linguaggio prevede che le condizioni di tutte le regole *whenever* indicate nel corpo `whenever-forms` vengano continuamente controllate da un "processo" dedicato il quale, non appena si verifica una di tali condizioni, si preoccupa dell'esecuzione della corrispondente regola *whenever*, ignorando temporaneamente le altre.

Anche le regole *exclusive* possono essere innestate nel corpo `body-forms` di regole *whenever*.

Una AFSM di primo livello può essere definita singolarmente con la sintassi:

```
(defmachine name declarations rule)
```

in cui `rule` è una regola real-time.

25 È lecito anche che il corpo di una regola non termini mai.

26 A tale scopo è definita una particolare istruzione che permette di uscire da una regola *whenever* interna che altrimenti continuerebbe ad essere valutata all'infinito senza ritornare alla regola in cui è inclusa.

Ad ogni AFSM sono associati registri di input (a cui è possibile anche assegnare un valore iniziale) e porte di output le cui definizioni vengono inferite sintatticamente dai riferimenti ritrovati nella specifica della regola implementata dall'AFSM.

Collezioni di AFSM possono essere raggruppate in *behavior* col vantaggio che i questi forniscono un'astrazione di granularità meno fine rispetto alle semplici AFSM, e che un *behavior*, e quindi tutte le AFSM che include, può essere attivato o disattivato secondo diversi meccanismi.

La regola grammaticale per definire un *behavior* è:

`(defbehavior` *name &key inputs outputs decls processes*`)`

in cui *inputs* è l'elenco dei registri di input, *outputs* è l'elenco delle porte di output e *processes* è l'elenco di regole real-time (*whenever* e *exclusive*) relative alle AFSM che compongono il *behavior*.

Regole real-time raggruppate in *behavior* condividono le porte di input e di output e comunicano tra loro attraverso connessioni definite implicitamente. Segue è un esempio di *behavior*:

```
(defbehavior tester
  :inputs (f1 f2)
  :decls ((total :init 0))
  :processes ((whenever (received? F1)
                (setf total (+ total f1)))
              (whenever (received? F2)
                (setf total (- total f2)))))
```

Per connettere tra loro *behavior* e/o AFSM singole si utilizza la primitiva *connect* che è definita come:

`(connect` *source dest1 &rest more-dests*`)`

in cui sorgente e destinazioni sono porte di determinati *behavior*. È possibile infine specificare i meccanismi di inibizione, soppressione e default usando come destinazione `(inhibit` *output-port*`)`, `(suppress` *input-port*`)` o `(default` *input-port*`)`. La semantica di una connessione prevede che ogni qualvolta venga assegnato un valore alla porta sorgente tale valore sia trasmesso a ciascuna delle destinazioni indicate.

Con il *Behavior Language* vengono introdotti anche concetti ispirati all'architettura *Action-Selection* quali il concetto di "attivazione" dei *behvaior*: ogni *behavior* può trovarsi in due stati: attivo o inattivo. Come conseguenza diretta di questa caratteristica è possibile specificare ulteriori regole real-time, oltre a quelle indicate nella sezione `processes`, in altre due sezioni denominate `h-processes` e `i-processes`. Le regole indicate nella sezione `h-processes` sono messe in esecuzione solamente quando il *behavior* è nello stato attivo, mentre quelle indicate nella sezione `i-processes` sono mantenute sempre in esecuzione ma nel caso in cui il *behavior* sia inattivo tutti gli output generati da tali regole sono automaticamente inibiti.

Ad ogni *behavior* è infine associata un'espressione booleana (*precondition*), un'espressione numerica (*activation*) ed un valore di soglia (*threshold*); queste tre entità vengono usate per determinare lo stato di attivazione di un *behavior*. La funzione *activation* viene regolarmente valutata e il *behavior* è considerato attivo se il valore restituito da questa è maggiore della *threshold* e contemporaneamente la *precondition* è verificata.

Il meccanismo di attivazione dei *behavior* può quindi essere impostato come si desidera definendo correttamente queste entità. A livello di linguaggio sono disponibili un paio di meccanismi di base. Uno è basato su di un sistema "ad ormoni" in cui è possibile rendere le espressioni *activation* dipendenti dal valore di particolari variabili numeriche il cui valore decresce continuamente nel tempo e che possono essere "eccitate" (incrementate) da qualsiasi regola real-time. Il secondo è basato sulla possibilità di "diffondere l'attivazione" aumentando direttamente il valore di un registro di attivazione associato a ciascun *behavior*, di una quantità non superiore al corrente livello di attivazione del *behavior* che esegue tale operazione: questo registro di attivazione può poi essere referenziato nell'espressione *activation* del *behavior* a cui appartiene.

I dettagli sul *Behavior Language* possono essere trovati in [5].

2.5 Motor Schemas

L'architettura *Motor Schemas* è stata introdotta da Ronald Arkin poco dopo quella a sussunzione e si differenzia da questa per una serie di ragioni. La più rilevante tra esse riguarda il fatto che questa architettura è molto più guidata da principi presi in prestito dalla biologia rispetto a quanto lo sia quella a sussunzione. Nello specifico questa architettura si basa sulla teoria psicologica degli *schemi*[27]. Tale teoria fornisce una serie di caratteristiche e principi base che possono essere sfruttati per sviluppare sistemi *behavior-based* [1]:

- il comportamento motorio può essere spiegato in termini di un controllo concorrente da parte di attività differenti;
- uno *schema* codifica sia come reagire sia come la reazione possa essere realizzata;
- il modello computazionale è intrinsecamente distribuito;
- percezione ed azione sono connessi fra loro;
- dei livelli di attivazione sono associati agli *schemi* e ne determinano la disponibilità o l'adeguatezza all'azione;
- l'apprendimento è reso possibile sia attraverso l'acquisizione di nuovi *schemi* sia tramite l'ottimizzazione degli *schemi* presenti.

È evidente che il concetto di *schema* può essere ricondotto a quello di *behavior*. La teoria degli *schemi* comporta per di più alcune implicazioni sui sistemi di controllo autonomi [1]:

- fornisce un livello di granularità meno fine per esprimere la relazione tra il controllo di moto e le percezioni (in contrasto ad esempio con modelli quali le reti neuronali);
- permette di modellare una soluzione in termini di entità distribuite (che collaborano o competono) quindi direttamente implementabile su architetture parallele;

27 Secondo questa teoria uno *schema* è un mezzo attraverso il quale un individuo è in grado di categorizzare le percezioni sensoriali durante il processo di apprendimento di conoscenza o di esperienza, e può essere impiegato per modellarne il comportamento [1].

- fornisce un insieme di primitive comportamentali che possono essere sfruttate per costruire *behavior* più complessi (assemblaggi);
- può avvalersi del supporto e delle future scoperte delle scienze cognitive e delle neuroscienze.

Le caratteristiche principali che contraddistinguono questa architettura dalle altre sono le seguenti [1]:

- le risposte fornite dai *behavior* sono vettori di formato uniforme generati secondo l'approccio basato sui campi di potenziale (codifica continua della risposta);
- la coordinazione dei *behavior* consiste nella loro cooperazione ottenuta tramite somma vettoriale delle risposte che forniscono;
- non esiste una gerarchia di coordinazione dei *behavior* impostata in fase di progettazione ma, al contrario, il sistema può essere configurato a run-time in accordo con la situazione corrente e gli schemi possono essere istanziati ed eliminati durante l'esecuzione dando così vita ad una rete di *behavior* in continuo cambiamento;
- l'arbitraggio puro non viene mai usato e ogni *behavior* contribuisce alla risposta complessiva del robot a vari livelli.

L'architettura *Motor Schemas* è basata su *behavior* solitamente di alto livello che possono essere riutilizzati in varie circostanze e che cercano in genere di riprodurre i comportamenti degli animali (almeno per quanto riguarda gli spostamenti in ambienti dinamici e non strutturati).

In questa architettura particolare importanza viene data al ruolo della percezione, tant'è che in ogni *schema* sono inclusi dei *perceptual schema*. Questi secondi processi hanno il determinato scopo di elaborare le informazioni proveniente dai sensori a cui sono interessati e produrre informazioni percettive appositamente strutturate e preparate per lo specifico *motor schema* a cui sono associati (caratteristica denominata *action-oriented perception*) [1]. Dopo di che i *motor schema* reagiscono agli stimoli forniti da queste percezioni. Ogni

perceptual schema può a sua volta essere decomposto in *perceptual subschema* che elaborano solo parte dell'informazione disponibile.

Ciascun *motor schema* fornisce in output un vettore in cui sono codificati i comandi di spostamento che desidera fornire agli attuatori; la coordinazione tra i vari *behavior* avviene per mezzo di una somma vettoriale degli output forniti. L'influenza di ogni *behavior* è determinata dal valore del guadagno *g* che vi è stato associato e che rievoca il concetto di livello di attivazione menzionato in precedenza in merito alla teoria degli schemi. Il vettore di output di ogni *behavior* viene moltiplicato per il guadagno relativo e sommato a quello degli altri; il risultato è un vettore che, una volta normalizzato in qualche modo, rappresenta la risposta complessiva del sistema di controllo. Questo processo complessivo di percezione e reazione viene ripetuto continuamente alla massima velocità possibile e gli *schemi* operano in maniera totalmente indipendente ed asincrona fornendo output nel minor tempo possibile [1]. In genere i valori dei guadagni assegnati agli schemi restano costanti durante l'esecuzione tuttavia è possibile estendere l'architettura mettendo in campo tecniche che permettono di modificare il valore di tali parametri dinamicamente arricchendo il sistema di controllo con capacità di adattamento ed apprendimento.

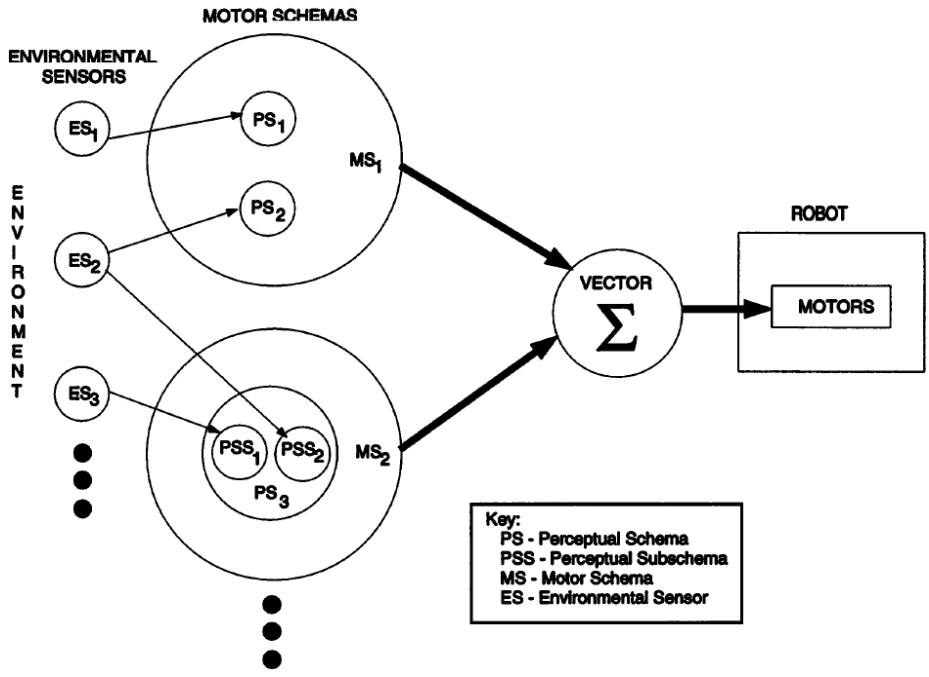


Figura 27: Esempio di architettura *Motor Schemas*.

L'utilizzo per eccellenza che ne è stato fatto di questa architettura riguarda la navigazione di robot mobili adottando l'approccio basato su campi di potenziale. Secondo tale approccio ciascun *behavior* in ogni istante fornisce in output il vettore gradiente del campo potenziale a cui è associato. Basandosi sull'ipotesi che il campo di potenziale globale in cui il robot è immerso sia ottenuto come combinazione lineare dai campi di potenziale associati ai vari *beha*vior, la risposta istantanea del robot sarà anch'essa una combinazione lineare (con gli stessi coefficienti) delle risposte fornite da questi *beha*vior.

Per i dettagli riguardanti questa architettura, la navigazione basata su campi di potenziale e i problemi che questa comporta si rimanda a [1].

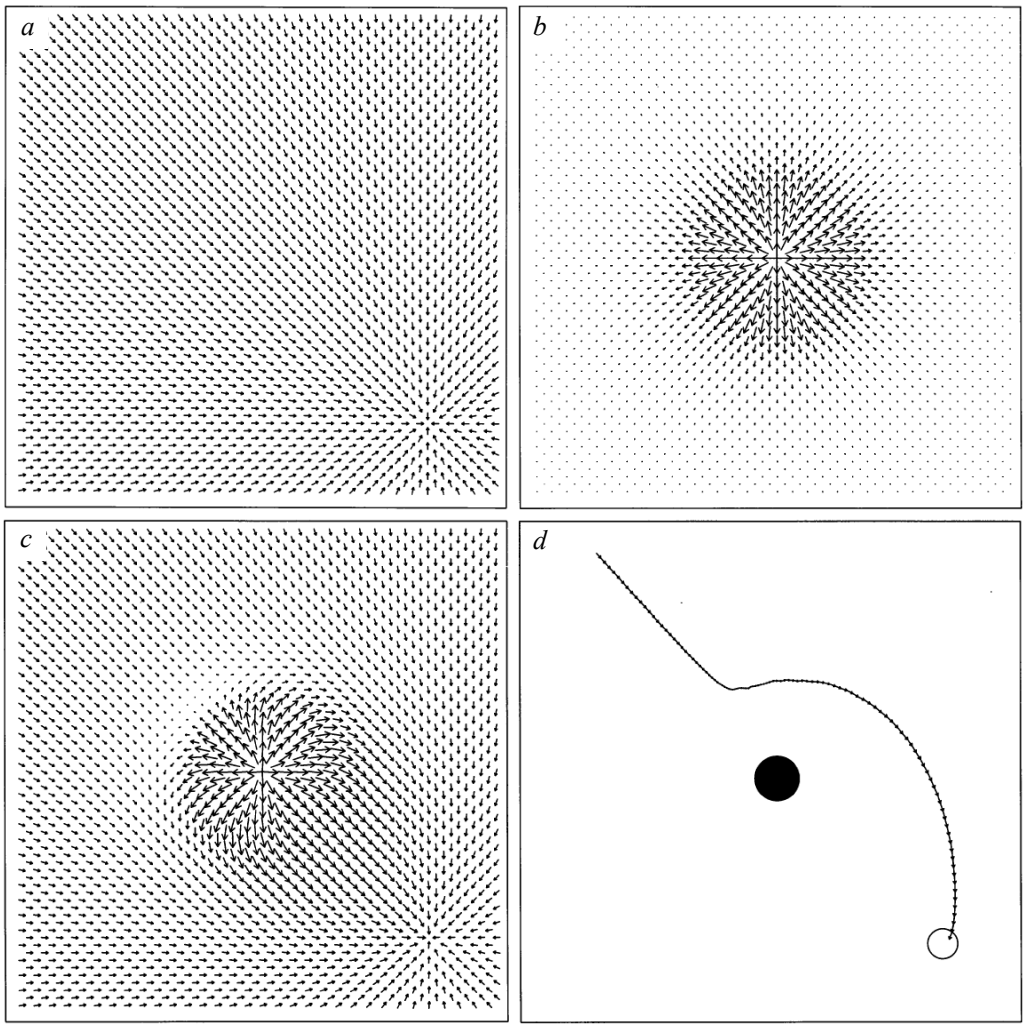


Figura 28: Esempio di navigazione basata su campi di potenziale: i campi generati dai *behavior* di "attrazione verso un goal" (*a*) e "repulsione verso gli ostacoli" (*b*), combinati tra loro, danno vita al campo globale (*c*) e perciò alla possibile traiettoria mostrata in *d*.

2.6 Altre architetture

Sull'onda delle architetture a sussunzione e *Motor Schemas* tante altre architetture *behavior-based* sono state sviluppate nel tempo ciascuna con le sue peculiarità che riguardano soprattuto i meccanismi di coordinazione adottati, il metodo di codifica delle risposte e la granularità dei *behavior*.

Tra queste vale la pena citare [1]:

- **Action-Selection** – usa un meccanismo dinamico per la selezione dell'azione da compiere basata su precondizioni e livelli di attivazione associati ai *behavior* che si possono stimolare o inibire a vicenda (metodo competitivo già descritto nella sezione 2.3).
- **Colony Architecture** – discendente diretta dell'architettura a sussunzione ma leggermente più flessibile e più semplice di questa (si appoggia al solo meccanismo di soppressione).
- **Circuit Architecture** – adotta un approccio gerarchico secondo il quale un *behavior* può essere costituito da un assemblaggio di più basso livello di astrazione permettendo alla coordinazione di agire ad ogni livello. Questo modello gode di proprietà significative in quanto i *behavior* sono espressi attraverso logiche formali permettendone la compilazione diretta in hardware e facilitandone la verifica automatica delle prestazioni.
- **Distributed Architecture for Mobile Navigation (DAMN)** – possiede un modello di coordinazione unico basato su di un insieme discreto di possibili risposte che vengono “votate” dai vari *behavior*, dopo di che quella che ha ricevuto più voti viene eseguita (metodo competitivo già descritto nella sezione 2.3).

Recentemente almeno un altro paio di architetture sono state proposte più o meno innovative dal punto di vista dei meccanismi di coordinazione e della modellazione dei *behavior*. Vengono discusse qui di seguito.

2.6.1 Extended Utility Function (EUF) method

Secondo questo metodo il software di controllo prende il nome di *brain* (cervello) ed è costituito da processi concorrenti chiamati *brain processes*. Questi si dividono in tre tipi [25]:

- *cognitive processes* – che non agiscono sugli attuatori;
- *locomotive behaviors* – che desiderano spostare fisicamente il robot;
- *movement behaviors* – che riguardano movimenti che non condizionano lo spostamento del robot (ad esempio il movimento di braccia meccaniche).

Locomotive e *movement behavior* costituiscono i *behavior* motorii del robot.

Il metodo EUF si affida al concetto di "utilità" introdotto da von Neumann e Morgenstern nel contesto della teoria dei giochi. Viene utilizzato un s*ensory preprocessing system* (SPS) con lo scopo di filtrare i valori ricevuti dai sensori e mapparne il contenuto informativo in variabili di stato. Ciascun processo è equipaggiato con una "funzione di utilità" con codominio [-1,1] utilizzata per valutare l'utilità di ogni processo in ogni istante: questa funzione è definita in termini delle variabili di stato e di altri parametri che possono essere utilizzati per modularne la forma oppure per forzare manualmente l'attivazione o la disattivazione di un *behavior* (un *behavior* può influenzare l'attivazione di un altro *behavior* variando uno di questi parametri).

Le funzioni e i parametri modificabili sono gestiti con equazioni differenziali che ne garantiscono un andamento con un certo grado di continuità evitando crescite e decrescite istantanee ed irregolari. Il valore assegnato ai parametri delle funzioni di utilità è cruciale per il comportamento complessivo assunto dal robot e possono essere determinati utilizzando algoritmi di ottimizzazione stocastica.

Ad ogni ciclo di controllo il metodo consiste nel coordinare i vari *brain processes* nel seguente modo:

- tra tutti i *cognitive processes* vengono attivati (concorrentemente) tutti quelli con utilità superiore a 0;

- tra tutti i *locomotive behaviors* viene attivato solamente quello con utilità massima;
- tra tutti i *movement behaviors* viene attivato solamente quello con utilità massima.

In ogni istante possono quindi essere attivi un numero qualsiasi di *cognitive processes* e al massimo un *locomotive behavior* e un *movement behavior*.

L'idea alla base dell'architettura è cercare di riprodurre il procedimento di selezione dell'azione più conveniente utilizzato dagli animali in funzione dei propri bisogni primari (determinati internamente al sistema) e della situazione corrente dell'ambiente esterno.

2.6.2 Integrated Behavior-Based Control (iB2C)

Secondo questa architettura un sistema di controllo è costituito da una rete di moduli omogenei interconnessi che dispongono di un'interfaccia standard ed incapsulano la nozione di *behavior*.

Il componente principale di questa architettura è il *behavior module* che impone l'interfaccia standard per tutti i *behavior* [21].

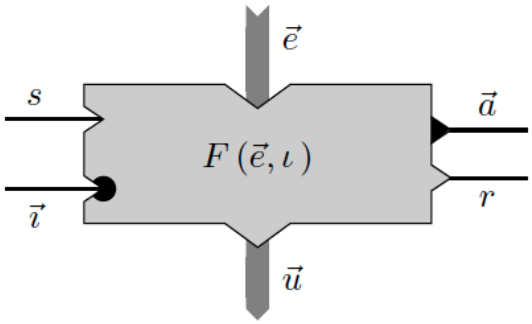


Figura 29: Rappresentazione grafica di un *behavior module* dell'architettura iB2C: sono esplicitamente mostrate le porte di input/output dell'interfaccia standard.

Ogni modulo presenta tre input:

- $\vec{e} \in \mathbb{R}^m$: *input vector* – valori provenienti dai sensori o da altri *behavior*.
- $s \in [0,1]$: *stimulation* – input che cerca di aumentare l'influenza di un *behavior* tramite stimolazione: 1 = massima, 0 = minima. Può essere usato per

impostare la rilevanza di un *behavior* rispetto agli altri (fissando un valore per *s*) oppure per permettere ad un *behavior* di essere "attivato" dagli altri: stimolazioni provenienti da più *behavior* vengono sommate.

- $\vec{i} \in [0,1]^k$, $i = MAX(i_j)$: *inhibition* – input (determinato da *k* altri *behavior*) che riduce la rilevanza di un *behavior* tramite inibizione: 1 = massima, 0 = minima.

L'*activation* $\iota \in [0,1] = s \cdot (1-i)$ di un *behavior* ne indica l'effettiva rilevanza dovuta a stimolazione ed inibizione. La stimolazione o l'inibizione di un *behavior* che non riceva nessuna influenza da altri *behavior* può essere imposta "dall'esterno" della rete, ad esempio con valori costanti assegnati in fase di inizializzazione del sistema.

Ogni modulo presenta tre output:

- $\vec{u} \in \mathbb{R}^n$: *output vector* – dati diretti agli attuatori o ad altri *behavior*.
- $\vec{a} = (a \in [0,1], \underline{\vec{a}} \in [0,1]^q)$: *activity* – l'*activity signal a* (con $a \leqslant \iota$) indica l'influenza del *behavior* nella situazione corrente (1 = massima, 0 = minima), le *derived activities* $\underline{\vec{a}}$ (con $\underline{a}_i \leqslant a$) possono essere usate per influenzare singolarmente altri *q behavior* tramite i loro input *s* e $\vec{i}$.
- $r \in [0,1]$: *target rating* – output che indica la "soddisfazione" del *behavior* nella situazione attuale (0 = massima, 1 = minima).

Ogni modulo è descritto da tre funzioni:

- $F(\vec{e}, \iota)$: *transfer function* – usata per determinare il valore degli output $\vec{u}$. Determina in pratica "l'intelligenza" del *behavior* e può essere implementata in qualsiasi modo: un semplice accoppiamento reattivo input → output, un sofisticato algoritmo (pianificazione), una macchina complessa dotata di stato interno, etc.
- $f_a(\vec{e}, \iota)$: *activity function* – usata per determinare il valore di $\vec{a}$.
- $f_r(\vec{e})$: *target function* – usata per determinare il valore di *r*.

Il secondo componente previsto dall'architettura è il *fusion behavior module* che è un particolare tipo di *behavior module* usato per mediare gli output di altri *behavior*: si tratta in pratica dei coordinatori del sistema. Per quanto riguarda questo modulo il suo input $\vec{e}$ proviene dai *behavior* che deve coordinare e comprende le loro *activity* a_i (una qualsiasi componente delle *activity*), dai loro *target rating* r_i e dai loro *output vector* $\vec{u}_i$.

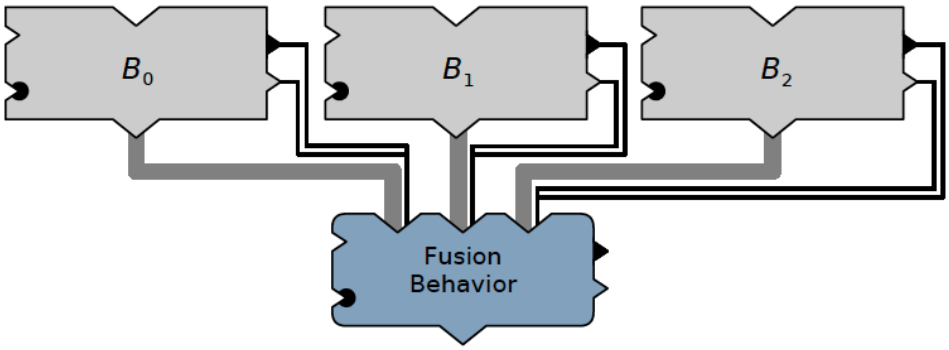


Figura 30: Esempio di *fusion behavior module* che coordina gli output generati da tre *behavior* B_0, B_1 e B_2.

La *transfer function* è in definitiva la funzione che effettua la coordinazione desiderata e deve rispettare l'assunzione che *behavior* con maggiori livelli di attività influiscono di più sul valore degli output. Una serie di moduli di fusione prestabiliti è disponibile di default:

- *maximum fusion* (di tipo "winner-takes-all") – riporta in uscita l'*output vector*, l'*activity* e il *target rating* del *behavior* con *activity* maggiore:

  $w = \underset{c}{argmax}(a_c),\ \vec{u} = \vec{u}_w,\ a = \underset{c}{max}(a_c),\ r = r_w;$

- *weighted fusion* – riporta in uscita una media pesata dell'*output vector*, dell'*activity* e del *target rating* dei *behavior* utilizzando l'*activity* come peso:

  $$t = \sum_{j=0}^{p-1} a_j,\ \vec{u} = \sum_{j=0}^{p-1} \frac{a_j \cdot \vec{u}_j}{t},\ a = \sum_{j=0}^{p-1} \frac{a_j^2}{t} \cdot \iota,\ r = \sum_{j=0}^{p-1} \frac{a_j \cdot r_j}{t} \cdot \iota;$$

- *weighted sum fusion* – riporta in uscita una somma pesata dell'*output vector* e dell'*activity* dei *behavior* utilizzando l'*activity* come peso e divisa per l'*activity massima* (standardizzazione), mentre per quanto riguarda il *target rating* effettua una media pesata come avviene nel caso di *weighted fusion*:

$$m = \max_{c}(a_c),\ \vec{u} = \sum_{j=0}^{p-1} \frac{a_j \cdot \vec{u}_j}{m},\ a = min\left(1, \sum_{j=0}^{p-1} \frac{a_j^2}{m}\right) \cdot \iota,\ t = \sum_{k=0}^{p-1} a_k,\ r = \sum_{j=0}^{p-1} \frac{a_j \cdot r_j}{t} \cdot \iota.$$

In figura 31 è mostrato un esempio di sistema di controllo per un robot mobile che deve seguire un oggetto evitando gli ostacoli (nell'immagine è rappresentata solo la parte relativa alla rotazione del robot): si vede come il modulo "avoid collision" inibisce il modulo "turn to object" in modo da ridurne il livello di attività e risultare così quello più influente.

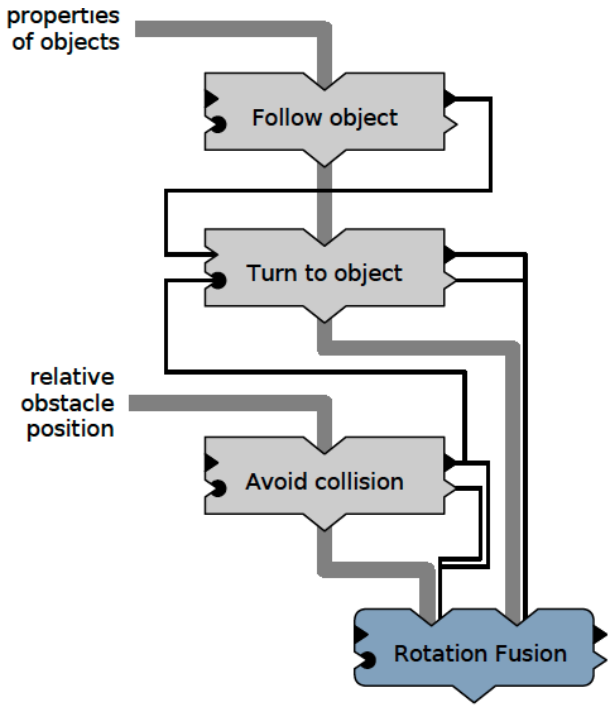


Figura 31: Esempio di sistema di controllo: si nota come i *behavior* interagiscano tramite stimolazione, inibizione e scambio diretto di informazioni.

In [21] è dimostrato che componendo correttamente i moduli dei *behavior* ed i moduli di fusione è in pratica possibile riprodurre tutti i meccanismi di coordinazione storici descritti in letteratura (si vedano quelli descritti nella sezione 2.3). Inoltre, riprendendo il concetto di "design pattern" sviluppato nell'ambito dell'*Object-Oriented Programming*, si possono esprimere alcuni utili pattern ricorrenti nelle reti dei sistemi di controllo iB2C, tra cui ad esempio il pattern *Behavioral Group* che prevede di unire gruppi di *behavior* in un modulo di più alto livello da poter poi riusare liberamente.

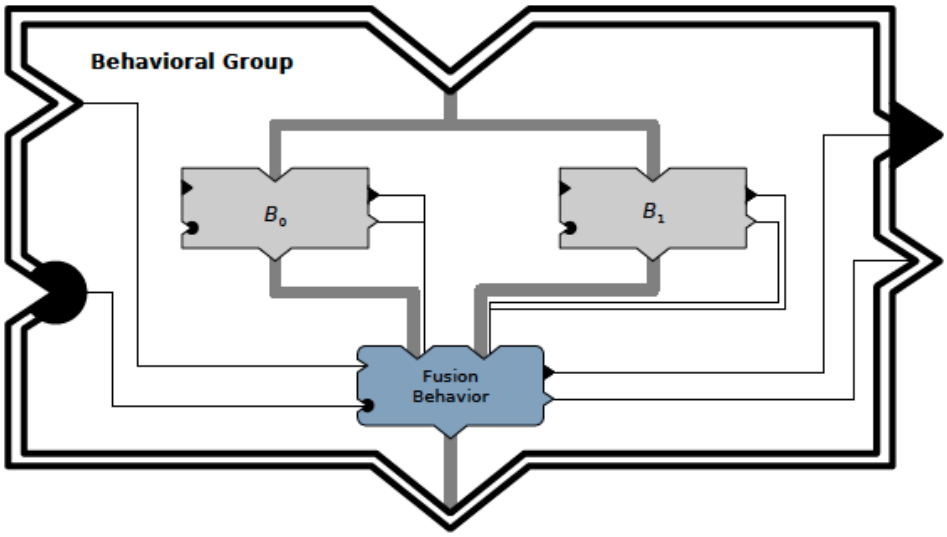


Figura 32: Esempio di *Behavioral Group* con due *behavior* coordinati internamente.

Dal punto di vista implementativo l'architettura iB2C [35] è sviluppato come estensione della *Modular Controller Architecture* (MCA) [32] e si basa su di questa per implementare i moduli descritti. Il framework iB2C/MCA mette inoltre a disposizione una serie di strumenti per il monitoraggio del programma di controllo durante la sua esecuzione: ad esempio è possibile vedere il diagramma dei *behavior* e i relativi valori di stimolazione, inibizione, attivazione e attività, nonché utilizzare strumenti di analisi capaci di individuare particolari proprietà del sistema (come frequenti oscillazioni nei valori di determinati segnali).

I dettagli riguardanti questa architettura e le sue proprietà sono descritti in [21] insieme alla proposta di un insieme di strategie per lo sviluppo di controllori iB2C che riprende le linee guida esposte nella successiva sezione 2.7.

2.7 Linee guida

La labile definizione di *behavior* e la grande varietà di tecniche di progettazione *behavior-based* discusse in letteratura, rende il processo di sviluppo di questi sistemi relativamente poco nitido.

In [15] Maja Matarić espone una serie di strategie che consiglia di seguire come linee guida per la creazione di controllori *behavior-based*.

La decomposizione e la costruzione di un sistema di controllo *behavior-based* inizia con l'identificazione dell'insieme di "riflessi" di base, tipicamente implementati nella forma di semplici regole reattive, in grado di fornire al robot capacità di sopravvivenza negli ambienti dinamici e non strutturati in cui dovrà operare. Il processo è fortemente bottom-up ma anche nella pianificazione dei comportamenti di base è buona norma tenere in considerazione le necessità dei comportamenti che potrebbero essere aggiunti successivamente rispettando, per così dire, ipotetici vincoli "provenienti dall'alto".

Sono inoltre suggeriti tre passi di carattere molto generale per identificare un valido processo di sviluppo per sistemi *behavior-based*:

1. *Specificare il comportamento desiderato in termini qualitativi* – specificare il comportamento complessivo desiderato per il robot dal punto di vista dell'osservatore, determinando i vincoli di funzionamento del controllore.
2. *Specificare il comportamento in termini delle azioni dal punto di vista dell'osservatore* – decomporre il comportamento desiderato in azioni più elementari, disgiunte e singolarmente osservabili dall'esterno.
3. *Specificare le azioni in termini degli attuatori del robot* – selezionare l'insieme di azioni che il robot può compiere (in relazione agli attuatori) e deciderne il livello di granularità, che deve essere almeno fine quanto la precisione richiesta dal task. Per massimizzare la consapevolezza sensoriale è indicato effettuare movimenti brevi ed incrementali, decomponendo un'azione prolungata in tanti passi elementari separati periodicamente dall'analisi delle informazioni fornite dai sensori.

Capitolo 3

Il framework proposto

Come analizzato nei capitoli precedenti, una grande varietà di architetture *behavior-based* sono state introdotte, ciascuna con le proprie peculiarità. A quelle già descritte se ne aggiungono tante altre[28] sviluppate nel contesto di specifici gruppi di ricerca che in genere tuttavia non sono molto supportate tecnologicamente in quanto poco approfondite e quasi mai accompagnate da strumenti di sviluppo adeguati che ne facilito l'utilizzo. Inoltre è evidente come non esista un modello formale comune a tutte le architetture trattate; ciò che più vi si avvicina sono i formalismi messi in campo da Ronald Arkin descritti nelle sezioni 2.2 e 2.3.

In vista dell'espansione a cui sarà prima o poi soggetto il campo della robotica si ritiene dunque opportuno creare un framework di semplice utilizzo che consenta di esplorare e sperimentare in maniera più sistematica la programmazione di robot secondo l'approccio *behavior-based* e che sia tra l'altro fortemente orientato alla fase di simulazione data la grande importanza che questa ricopre nel processo di sviluppo di un robot per valutare le prestazioni dei sistemi di controllo (si ricordi infatti quanto visto nella sezione 1.3).

Prima di tutto è però fondamentale cercare di delineare distintamente il modello di alto livello sul quale la teoria dei sistemi *behavior-based* poggia le proprie fondamenta. Fatto ciò il framework oggetto della discussione dovrà implementare tale modello fornendo così anche uno strumento pratico direttamente impiegabile per lo sviluppo di controllori *behavior-based*.

28 Cercando in letteratura si possono trovare molte proposte di architetture di questo genere: alcune di queste sono riportate tra i riferimenti bibliografici.

3.1 Meta-modello

I sistemi di controllo, classici e non, sono per loro natura macchine cicliche che ad ogni iterazione hanno il compito di determinare i comandi da impartire agli attuatori sulla base delle percezioni ottenute tramite i sensori. L'intuizione alla base delle architetture *behavior-based* è quella di mettere in campo diversi comportamenti, attivi contemporaneamente e potenzialmente contrastanti, che monitorano costantemente (ciclicamente) le percezioni fornite al sistema e agiscono con lo scopo di ottenere (o mantenere) un determinato obiettivo. Questo approccio, contrapposto a quello tipico *sense-model-plan-act*, permette di costruire sistemi relativamente semplici, con buoni tempi di risposta, quindi molto reattivi ed ideali per applicazioni real-time, intrinsecamente modulari ed estensibili, quindi adatti ad un processo di sviluppo incrementale, e composti da elementi interconnessi potenzialmente distribuibili su più supporti computazionali, che quindi scalano naturalmente in relazione alle risorse disponibili.

Sulla base della sintesi appena esposta e riprendendo, estendendo e generalizzando le considerazioni fatte da Ronald Arkin e i formalismi da lui adoperati (descritti nella sezione 2.2), si definisce un modello generico capace di delineare la natura di un sistema di controllo *behavior-based*.

La prima entità considerata è ovviamente il *behavior*. Un *behavior B* può essere modellato formalmente coma una funzione di mapping comportamentale (vedi sezione 2.2.1) che in ciascun istante t genera degli output $o \in O$ elaborando degli input $i \in I$. Nel caso di *behavior* puramente reattivi il valore degli output o è determinato unicamente in relazione al solo valore corrente degli input i, mentre nel caso generale un *behavior* può essere costituito da una macchina complessa e dotata di stato interno $s \in S$ che viene di volta in volta consultato per generare gli output corretti. Un singolo *behavior* può quindi essere rappresentato da una formulazione del tipo $B : I \times S \rightarrow O \times S$ in cui I è il dominio degli input, O è il dominio degli output e S è il dominio dello stato interno del *behavior*. Ovviamente la natura degli elementi di I, O e S dipende dallo specifico caso in esame: ad esempio si può trattare di semplici valori numerici,

oppure di vettori, o addirittura di strutture dati particolari ed arbitrariamente complesse che contengano tutte le informazioni necessarie per la specifica applicazione. La generazione degli output e dello stato futuro da parte di un *behavior* può infine essere indicata con la notazione $B(i^t, s^t) = \{o^t, s^{t+1}\}$ stante ad indicare appunto la generazione dell'output corrente o^t e dello stato preso in considerazione all'iterazione successiva s^{t+1}.

Queste definizioni non sono però sufficienti a catturare tutti i tipi di moduli che compongono un sistema *behavior-based*. Comunemente infatti *behavior* diversi intendono agire sugli stessi attuatori spesso con l'intenzione di raggiungere obiettivi discordanti. A questo scopo sono necessari dei componenti che mediano i comandi inoltrati dai vari *behavior* al fine di ottenere il comportamento globale desiderato. Questi componenti, ampiamente trattati nei capitoli precedenti, sono denominati funzioni coordinatrici (o coordinatori) C e presentano essenzialmente la stessa forma di un normale *behavior*. Solitamente sono costituiti da una semplice funzione diretta $C: I \rightarrow O$ ma è tuttavia plausibile immaginare che in determinate situazioni potrebbero anche rivelarsi necessari coordinatori che sfruttino un proprio stato interno divenendo pertanto in tutto e per tutto equivalenti ai comuni *behavior* dal punto di vista strutturale e dell'interfaccia esterna; sono perciò formalizzabili anch'essi come $C: I \times S \rightarrow O \times S$. Quello che li distinguerà dai normali *behavior* è il fatto che le loro azioni non si focalizzano sul perseguimento di un certo obiettivo, ma solamente sulla coordinazione degli input provenienti da altri *behavior*, comportandosi quindi come "filtri passivi" che riportano in uscita una versione leggermente elaborata[29] di ciò che ricevono in ingresso.

Determinati i modelli di *behavior* e coordinatori resta da specificare come avvengono le interazioni tra questi moduli. Trattandosi in linea teorica di entità parallele ed indipendenti, potenzialmente distribuibili su più nodi computazionali, il modello di interazione più consono è quello che prevede un scambio

29 Questa elaborazione dipende strettamente dal tipo di coordinazione messa in campo (vedi sezione 2.3) ma in ogni caso non deve alterare la semantica dell'informazione trasmessa né tantomeno introdurvi alterazioni arbitrarie.

asincrono[30] di informazioni sotto la forma di messaggi. Entrando nel dettaglio si tratta di instaurare dei collegamenti "virtuali" tra gli output e gli input dei vari moduli attraverso i quali possano "transitare" questi messaggi.

Nel caso comune gli output generati da un *behavior* riguardano direttamente comandi da sottoporre a determinati attuatori e dovrebbero perciò essere diretti a coordinatori incaricati di assicurarvi un corretto accesso da parte di tutti i *behavior* interessati. In generale però si è riscontrato che non è insolito imbattersi in controllori che si appoggino a strategie particolari, spesso ispirate al mondo della biologia, in cui i *behavior* sono direttamente interconnessi fra loro e si scambiano informazioni e/o particolari segnali di attivazione (stimolazione ed inibizione). Di conseguenza tramite collegamenti *inter-behavior* di questo tipo è possibile instaurare "reti di moduli" arbitrariamente complesse.

Un output di un *behavior* può essere connesso ad un numero qualsiasi di input di altri moduli mentre, di norma, ad ogni input di un modulo è collegato un solo output di un secondo *behavior*. Possono in realtà esistere casi particolari di tipologie di input in grado di aggregare informazioni provenienti da più output: si pensi ad esempio al caso di segnalazioni di stimolazione/inibizione provenienti da più moduli e dirette tutte da uno specifico input di un *behavior*.

Secondo il modello teorico i *behavior* sono sempre attivi nel costante tentativo di perseguire l'obiettivo per cui sono stati creati. D'altro canto però è ragionevole pensare a particolari architetture che permettano ad un *behavior* di essere temporaneamente disattivato, magari sulla base dello stato corrente del controllore o dell'influenza di altri *behavior*, ottenendo per di più una maggiore efficienza evitando di eseguire computazioni il cui risultato verrebbe inesorabilmente ignorato.

30 Scambio di informazioni asincrono significa che mittente e destinatario dell'informazione sono completamente indipendenti e non devono essere impegnati contemporaneamente nelle operazioni di trasmissione e ricezione; il mittente è libero di riprendere il proprio lavoro non appena l'informazione è stata inviata, e il destinatario è libero di recuperare un'informazione ricevuta quando più gli aggrada. Come conseguenza diretta è necessario che le comunicazioni avvengano tramite un mezzo che disponga di un meccanismo che funga da "memoria tampone" (buffer) in grado di ricordare temporaneamente le informazioni trasmesse dal mittente e non ancora prelevate dal destinatario.

Confrontando il meta-modello qui esposto con i modelli (spesso informali) presi in considerazione dalle architetture *behavior-based* più diffuse (si considerino ad esempio quelle discusse nel capitolo 2) si può notare come questo si occupi solamente di inquadrare la strategia di controllo *behavior-based* al più alto livello di astrazione possibile. La maggior parte delle architetture infatti si concentrano spesso su di un solo modello di *behavior*, di coordinazione e di interazione tra questi. Sotto tale aspetto in realtà ogni architettura di programmazione *behavior-based* può essere vista come una particolare istanza di questo meta-modello e può essere ottenuta semplicemente puntualizzando certi dettagli ed imponendo alcuni vincoli.

É ovvio che analizzare ed affrontare un problema sulla base di un livello più astratto permette di valutare un maggior numero di alternative rispetto a quanto sarebbe possibile prendendo in uso una specifica architettura. Questo livello di astrazione può infatti essere considerato un punto di forza del modello in quanto meno limitante e molto più flessibile ed adattabile agli specifici casi in esame rispetto a quanto lo siano le classiche architetture di controllo. Seguendo tale modello infatti è possibile supportare la specifica di qualsiasi sistema *behavior-based* che semplicemente sia conforme ai dettami teorici della strategia di controllo comportamentale.

Per contro però, come già discusso in precedenza, la sola base teorica e la mancanza di dettagli e di vincoli lascia allo sviluppatore molte libertà progettuali che potrebbero in realtà rendere più difficoltoso lo sviluppo di un sistema di controllo. Questa mancanza di struttura può ovviamente essere compensata con l'adozione di una specifica architettura che si appoggi ad un modello più preciso che sia istanza del presente meta-modello.

3.2 Architettura del framework

Sulla base del modello appena descritto si procede con la progettazione del framework, denominato per l'occasione *Behavior Control Network* (BCN)[31]. Come anticipato nella sezione 1.3.1 nel seguito di questo lavoro viene utilizzato il simulatore *Webots* e di conseguenza il framework dovrà essere compatibile con quest'ultimo. In realtà ciò non influisce significativamente sulla soluzione adottata che potrebbe essere convertita senza problemi ad un altro simulatore o all'esecuzione diretta sull'hardware di un robot saltando completamente la fase di simulazione. Come linguaggio per l'implementazione viene impiegato *Java* perché più flessibile ed espressivo del semplice linguaggio *C* consigliato dallo stesso *Webots*, anche se ciò comporta degli svantaggi minimi in termini di prestazioni e portabilità.

Il framework BCN assume la forma di una libreria software che fornisce una serie di classi di base che l'utente può utilizzare per creare il proprio controllore. Ovviamente ricalca il modello definito nella sezione precedente e vede come entità centrale il *behavior* usato per rappresentare ogni tipo di modulo presente nel sistema. Questi moduli saranno definiti specificatamente dall'utente in funzione degli obiettivi che devono perseguire e del comportamento complessivo che il robot deve assumere. Una volta determinati questi moduli il controllore sviluppato dall'utente non deve fare altro che creare le istanze dei *behavior* coinvolti nel sistema di controllo, configurare le connessioni tra queste, ed infine affidarne l'esecuzione da uno *scheduler* che si prende carico di far progredire il controllo e di simulare il parallelismo tra i *behavior*.

Behavior

Un *behavior* è modellato come una macchina ciclica le cui iterazioni si ripetono ad intervalli regolari con periodo elementare pari al *control step* definito nella sezione 1.3 sulla simulazione. L'entità *behavior* è rappresentata dalla classe astratta `Behavior` che definisce l'interfaccia comune all'interno della

31 Tutto il codice relativo all'implementazione del framework, dell'IDE Xtext (descritto nella sezione successiva) e del caso di studio (descritto nel capitolo successivo) sono reperibili online all'indirizzo https://sourceforge.net/projects/bcn.

quale devono essere incapsulati tutti i moduli che compongono il controllore. Tale interfaccia dispone di due operazioni:

- `init` – invocata dallo *scheduler* nel momento in cui gli viene affidato il *behavior*: di default non esegue nessuna computazione ma l'utente è libero di reimplementarla per specificare particolari attività di inizializzazione.
- `step` – invocata dallo *scheduler* ciclicamente durante l'esecuzione del controllore: costituisce di fatto la funzione di mapping comportamentale *B* (o la funzione coordinatrice *C*) prevista dal modello ed è il punto in cui l'utente deve codificare il comportamento attuato dallo specifico modulo.

Ogni *behavior* e coordinatore messo in campo dall'utente deve essere rappresentato da una classe *Java* che estenda questa classe base ed implementi il metodo `step` al fine di specificarne il funzionamento.

Per abilitare le comunicazioni tra i moduli della rete ciascuno di essi può affidarsi alle entità `InputPort<T>` ed `OutputPort<T>` fornite dal framework per definirne l'interfaccia in termini di porte di input e di output. Entrambe queste entità sfruttano il meccanismo del polimorfismo parametrico (fornito da *Java*) e sono parametrizzate su di un tipo di dato generico (*template*) `T` associato alla particolare porta, il quale, di fatto, ne vincola l'utilizzo: ciò significa che, data una specifica porta, questa potrà ricevere o trasmettere solamente messaggi modellati sotto forma di oggetti di tipo `T`[32].

InputPort

Comunemente per ogni porta di input è fornito un buffer di dimensione limitata che è in grado di tenere in memoria gli ultimi *n* messaggi ricevuti[33] e che fornisce tre operazioni elementari per agire su tali messaggi:

- `put` – utilizzata dal framework per inserire nel buffer un messaggio proveniente dall'output di un secondo *behavior*;

32 Con “tipo di dato” ci si riferisce all'accezione specifica dei linguaggi *Object-Oriented*, in questo caso quella propria di *Java*.

33 Nel caso in cui fosse sufficiente tenere in memoria solamente un messaggio è consigliato usare un'istanza di classe *InputRegister*, in caso contrario è necessario usarne una di classe *InputBuffer*.

- `get` – utilizzata dal *behavior* per prelevare il messaggio meno recente disponibile nel buffer rimuovendolo;
- `peek` – utilizzata dal *behavior* per recuperare il messaggio meno recente disponibile nel buffer senza però rimuoverlo.

È importante notare che, affinché i *behavior* siano realmente indipendenti come previsto dal modello, deve essere mantenuto il disaccopiamento tra questi, ragione per cui le operazioni di lettura e di scrittura sul buffer di una porta di input sono sempre operazioni non bloccanti. La strategia adottata prevede che se si tenta di inserire un messaggio in un buffer pieno il messaggio meno recente viene eliminato per far spazio a quello nuovo, senza che il mittente resti bloccato in attesa che si liberi dello spazio; dualmente, se si tenta di prelevare un messaggio da un buffer vuoto l'operazione termina istantaneamente con una segnalazione di buffer vuoto[34], senza che il destinatario resti bloccato in attesa di ricevere un messaggio.

Al momento il framework mette a disposizione tre tipi di porte di input:

- `InputBuffer<T>` – una porta di input capace di mantenere in memoria gli *n* messaggi più recenti ricevuti.
- `InputRegister<T>` – una porta di input che memorizza solamente l'ultimo messaggio ricevuto: sostanzialmente è equivalente da una versione semplificata (e più efficiente) dell'`InputBuffer` con *n* posto a 1. Si consiglia di usare questa tipologia di porta ogni volta che per un *behavior* è sufficiente lavorare sul dato più recente fornitogli.
- `InputInfluence` – un particolare tipo di porta di input pensata per permettere ai *behavior* della rete di influenzare in qualche modo il *behavior* a cui la porta appartiene. Il livello di influenza è modellato come un numero in virgola mobile[35] che ricade all'interno di un intervallo limitato superiormente ed inferiormente. Alla ricezione di uno stimolo proveniente da un *behavior* questo viene sommato al valore corrente. Le operazioni `get` e `peek` in questo

34 Viene semplicemente restituito un riferimento *null*.

35 Porte di classe `InputInfluence` sono anche istanze del tipo `InputPort<Float>`.

caso sono equivalenti: restituiscono il corrente livello di influenza e, come effetto collaterale, lo modificano sottraendovi una certa quantità nel tentativo di riportare (più o meno lentamente) tale livello al valore normale. Un *behavior* che presenti una porta di questo tipo per permettere ad altri moduli di trasmettervi segnalazioni tipo stimolazione e/o inibizione, dovrebbe controllare ad ogni iterazione il livello di influenza e comportarsi coerentemente con esso. Per giunta in questo modo il livello di influenza tenderebbe continuamente a riportarsi al valore normale e quindi, a partire dall'istante in cui non vengano più ricevute segnalazioni, l'effetto dovuto all'influenza degli altri *behavior* svanirebbe rapidamente.

Come si può intuire le porte di input sono pensate per ricevere gli output generati da un solo modulo. Unica eccezione sono le porte di tipo `InputInfluence` che sono invece adatte ad aggregare le informazioni provenienti anche da un qualsiasi numero di *behavior* disparati.

OutputPort

Una porta di output permette ad un modulo di comunicare al resto della rete i risultati delle proprie computazioni. Esiste un solo tipo generico di porta di output che dispone di tre operazioni:

- `connectTo` – utilizzata, in genere in fase di inizializzazione del sistema, per aggiungere un numero arbitrario di destinatari alla porta di output;
- `disconnectFrom` – che può essere usata per rimuovere uno dei destinatari associati alla porta di output;
- `send` – operazione principale usata dal *behavior* per trasmettere un messaggio di output a tutti i destinatari che vi sono connessi.

Per collegare una porta di input di uno specifico *behavior* ad un porta di output si decide di affidarsi da un terzo tipo di entità chiamata `InputRef` che funga da riferimento alla suddetta porta di input. Nel momento in cui venga invocato i metodo `send`, il messaggio di output viene trasmesso a tutte le `InputPort` connesse per mezzo dei loro riferimenti indiretti di tipo `InputRef`. Per il momento è possibile creare solamente riferimenti “locali”, istanze della classe `LocalIn-`

`putRef` che incapsulano un riferimento diretto da una `InputPort` locale. Con "locale" si intende nel contesto dello stesso processo software, nel caso di *Java* si tratta della stessa *Java Virtual Machine.* Nulla esclude infatti che in futuro il framework possa essere esteso permettendo una distribuzione fisica del sistema di controllo su più nodi computazionali: in questo caso sarebbe necessario introdurre anche una seconda entità, denominata per esempio `RemoteInputRef`, che permetta di comunicare con una porta di input di un *behavior* in esecuzione su di una differente *Java Virtual Machine* appoggiandosi su qualche protocollo di comunicazione.

All'atto della definizione dei moduli specifici per l'applicazione in esame, l'utente è libero di svilupparli usando un qualsiasi numero di porte di input e di output ed invocando liberamente le operazioni che queste forniscono. Data la natura "aperta" del framework l'utente ha ovviamente anche la possibilità di estendere le entità basilari fornite con la libreria creandone delle nuove più adatte alle sue necessità (ad esempio nuovi tipi di porte di input) oppure, nel caso limite, modificando quelle presenti.

Compito dello sviluppatore è definire i *behavior* che intende utilizzare e, in fase di inizializzazione del controllore, creare le istanze dei *behavior* che compongono la rete ed impostare i collegamenti tra di esse. A seconda di come vengono definiti i *behavior* sarà ovviamente possibile mettere a disposizione di quest'ultimi i riferimenti ai sensori e agli attuatori fisici disponibili sul robot. Fatto ciò è sufficiente creare un'istanza della classe `Scheduler` a cui assegnare tutti i *behavior* della rete ed avviarne l'esecuzione.

Scheduler

Lo *scheduler* è l'entità predisposta ad implementare il ciclo di controllo vero e proprio mettendo in esecuzione i vari *behavior* e simulando il parallelismo che sussiste tra questi. Le operazioni che fornisce sono tre:

- `add` – usata per affidare i *behavior* allo *scheduler*;
- `remove` – che può essere usata per rimuovere in *behavior* dal sistema;

- `run` – operazione fondamentale che esegue il controllore *behavior-based*.

La funzione dell'operazione `run` è in pratica quella di invocare regolarmente i metodi `step` dei vari *behavior* e, nello stesso tempo, di comunicare con il simulatore *Webots* per far progredire la simulazione. Quest'ultimo è l'unico punto in cui si nota la presenza di tale simulatore; modificando infatti questo singolo aspetto il framework può essere facilmente convertito ad un altro simulatore, di simili fattezze, o reso adatto all'esecuzione diretta da parte di un robot reale.

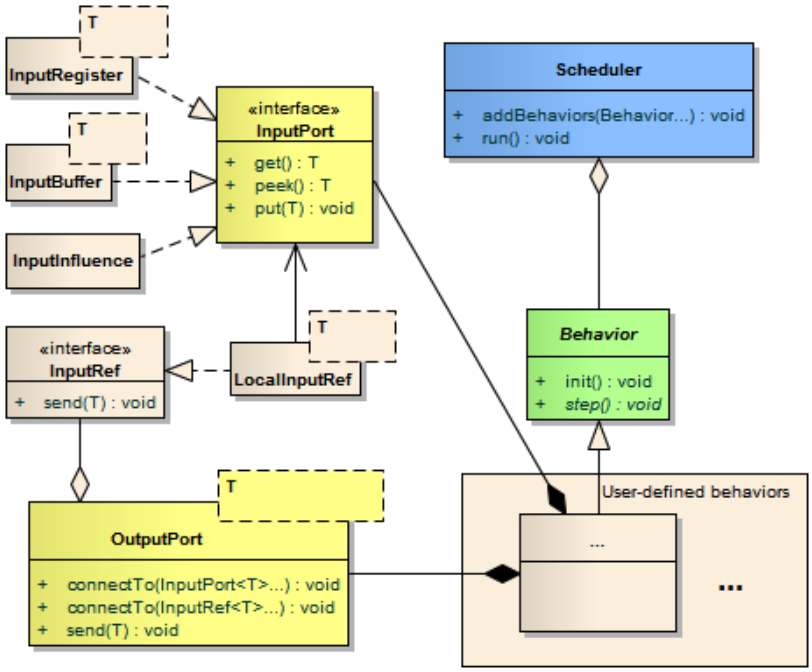


Figura 33: Diagramma delle classi che compongono il framework.

In figura 34 è mostrata una possibile rappresentazione grafica di un controllore *behavior-based* sviluppato con questo framework in cui sono mostrate porte di input, porte di output e connessioni tra *behavior* paralleli. Lo stesso sistema, dopo aver definito i *behavior* `B1`, `B2`, `B3` e `B4`, e le rispettive porte di input e di output, può essere impostato come:

```
public class ExampleController {
  static private final int CONTROL_STEP = 32;

  static public void main(String[] args) {
    Robot robot = new Robot();  // Webots APIs
    // get and init sensors
    SensorType1 sensor1 = robot.get...
    SensorType2 sensor2 = robot.get...
```

```
    // get and init actuators
    ActuatorType1 actuator1 = robot.get...
    // init behaviors
    B1 b1 = new B1(sensor1, sensor2);
    B2 b2 = new B2(sensor2);
    B3 b3 = new B3(2);
    B4 b4 = new B4(actuator1);
    // connect behaviors
    b1.get_out1().connectTo(b2.get_in1(), b3.get_in(0));
    b2.get_out1().connectTo(b3.get_in(1));
    b3.get_out1().connectTo(b4.get_in1());
    // run controller
    Scheduler scheduler = new Scheduler(robot, CONTROL_STEP, new Random(),
                                        b1, b2, b3, b4);
    scheduler.run();
  }
}
```

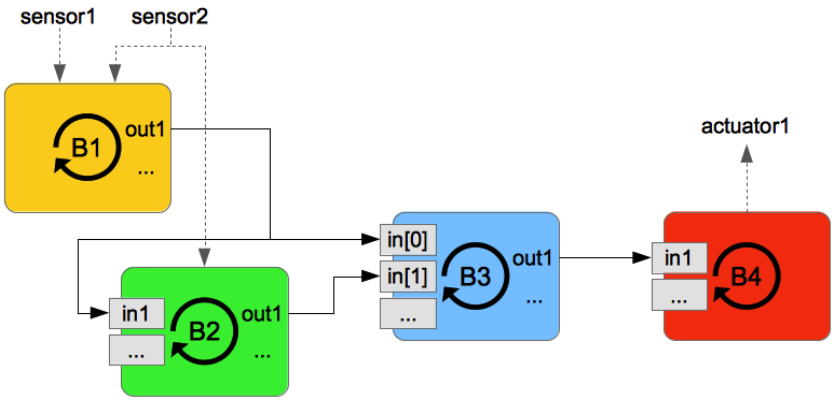


Figura 34: Esempio di schema grafico di un controllore sviluppato con questo framework.

Come è stato più volte ripetuto il modello teorico su cui questo framework si basa prevede che i *behavior* siano entità parallele ed asincrone. La conseguenza che questo comporta è che, nonostante l'esecuzione dei *behavior* corrisponde all'invocazione periodica e regolare delle relative funzioni di mapping comportamentale, non possono essere fatte assunzioni di alcun tipo sull'ordine con cui tali eventi si verifichino. In poche parole dati *n behavior* non è possibile prevederne l'ordine di esecuzione, ma solo il fatto che ogni *behavior* viene richiamato ad intervalli temporali più o meno costanti (pari al *control step*). Questo caso è quindi equivalente agli scenari che si riscontrano tipicamente nel contesto dei sistemi operativi multitasking in cui, anche a fronte di task periodici con stesso identico periodo, l'effettivo ordinamento di questi non può

essere mantenuto equivalente per tutta la durata dell'esecuzione[36]. Per simulare questo parallelismo è pertanto sufficiente che lo *scheduler* si limiti a richiamare i *behavior* in un ordine che sia casuale e ogni volta diverso, obbligando così lo sviluppatore a progettare *behavior* che tengano conto di questo asincronismo. Per esempio, in un sistema in cui ad ogni iterazione un *behavior* `B1` invii un messaggio ad un secondo modulo `B2`, il quale si aspetta di elaborarlo anch'esso regolarmente ad ogni *control step*, può capitare che ogni tanto `B2` non si trovi un input disponibile oppure si ritrovi un input obsoleto (magari già elaborato)[37]: questo accade ogniqualvolta `B2` esegua la propria parte del controllo prima che `B1` abbia la possibilità di trasmettergli l'informazione desiderata. Tutto ciò è normale ed è proprio quel che accadrebbe se i *behavior* fossero realmente in esecuzione su macchine fisiche distinte ed asincrone. Tuttavia, nel caso peggiore, tale messaggio (o uno più nuovo) sarà processato al ciclo successivo di `B2` limitando perciò il ritardo massimo di reazione ad un tempo circa uguale al *control step*, e cioè a poche decine di millisecondi.

Ricordando quanto visto a proposito della simulazione nella sezione 1.3 è bene ricordare che affinché questa sia corretta dal punto di vista delle tempistiche in gioco, ad ogni iterazione la fase di controllo, una volta che il software venga trasferito sull'hardware di un ipotetico robot reale, non deve richiedere un tempo di computazione superiore al *control step*. Da questo punto di vista lo *scheduler* del framework cerca di ottenere i risultati migliori sfruttando tutti i core disponibili nel processore sul quale è in esecuzione. In teoria, se fosse disponibile un processore per ogni modulo il limite temporale per l'esecuzione della funzione di mapping comportamentale di ogni *behavior* sarebbe banalmente uguale al *control step*. Nel caso generico invece il tempo a disposizione di ciascun *behavior* è strettamente dipendente dal numero di flussi “fisici” concorrenti disponibili. In figura 35 sono mostrate due situazioni di *scheduling* differenti: nel caso *a* è disponibile un solo thread di esecuzione, mentre nel caso *b*

36 Se questo dovesse essere necessario si deve per forza ricorrere a tecniche o meccanismi di sincronizzazione.

37 A seconda che usi l'operazione `get` o `peek` della porta di input.

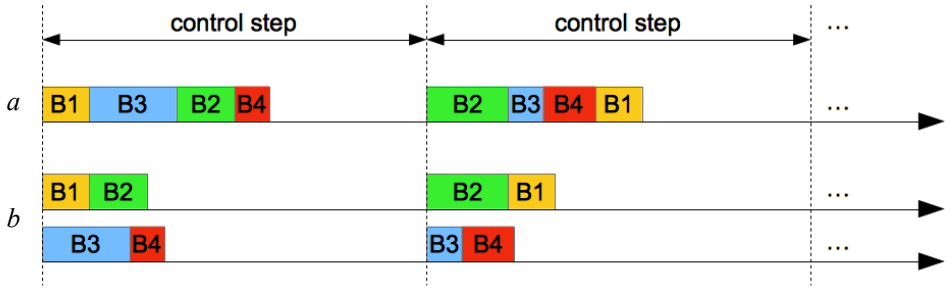


Figura 35: Diverse situazioni di scheduling a seconda del numero di core disponibili.

ve ne sono due. É evidente come nel secondo caso i vincoli temporali siano rispettati con molta più facilità.

Il problema di verificare se l'insieme dei *behavior* messi in campo è effettivamente eseguibile in un tempo non superiore al *control step* è un classico problema di *scheduling* ricorrente nel contesto dei sistemi *hard* e *soft real-time* che però esula dagli obiettivi di quest'opera e non verrà discusso qui.

Per quanto concerne invece le comunicazioni si può considerare un vincolo qualitativo che preveda che ad ogni iterazione possa essere trasmesso non più di un messaggio per ogni singola linea di output. Tale vincolo non viene imposto né verificato in alcuno modo dall'infrastruttura: spetta allo sviluppatore impegnarsi a rispettarlo.

Come nota conclusiva si sottolinea che un sistema di controllo sviluppato con questo framework è facilmente riconfigurabile a tempo di esecuzione. Per esempio si possono aggiungere o rimuovere *behavior*, e si possono aggiungere o rimuovere connessioni ad una porta di output. L'unica cosa che non è di competenza del framework è la generazione di nuove porte di input e/o di output associate ai *behavior* coinvolti in una modifica del sistema: in situazioni del genere spetta infatti all'utente progettare *behavior* dinamici che offrano questa opportunità. La capacità di modificare un sistema durante l'esecuzione dello stesso renderebbe possibile mettere in campo speciali tecniche di adattamento, alcune delle quali molto discusse in letteratura, che permetterebbero al robot di ottenere competenze e prestazioni superiori.

3.3 Coordinatori

Il framework è accompagnato da un insieme di semplici coordinatori già sviluppati e preconfigurati che l'utente è libero di adoperare nei propri controllori. Tutti questi coordinatori sono parametrizzati su di un tipo di dato generico `T` che rappresenta il tipo di messaggi che i *behavior* possono trasmettervi e possiedono un array di input (parametrizzati su `T`) tale che ognuno dei *behavior* che si desidera coordinare deve esservi connesso in una particolare posizione. Inoltre questi coordinatori sono di carattere generico e non si curano minimamente della natura di `T`.

PriorityCoord

Questo coordinatore riproduce una tecnica di coordinazione a priorità fissa. Gli *n behavior* coordinati sono connessi in ciascuna posizione di un vettore di *n* `InputRegister` in ordine di priorità decrescente. Il funzionamento del coordinatore prevede semplicemente di prelevare in sequenza tutti gli input ricevuti e trasmettere in uscita solamente quello con priorità massima. Se un *behavior*, in un dato momento, non desidera trasmettere alcuna informazione è sufficiente che non invii alcun messaggio. Se invece desidera trasmettere un comando, diretto ad esempio ad un attuatore, lo deve fare con la massima frequenza consentita (cioè un messaggio per ogni *control step*) in quanto deve competere con i *behavior* a priorità inferiore che, a loro volta, potrebbero inviare nuove informazioni con continuità.

Alla luce del problema discusso prima riguardante lo scambio di messaggi periodici tra moduli asincroni, si nota che anche questo tipo di coordinatore soffre di tale inconveniente. Nella fattispecie durante una fase di controllo in cui un *behavior* a priorità superiore desideri trasmettere costantemente informazioni, può accadere che ogni tanto il coordinatore non si trovi il messaggio proveniente da tale *behavior* e nello stesso tempo abbia invece a disposizione un messaggio proveniente da un *behavior* a priorità inferiore che riesce così a “scavalcare” il primo compromettendo il comportamento assunto dal robot durante la fase di controllo in corso. Si decide di risolvere questo problema

sfruttando la stessa tecnica usata nell'architettura a sussunzione (vedi sezione 2.4, nota 22) permettendo perciò ad un messaggio proveniente da un *behavior* a priorità alta di mantenere il controllo sull'output del coordinatore, e quindi prevalere su tutti i messaggi provenienti dai *behavior* con priorità inferiore, per un tempo pari a due volte il *control step* cosicché se un *behavior* continua a trasmettere informazioni con la massima frequenza consentita nessun *behavior* a priorità inferiore riuscirà mai ad intromettersi.

Si decide inoltre di fornire in uscita l'identificativo del *behavior* che ha vinto la contesa: l'identificativo è in pratica la posizione nell'array degli input a cui è connesso il suddetto *behavior*. Questo output può essere usato successivamente da chiunque fosse interessato a conoscere tale informazione.

ActivationCoord

Questo coordinatore riproduce la tecnica di coordinazione *action-selection* in cui ogni messaggio è affiancato ad un determinato livello di attivazione (modellato come un valore numerico in virgola mobile maggiore o uguale a zero), il tutto incapsulato in un'istanza della classe `MsgAct<T>`, che costituisce di fatto una coppia del tipo `<messaggio, livello di attivazione>`. Il coordinatore ad ogni iterazione riporta in uscita il messaggio che presenta il livello di attivazione maggiore: nel caso in cui tutti i messaggi presentano un livello di attivazione nullo (uguale a zero) nessun messaggio viene trasmesso. In questo caso i messaggi non vengono mai rimossi dai registri di input, dunque se un *behavior* non desidera trasmettere alcuna informazione è sufficiente che segnali (anche solo una volta) un livello di attivazione nullo.

In uscita il coordinatore riporta quattro tipi di informazioni che sono:

- l'identificativo del *behavior* vincitore (come avviene per il `PriorityCoord`);
- il messaggio originale corredato da livello di attivazione – tipo: `MsgAct<T>;`
- il messaggio originale senza il livello di attivazione – tipo: `T`;
- il solo livello di attivazione – tipo: `Float`.

Questi output possono essere usati liberamente e trasmessi, tramite connessioni, agli input di altri *behavior* dando piena libertà di sviluppo.

FusionCoord

Questo è l'unico coordinatore di tipo cooperativo disponibile e costituisce in realtà solamente la struttura base di un coordinatore. Si tratta infatti di una classe astratta che l'utente deve estendere implementandone l'operazione `fuse` che viene invocata ad ogni iterazione per generare l'output in funzione degli input correnti. Gli input e gli output forniti da un coordinatore di questo tipo sono gli stessi del coordinatore `ActivationCoord` e quindi ad ogni informazione, sia a quelle trasmesse agli input sia all'output generato dalla funzione `fuse`, è correlato un livello di attivazione.

3.4 Linguaggio di specifica e IDE Xtext

Come si è visto il framework è costituito da una libreria di classi *Java* che l'utente può sfruttare per implementare i *behavior* del sistema di controllo e per strutturare il sistema stesso. Risulta però relativamente tedioso svolgere "a mano" alcune operazioni quali impostare la struttura di un *behavior* in termini di porte di input e di output e configurare le connessioni tra i vari moduli. Si decide pertanto di accompagnare il framework con un linguaggio *domain specific* di alto livello ed un IDE sviluppati utilizzando la tecnologia *Xtext* [48]. *Xtext* è uno strumento introdotto nel contesto dell'*Eclipse Modelling Framework* per supportare lo sviluppo agile di linguaggi di programmazione e linguaggi *domain specific* ed è distribuito sotto forma di plug-in integrato nell'ambiente di sviluppo *Eclipse*. Ciò che lo caratterizza è il fatto che, a partire dalla definizione delle regole grammaticali del linguaggio d'interesse, si preoccupa di generare automaticamente l'insieme delle entità che costituiscono il modello su cui tale linguaggio si appoggia, costruire un parser in grado di interpretare una specifica nel dato linguaggio e trasformarla nel corrispondente *Abstract Syntax Tree* composto dalla entità del modello appena citato, ed infine produrre un editor guidato dalla sintassi integrato anch'esso in *Eclipse* come plug-in.

Il linguaggio introdotto come supporto per questo framework viene predisposto alla definizione della sola parte strutturale di un sistema di controllo *behavior-based*: permette infatti di specificare i tipi di *behavior* presenti in termini dell'interfaccia che espongono (porte di input e di output), le istanze dei *behavior* che costituiscono la rete del sistema di controllo, ed infine le connessioni esistenti tra le loro porte di comunicazione. Insieme al linguaggio viene ovviamente sviluppato un compilatore, come parte integrante dell'IDE generato da *Xtext*, che è in grado di interpretare l'*Abstract Syntax Tree* risultante dalla fase di parsing e utilizzarlo per generare un insieme di classi di base da cui l'utente può partire per uno sviluppo più agevole del suo specifico controllore.

Qui di seguito è riportata una versione leggermente semplificata della grammatica formale del linguaggio. Si nota subito come un controllore (regola `Con-`

troller) sia espresso in termini di definizioni di *behavior*, istanze di *behavior* e connessioni tra quest'ultime.

```
Controller:
  'package' FullID ';'
  'controller' ID ';'
  (Declaration)*
;

Declaration:
  BehaviorDef | Instance | Connection
;

BehaviorDef:
  'behavior' ID (JAVA_GENERIC_TYPES)?
                                ('(' (JavaParam)? (',' JavaParam)* ')')? '{'
    ('input' '{' (InputPortDef ';')* '}')?
    ('output' '{' (PortDef ';')* '}')?
  '}'
;

JavaParam:
  JavaType ID
;

InputPortDef:
  InfluenceDef | PortDef (':' 'buffer-size' '=' JavaIntExpr)?
;

InfluenceDef:
  'influence' ID ('[' JavaIntExpr ']')? ':' 'min' '=' JavaFloatExpr ','
                        'max' '=' JavaFloatExpr ',' 'dec' '=' JavaFloatExpr
;

PortDef:
  JavaType ID ('[' JavaIntExpr ']')?
;

Instance:
  ID (JAVA_GENERIC_TYPES)? ID ';'
;

Connection:
  'connect' PortRef 'to' PortRef (',' PortRef)* ';'
;

PortRef:
  [Instance] '.' ID ('[' JavaIntExpr ']')?
;

FullID returns ecore::EString: ID ('.' ID)*;
Float returns ecore::EFloat: ('-')? INT ('.' INT)? (('e'|'E') ('-')? INT)?;
JavaType returns ecore::EString: FullID ('[' ']')*;
JavaIntExpr returns ecore::EString: INT | JavaVarAccess;
JavaFloatExpr returns ecore::EString: Float | JavaVarAccess;
JavaVarAccess returns ecore::EString: FullID ('[' JavaIntExpr ']')*;
terminal JAVA_GENERIC_TYPES returns ecore::EString: '<' -> '>';
```

Un *behavior* (regola `BehaviorDef`) viene definito elencandone le porte di input e di output. Per le porte di input è possibile scegliere se usare un `InputRegister`, adottato di default, oppure se impiegare un `InputBuffer` con una determinata dimensione specificata con la keyword `buffer-size`. Tra le porte di input è presente anche il caso particolare di porta di tipo `InputInfluence` che deve però essere trattato individualmente in quanto differisce molto dagli altri (regola `InfluenceDef`). Le porte di output sono invece definite semplicemente specificando il tipo di dato relativo. Si noti che il linguaggio supporta anche la definizione di array di porte di comunicazione. Sia la dimensione di questi array che quella degli eventuali `InputBuffer` utilizzati sono trattate dal linguaggio come espressioni numeriche molto semplici che possono essere costituite da un valore numerico diretto o dall'accesso ad una variabile. A tal scopo si permette di definire un insieme di parametri a livello di *behavior* che possono essere poi sfruttati a tempo di esecuzione per la valutazione di queste espressioni: ad esempio si può definire un tipo di *behavior* che presenti un array di porte la cui effettiva dimensione viene determinata solo nel momento della creazione di un'istanza del *behavior* tramite un parametro.

Le istanze dei *behavior* che compongono la rete sono definite semplicemente indicando il tipo di *behavior* di appartenenza ed è ovviamente possibile dichiarare un numero arbitrario di istanze di uno stesso tipo di *behavior*[38]. Si fa notare che è possibile dichiarare anche istanze di *behavior* che non siano definiti nel contesto del controllore in esame, al contrario, si può fare riferimento a qualsiasi tipo di modulo definito esternamente come, per esempio, ai coordinatori standard forniti con il framework.

Infine le connessioni tra i *behavior* sono specificate semplicemente indicando una porta sorgente (di output) e una serie di una o più porte di destinazione (di input). Ogni porta è referenziata con una notazione puntata composta dal nome dell'istanza a cui appartiene seguito da quello della porta stessa.

38 Non è infatti insolito progettare sistemi di controllo in cui vi siano più istanze di una stessa tipologia di *behavior*. Si faccia riferimento per esempio al sistema di controllo di *Genghis* esposto in figura 26.

Tutte le entità in gioco, eccetto le connessioni, sono identificate tramite un nome simbolico che può logicamente essere usato per farvi riferimento.

Si noti come le regole grammaticali per la dichiarazione dei *behavior* e delle istanze prevedano la possibilità di usare tipi parametrici per creare *behavior* particolarmente flessibili: ad esempio un *behavior* può possedere delle porte di input e output associate ad un tipo di dato generico che sarà poi determinato solamente al momento della dichiarazione delle specifiche istanze (questo è in definitiva il principio base su cui si fondano tutti i coordinatori visti prima).

Il compilatore del linguaggio viene sviluppato usando la tecnologia *Xtend* come suggerito da *Xtext* stesso. Si tratta di un generatore che analizza il risultato della fase di parsing e crea una classe *Java* di supporto per ogni *behavior* definito nel sorgente e una per il sistema complessivo. Per distinguere queste classi da quelle specificate dall'utente al loro nome viene aggiunto il suffisso "_stub". Il compito dell'utente sarà poi quello di completare lo sviluppo del sistema implementando i metodi `step` delle classi relative ai *behavior* generate dall'IDE e rifinire la fase di inizializzazione del controllore con la creazione delle istanze dei *behavior* in gioco.

Per ogni *behavior* viene generata una classe di supporto che si occupa di dichiarare ed istanziare le porte di input e di output e di renderle accessibili dall'esterno tramite dei metodi selettori aventi come nome quello della porta corrispondente a cui è semplicemente aggiunto il prefisso "get_" (in caso di array di porte di comunicazione il metodo richiede come parametro l'indice della porta d'interesse). Insieme alla classe di supporto appena descritta viene generata anche la bozza di una classe che estende quest'ultima e che deve essere personalizzata dall'utente implementando il noto metodo `step`.

Per il sistema di controllo nel suo complesso si genera una classe di supporto all'interno della quale sono dichiarate le istanze dei *behavior* presenti ed è disponibile un metodo denominato `connectAndRun` che si preoccupa di stabilire le connessioni tra i *behavior* e di mettere in esecuzione lo *scheduler*. Anche in questo caso viene generata una ulteriore bozza di classe che estende questa e

predispone l'*entry point* del controllore (la funzione `main`) e una operazione incompleta `setupAndRun` che suggerisce all'utente i passi da implementare per la fase di inizializzazione del sistema, tra i quali la creazione delle istanze dei *behavior* e, da ultimo, l'invocazione del metodo `connectAndRun`.

Qui sotto viene riproposto l'esempio di sistema di controllo già mostrato prima e rappresentato in figura 34; in questo caso però la struttura della rete è descritta mediante l'uso del linguaggio appena introdotto:

```
package example;
controller ExampleController;

behavior B1 {
  output {
    JavaType1 out1;
    //...
  }
}

behavior B2 {
  input {
    JavaType1 in1;
    //...
  }
  output {
    JavaType1 out1;
    //...
  }
}

behavior B3<JavaFormalGenericType>(int n) {
  input {
    JavaType1 in[n];
    //...
  }
  output {
    JavaFormalGenericType out1;
    //...
  }
}

behavior B4<JavaFormalGenericType> {
  input {
    JavaFormalGenericType in1;
    //...
  }
}

B1 b1;
B2 b2;
B3<JavaActualGenericType> b3;
B4<JavaActualGenericType> b4;
```

```
connect b1.out1 to b2.in1, b3.in[0];
connect b2.out1 to b3.in[1];
connect b3.out1 to b4.in1;
```

Per concludere si avvisa che questo semplice linguaggio non è stato studiato approfonditamente, al contrario, è stato introdotto con il solo scopo di velocizzare la parte di progettazione della struttura del sistema di controllo e non costituisce in alcun modo un linguaggio di programmazione di alto livello che possa essere usato per implementare tali sistemi. L'IDE infatti si preoccupa solamente della correttezza sintattica e della trasformazione in codice *Java* evitando completamente di effettuare controlli di tipo sulle entità in gioco tanto meno di verificarne la correttezza semantica. Dalle regole grammaticali si può infatti notare che molti degli oggetti del linguaggio (come espressioni numeriche, indicazioni di tipi generici, riferimenti a porte di comunicazione, etc.) sono trattati dal generatore addirittura come semplici stringe che vengono ricopiate alla lettera nei file sorgenti *Java* generati. Sarà poi delegato al compilatore *Java* l'incarico di verificare che non vi siano certi errori, ad esempio nell'uso dei tipi di dato[39].

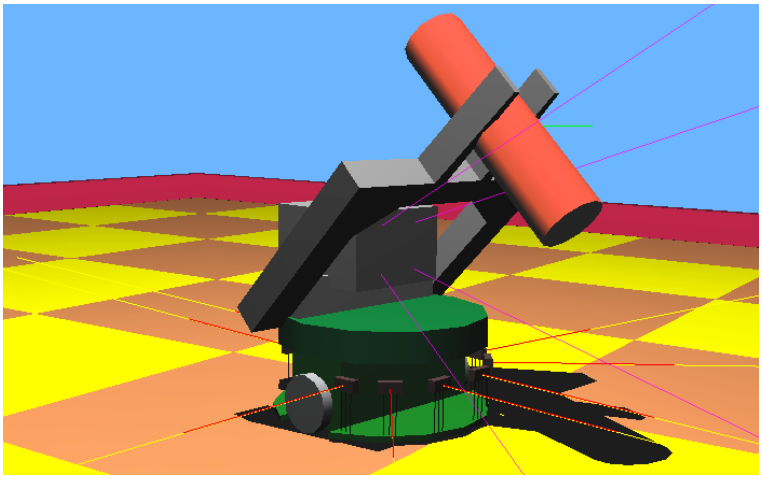
39 Il compilatore *Java* si renderà conto di errori quali: il tentativo di connettere porte di comunicazione non compatibili fra loro (in quanto si verificherà un errore di tipo), il tentativo di accedere da una porta inesistente (in quanto non esisterà il relativo metodo selettore), l'inesistenza di eventuali tipi di dato indicati nella specifica, potenziali incorrettezze nell'uso del polimorfismo parametrico, e così via.

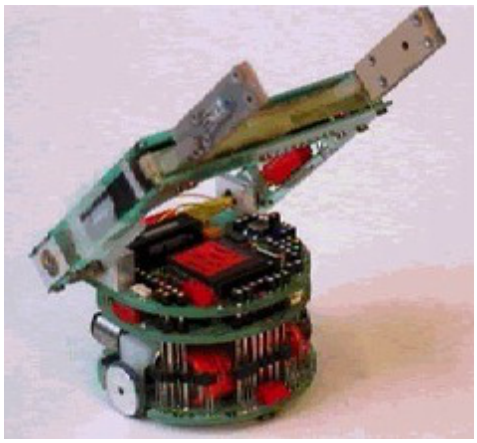
Capitolo 4
Un caso di studio

Per valutare il framework appena sviluppato, e più in generale l'approccio di controllo *behavior-based*, si decide di risolvere un caso di studio molto frequente in letteratura indicato con il nome di *foraging* (traducibile in italiano con "ricerca di cibo"). Questo task prevede di riprodurre un comportamento tipico degli animali che in natura si presenta quotidianamente: si tratta infatti della ricerca e raccolta di risorse alimentari all'interno di un determinato territorio. Nello specifico il task prevede che un agente (o un insieme di agenti) esplori questo territorio nella speranza di trovare degli oggetti "attrattori" che deve raccogliere e radunare in una regione "goal". Durante lo spostamento sul territorio ovviamente deve anche evitare di avvicinarsi troppo o addirittura di scontrarsi con entità "repulsive" quali ad esempio ostacoli invalicabili.

Si utilizza *Webots* per effettuare la simulazione e modellare l'arena in cui questa avrà luogo. Gli oggetti da raccogliere sono rappresentati da piccoli solidi di colore rosso vivo con forme e dimensioni leggermente diverse tra loro. Il robot impiegato è il *Khepera I* prodotto da *K-Team* [34] (al momento alla versione *II* e *III*) a cui è aggiunta una piccola telecamera ed un particolare modulo, chiamato *gripper*, composto da vari attuatori che riproducono il funzionamento di una pinza meccanica con cui il robot è in grado di afferrare gli oggetti. Si tratta di un piccolo robot mobile, di pochi centimetri di diametro, con guida differenziale, che quindi possiede due motori indipendenti ai quali sono connesse le due ruote. Oltre ai motori, alla telecamera e alla pinza il robot è fornito di otto sensori di distanza ad infrarossi attivi, otto sensori di luminosità, un sensore per il livello di carica della batteria, un *encoder* su entrambe le ruote e sensori di posizione e di forza (*force feedback*) per ognuno degli attuatori. La disposi-

zione dei sensori è mostrata in figura 36. Ai fini di questa simulazione si aggiungono al robot un dispositivo GPS e una bussola digitale che gli permetterà di sfruttare l'enorme vantaggio di conoscere con buona certezza la sua posa assoluta composta da posizione ed orientamento in relazione al sistema di riferimento dello scenario. Nel caso in cui il robot dovesse in realtà operare in sistemi indoor sono comunque possibili altre forme di tecnologie di posizionamento. Se anche questo non si verificasse resterebbe tuttavia possibile ripiegare su tecniche di odometria che tentano di risalire alla posa del robot (in relazione ad un certo sistema di riferimento fisso) tenendo costantemente sotto controllo (per mezzo di un'operazione di integrazione matematica) i valori forniti dagli *encoder* delle due ruote [26]: questa tecnica è però sconveniente perché soffre terribilmente delle imperfezioni del mondo reale e richiede di effettuare, in qualche modo, frequenti ricalibrazioni.





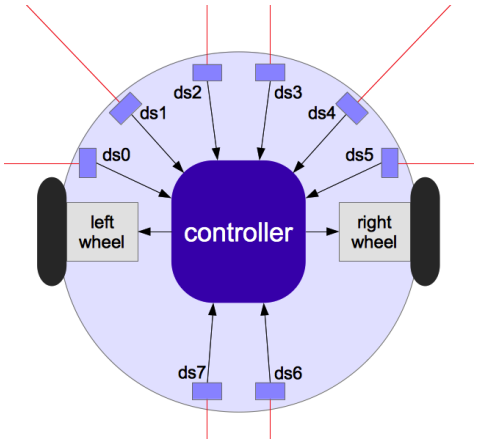


Figura 36: *Khepera I* simulato e reale – le linee rosse rappresentano i sensori di prossimità, le linee gialle i sensori di luminosità e le linee viola delimitano il frustum della telecamera.

Nell'arena in cui si svolge la simulazione, oltre al robot e agli oggetti che questo deve raccogliere, sono disposti anche degli ostacoli che il robot deve saper evitare e una particolare entità che funge da caricatore. Spiegato in poche parole un caricatore è semplicemente una regione limitata dello spazio all'interno della quale il robot può ricevere energia che gli permetta di "sopravvivere" ricaricandone le batterie. Viene infine aggiunta una forma circolare semitrasparente che indica la regione goal in cui devono essere rilasciati gli oggetti. Nella figura che segue è riportata un'istantanea scattata durante una simulazione: si può vedere il robot all'opera mentre si dirige verso la regione goal, indicata in colore azzurro, per rilasciare l'oggetto che ha appena recuperato; infine la forma semitrasparente di colore verde indica la posizione e il raggio d'azione del caricatore.

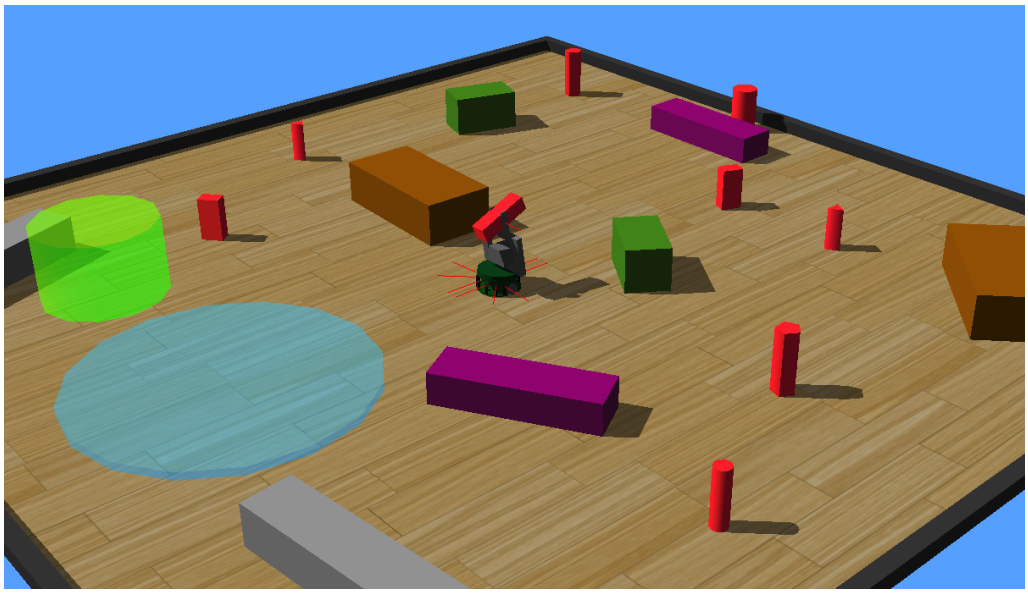


Figura 37: Arena virtuale in cui si svolge la simulazione.

Seguendo il processo di sviluppo bottom-up introdotto con il paradigma *behavior-based* e le linee guida elencate nella sezione 2.7, si procede gradualmente alla soluzione di questo caso di studio. Si decide infatti di sviluppare tre sistemi di controllo a complessità crescente, denominati in successione `CtrlWander`, `CtrlWanderRecharge` e `CtrlWanderRechargeForage`. Come è facile intuire dai nomi stessi dei controllori al passaggio da un sistema all'altro vengono semplicemente aggiunte nuove competenze:

- il primo controllore permette al robot di girovagare casualmente nell'arena evitando di scontrarsi con gli ostacoli che incontra;
- il secondo aggiunge un livello di competenza che consente al robot di monitorare la carica della batteria e raggiungere il caricatore prima che sia troppo tardi;
- infine l'ultimo completa la soluzione implementando il vero comportamento desiderato secondo il quale il robot deve riconoscere gli oggetti d'interesse, raccoglierli, ed infine rilasciarli nella regione goal.

4.1 FSM

Prima di proseguire con la progettazione dei controllori ci si premunisce di una classe di utilità che permetta di implementare agevolmente una macchina a stati finiti (forse poco convenzionale) in quanto spesso i *behavior* di un controllore *behavior-based* sono sviluppati proprio secondo tale modello.

La classe prende il nome di `FSM` ed è in grado di prendere in carica un insieme di stati i quali sono istanze della classe astratta `FSM.State`. L'esecuzione avviene per mezzo del metodo `exec` fornito da `FSM` che si rivolge a sua volta allo stato corrente. Ad ogni stato sono associate tre operazioni invocate dall'automa in determinate situazioni:

- `entry` – quando l'automa entra in questo specifico stato;
- `exec` – durante l'esecuzione dell'automa se questo è lo stato corrente;
- `exit` – quando l'automa esce da questo specifico stato.

L'idea è che l'utente di questa classe crei un automa associato ad una serie di stati; fatto ciò per far progredire la computazione dell'automa deve semplicemente invocarne il metodo `exec` regolarmente. Le transizioni da uno stato all'altro sono gestite direttamente dagli stati stessi i quali hanno a disposizione un metodo protetto `transition` che permette di specificare lo stato successivo in cui l'automa deve portarsi. In questo caso l'idea è che lo stato corrente verifichi in continuazione le dovute condizioni di transizione, all'interno della propria operazione `exec`, e richieda di effettuare le relative transizioni[40].

Per concludere, la classe `FSM` fornisce all'esterno anche un'operazione di `reset` che può essere invocata per riportare istantaneamente l'automa al suo stato iniziale.

40 Data la natura dell'operazione, una richiesta di transizione deve essere l'ultima azione compita da uno stato prima di terminare il corpo del metodo `exec`.

4.2 Wander

Il primo controllore richiede subito di mettere in campo i due comportamenti "spostarsi casualmente" ed "evitare gli ostacoli" che possono certamente risultare contrastanti, soprattutto in presenza di ostacoli nelle vicinanze. Lo schema riportato sotto raffigura come si desidera affrontare questo problema. Data la vitale importanza del secondo comportamento si opta per un approccio competitivo che prevede di dare massima priorità proprio a questo. Si decide inoltre di non effettuare una decomposizione troppo fine dei *behavior* e pertanto si mettono in campo solamente due *behavior* `Wander` e `Avoid`, un coordinatore di tipo `PriorityCoord<Drive.Speed>` e un modulo `Drive`.

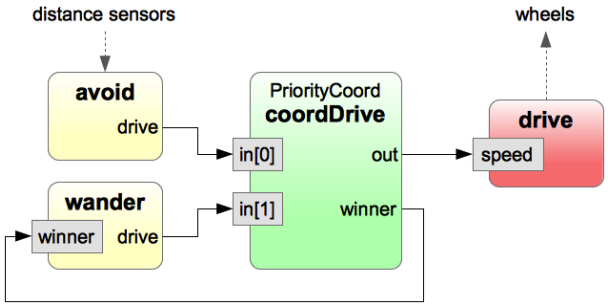


Figura 38: Schema del controllore `CtrlWander`.

Drive

Il primo modulo ad essere sviluppato è il *behavior* `Drive` che funge da "attuatore virtuale" e si prende carico di gestire i motori associati alle due ruote del robot. Chiunque desideri assegnare una certa velocità alle due ruote è tenuto a comunicare con questo modulo. Il valore della velocità delle ruote è indicato per mezzo una struttura dati `Speed` che contiene le due velocità, ciascuna delle quali è codificata come numero in virgola mobile nell'intervallo [-1,+1] con +1 che rappresenta la massima velocità della ruota in avanti, e -1 che invece indica massima velocità in senso contrario. Se ad esempio viene imposta una velocità (1,1) il robot si sposta in avanti alla massima velocità, con (0,0) resta fermo, con (-1,-1) indietreggia alla velocità massima, mentre ogni combinazione in cui

le due velocità sono diverse causano una rotazione del robot verso la direzione a cui è associata la velocità inferiore (considerando il segno)[41].

```
...
public class Drive extends foraging_stub.Drive_stub {
  static public class Speed {
    public final float left, right;
    public Speed(float left, float right) {
      this.left = Misc.clipRange(left, -1, 1);
      this.right = Misc.clipRange(right, -1, 1);
    }
  }
  ...
  public void step() {
    Speed s = _speed.get();
    if (s != null) {
      robot.setSpeed(s.left*maxSpeed, s.right*maxSpeed);
    }
  }
}
```

Wander

Il secondo *behavior* ad essere implementato è `Wander` che si occupa di "spostarsi casualmente" ed è costituito da un semplice automa con due stati: `Forward` e `Turn`. Nello stato `Forward` si limita a segnalare l'intenzione di spostare il robot in avanti dopo di che resta in attesa che sia trascorso un lasso di tempo casuale per poi passare allo stato `Turn`. In quest'ultimo invece richiede una rotazione in una direzione casuale e resta in attesa per un breve periodo di tempo per poi tornare allo stato `Forward`. Infine tramite l'output `winner` generato dal coordinatore è in grado di verificare se è stato soppresso da qualche altro *behavior*: in tal caso si riporta istantaneamente nello stato iniziale in modo che non appena gli venga affidato il controllo del robot questo inizi subito a spostarsi in avanti.

```
...
public class Wander extends foraging_stub.Wander_stub {
  ...
  public void step() {
    Integer w = _winner.peek();
    if (w != null  &&  w != id) {  // the behavior has been suppressed
      fsm.reset();
    } else {
      fsm.exec();
    }
```

41 Ulteriori dettagli sulla guida differenziale possono essere trovati in [26].

```
  }

  private final FSM.State FORWARD = new FSM.State() {
    private int wait;
    protected void entry() {
      wait = Misc.steps(100*Misc.rndRange(10, 51));  // [1,5] seconds
      _drive.send(new Drive.Speed(1, 1));
    }
    protected void exec() {
      wait--;
      if (wait <= 0) {
        transition(TURN);
      } else {
        _drive.send(new Drive.Speed(1, 1));
      }
    }
  };

  private final FSM.State TURN = new FSM.State() {
    private boolean left;
    private int wait;
    protected void entry() {
      wait = Misc.steps(100*Misc.rndRange(3, 16));  // [0.3,1.5] seconds
      left = Misc.RANDOM.nextBoolean();
      _drive.send(new Drive.Speed(left ? 0 : 1, left ? 1 : 0));
    }
    protected void exec() {
      wait--;
      if (wait <= 0) {
        transition(FORWARD);
      } else {
        _drive.send(new Drive.Speed(left ? 0 : 1, left ? 1 : 0));
      }
    }
  };
}
```

Avoid

Infine si sviluppa il terzo *behavior*, il più complesso per il momento, che garantisce al robot la capacità di "evitare gli ostacoli". Il *behavior* `Avoid` è implementato molto semplicemente come un automa con quattro stati: `Idle`, `Turn`, `Avoid` ed `Escape`. In ciascuno di questi stati vengono verificate certe condizioni a fronte delle quali si transita in un determinato stato futuro. Le condizioni sono espresse in funzione dei valori forniti dai sensori di distanza (si consiglia di prendere visione della disposizione dei sensori esposta in figura 36). Questi sono numerati progressivamente da `ds0` a `ds7` e ad ogni iterazione le letture che restituiscono vengono leggermente elaborate per filtrare valori fuori norma e per riportarle nell'intervallo [0,1] in cui l'estremo 0 indica che non si

rileva nessun ostacolo da quel particolare sensore, mentre 1 indica un ostacolo estremamente vicino al robot; tutti i valori intermedi indicano, con continuità, un ostacolo più o meno vicino. Le condizioni appena citate sono valutate in ordine di priorità e considerano tre situazioni:

1. `(ds2+ds3 > 0)` – significa che è presente un ostacolo di fronte al robot;
2. `(ds0+ds1+ds4+ds5 > 0)` – significa che sono presenti ostacoli sui lati sinistro e/o destro del robot;
3. `(ds6+ds7 > 0)` – significa che è presente un ostacolo dietro al robot.

Per questo *behavior* si preferisce non riportare direttamene il codice in quanto si ritiene più chiaro il diagramma mostrato qui sotto.

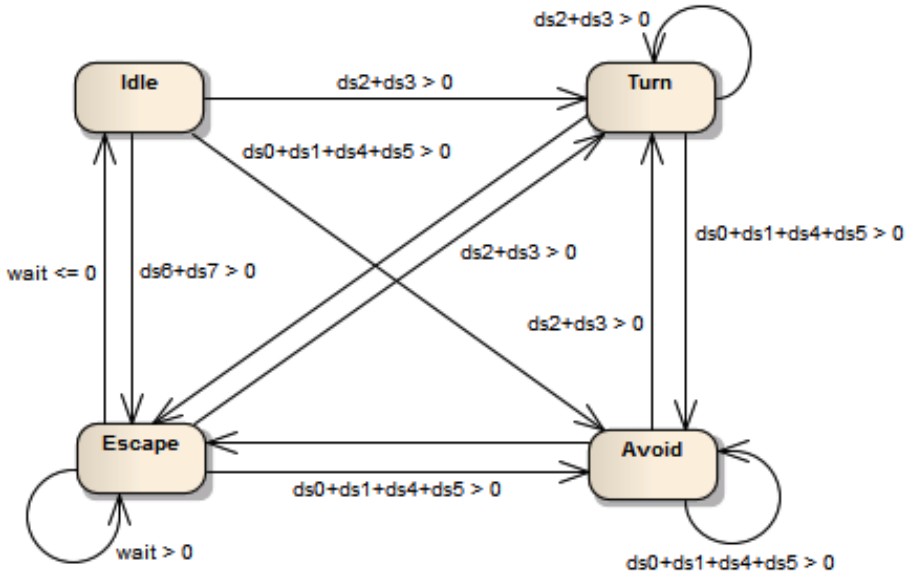


Figura 39: Diagramma degli stati del *behavior* `Avoid`.

Il compito dello stato `Idle` è quello di valutare queste condizioni in ordine di priorità e transitare nello stato opportuno non appena se ne verifica una.

```
private final FSM.State IDLE = new FSM.State() {
  protected void exec() {
    if (ds2+ds3 > 0) {  // obstacle ahead -> turn
      transition(TURN);
    } else if (ds0+ds1+ds4+ds5 > 0) {  // obstacles on the sides -> avoid
      transition(AVOID);
    } else if (ds6+ds7 > 0) {  // obstacle behind -> escape
      transition(ESCAPE);
    }
  }
```

```
};
```

Una volta transitati nello stato `Turn` viene individuato il lato del robot meno ostruito e si prosegue facendo ruotare il robot su sé stesso proprio in questa direzione. In realtà, dopo aver effettuato alcuni esperimenti, si decide di scegliere casualmente se ruotare dal lato meno ostruito o dal lato opposto, dando ovviamente più probabilità al primo caso. Questo perché se il robot ripete sempre le stesse scelte c'è il rischio che cada in una situazione di stallo in cui continui a intraprendere ripetutamente le stesse azioni con pochissima probabilità di uscirne. L'opportunità di cambiare strategia ogni tanto fornisce invece qualche possibilità in più di liberarsi da queste situazioni.

```
...
left = (ds0 + ds1 + ds2 < ds3 + ds4 + ds5);
if (Misc.happens(0.3f)) {
  left = !left;
}
...
if (ds2 + ds3 > 0) {
  _drive.send(new Drive.Speed(left ? -1 : +1, left ? +1 : -1));
}
...
```

Una volta presa la decisione sul lato verso il quale girarsi, il robot continua a ruotare nella stessa direzione finché i sensori anteriori non rilevano più ostacoli; a questo punto si transita nello stato `Avoid` o nello stato `Escape` a seconda dei valori forniti dai restanti sensori.

Lo stato `Avoid` è incaricato di evitare gli ostacoli che si trovino sui lati sinistro e destro del robot. Nel tentativo di generare dei movimenti fluidi si decide di adottare lo stesso approccio dei veicoli di Braitenberg (visti nella sezione 2.1). In questo modo il valore fornito dai sensori influisce direttamente sulla velocità delle ruote generando una gamma continua di risposte, da lievi deviazioni a fronte di ostacoli lontani, a brusche sterzate a fronte di ostacoli vicini. I coefficienti utilizzati sono stati valutati sperimentalmente. Si nota inoltre che questo algoritmo permette al robot anche di seguire i corridoi in quanto se ven-

gono rilevati ostacoli su entrambi i lati del robot alle due ruote viene impartita una velocità positiva portando quindi il robot a muoversi in avanti[42].

```
_drive.send(new Drive.Speed(1 + 2*ds0 + 4*ds1 - 4*ds4 - 2*ds5,
                            1 - 2*ds0 - 4*ds1 + 4*ds4 + 2*ds5));
```

Infine lo stato `Escape` è lo stato finale in cui convergono tutte le manovre di fuga dagli ostacoli. In questo stato il robot si limita a proseguire in avanti per un breve periodo di tempo nella speranza di allontanarsi dall'ostacolo appena evitato. Ovviamente anche durante questa operazione le condizioni di cui sopra sono monitorate costantemente.

42 È chiaro che questo può essere anche uno svantaggio in quanto il robot è "incuriosito" dai corridoi particolarmente stretti e di conseguenza corre il rischio di rimanervi incastrato.

4.3 Wander & Recharge

Il secondo controllore aggiunge al primo un ulteriore livello di competenza: perché il robot sia in grado di sopravvivere autonomamente per lunghi periodi di tempo deve provvedere a monitorare il livello delle batterie e a ritornare alla stazione di ricarica prima di esaurire completamente l'energia a disposizione.

In questo caso si preferisce decomporre il problema nei suoi comportamenti elementari. Il robot deve infatti saper svolgere due nuove operazioni: monitorare il livello della batteria per reagire di conseguenza ed essere in grado di raggiungere una posizione prestabilita. Questa seconda attività deve certamente essere prioritaria rispetto agli spostamenti casuali suggeriti dal *behavior* `Wander` ma nello stesso tempo non deve alterare il comportamento indotto da `Avoid`. Perciò si decide banalmente di darvi una priorità intermedia tra i due.

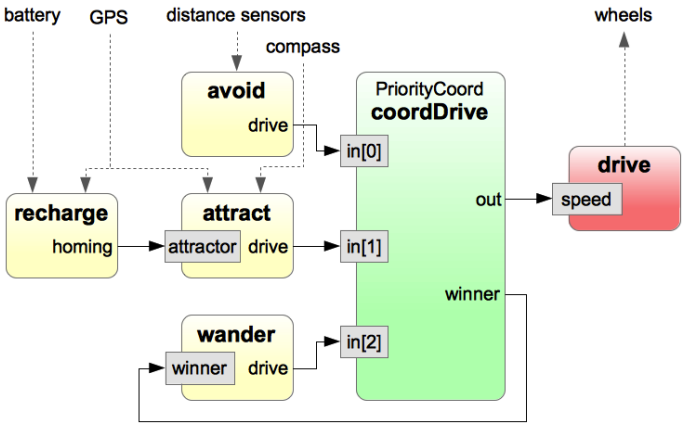


Figura 40: Schema del controllore `CtrlWanderRecharge`.

Attract

Per fornire la capacità di raggiungere una certa destinazione si predispone un *behavior* `Attract` che, sfruttando le informazioni di posizione ed orientamento rilevate dal dispositivo GPS e dalla bussola digitale, fornisce in output i comandi da impartire alle ruote del robot affinché questo si sposti in direzione di un "attrattore" e lì si fermi una volta raggiunto. La posizione dell'attrattore viene fornito come input al *behavior* in termini di una istanza di tipo `Location`.

```
public class Location {
  public final float x, z;
```

```
  public Location(float x, float z) {
    this.x = x;
    this.z = z;
  }
}
```

Per il momento si decide che al *behavior* debba essere costantemente fornita in ingresso una posizione verso cui attirare il robot. Gli input, finché disponibili, vengono infatti consumati e, a partire dal momento in cui non ve ne sono più, il *behavior* smette di fornire degli output. I *behavior* che volessero imporre il comportamento di questo modulo sono dunque tenuti ad inviare continuamente la posizione dell'attrattore desiderato. Come al solito una situazione del genere presenta l'inconveniente già discusso per il `PriorityCoord`, dovuto al continuo scambio di messaggi tra *behavior* paralleli. Come fatto in quel caso, il problema può essere risolto allungando la validità temporale di un messaggio ad un periodo pari a due volte il *control step*; ovviamente questo solo se nel frattempo non ne vengono ricevuti altri più recenti.

```
...
public class Attract extends foraging_stub.Attract_stub {
  ...
  public void step() {
    Location loc = _attractor.get();
    if (loc != null) {
      target = loc;  // consider the new target location
    }
    if (target != null) {
      // new or previous location (from the last control step)
      // try to reach the target location
      double[] pos = gps.getValues();
      double[] north = compass.getValues();
      float heading = (float)Math.atan2(north[0], north[2]);
      float dx = (float)pos[0] - target.x;
      float dz = (float)pos[2] - target.z;
      if ((dx*dx + dz*dz) < EPSILON*EPSILON) {
        // (almost) at the target location -> stop
        _drive.send(new Drive.Speed(0, 0));
      } else {  // move toward the target location
        float direction = Misc.stdAngle((float)Math.atan2(dz, dx)-heading);
        if (Math.abs(direction) > (Math.PI-0.01)) {
      // the target is (almost) behind -> always turn in the same direction
          _drive.send(new Drive.Speed(1, -1));
        } else {
          _drive.send(new Drive.Speed(
                  (direction >= 0 ? 1 : (1 + 2*direction/( float)Math.PI)),
                  (direction <= 0 ? 1 : (1 - 2*direction( float)Math.PI))));
        }
      }
    }
```

```
    if (loc == null) {  // don't consider the old target any more
      target = null;
    }
  }
}
```

Recharge

A questo punto il primo comportamento può essere ottenuto con una certa facilità. Il *behavior* `Recharge` viene impostato come un automa a due stati: `Idle` e `Recharge`.

Nello stato `Idle` semplicemente ci si limita a valutare il livello della batteria e se questo dovesse scendere sotto una certa soglia si procede transitando nello stato `Recharge`. Vista la disponibilità di un dispositivo GPS si opta per calcolare questa soglia in funzione della distanza relativa tra il robot e il caricatore. Ovviamente tanto maggiore è questa distanza tanto urgente diventa il bisogno di tornare alla zona di ricarica. Questo è banalmente dovuto al fatto che sulla strada di ritorno il robot continua a consumare energia e, per di più, maggiore è la distanza da percorrere maggiore saranno le probabilità di incontrare ostacoli lungo il cammino col rischio di rallentarlo fatalmente. Nello specifico si usa una soglia che varia linearmente con la distanza. Dopo aver impostato la simulazione di consumo energetico del robot e fatto alcuni esperimenti, si decide di prendere come punto fisso la distanza di circa un metro a cui è associa una soglia del 50% e di limitare inferiormente questo valore ad un minimo del 30% per motivi di sicurezza.

Il compito dello stato `Recharge` è invece quello di influenzare il *behavior* `Attract` affinché il robot si porti nella posizione del caricatore e restare in attesa che il livello delle batteria risalga fino circa al valore massimo.

```
...
public class Recharge extends foraging_stub.Recharge_stub {
  ...
  public void step() {
    // read sensor data
    battery = (float)robot.batterySensorGetValue();
    fsm.exec();
  }

  private final FSM.State IDLE = new FSM.State() {
    protected void exec() {
```

```
      double[] pos = gps.getValues();
      float x = (float)pos[0];
      float z = (float)pos[2];
      if (battery < Math.max(30,
                50*Misc.dist(x, z, Misc.CHARGER.x, Misc.CHARGER.z)/1.1f)) {
        transition(RECHARGE);
      }
    }
  };

  private final FSM.State RECHARGE = new FSM.State() {
    protected void entry() {
      _homing.send(Misc.CHARGER);
    }
    protected void exec() {
      if (battery > 99.5f) {  // resume normal behavior
        transition(IDLE);
      } else {  // returning to charger or recharging
        _homing.send(Misc.CHARGER);
      }
    }
  };
}
```

Questo *beahvior* così come è sviluppato si appoggia al funzionamento del caricatore di *Webots* che provvede autonomamente a ricaricare il robot il quale deve semplicemente trovarsi nella giusta posizione. In un caso reale è plausibile che il robot debba invece eseguire una serie di azioni di routine una volta raggiunta la posizione del caricatore: ad esempio rallentare e orientarsi in modo da avvicinarsi al caricatore da una certa direzione, o magari usare determinati attuatori per connettersi allo stesso. É ovvio che in situazioni del genere il *behavior* appena sviluppato dovrebbe essere esteso notevolmente.

4.4 Wander & Recharge & Forage

L'ultimo controllore sviluppato è quello che mette effettivamente il robot nelle condizioni di operare secondo tutti i requisiti previsti. In questo momento viene infatti aggiunto l'ultimo e più importante livello di competenza che riguarda la capacità di riconoscere, raccogliere e radunare gli oggetti protagonisti di questo caso di studio. Questa attività richiede di introdurre un grande numero di operazioni che si decide di suddividere nei seguenti *behavior*: `Detect`, `Gripper`, `Forage` ed infine `Repulse`. L'idea alla base è che il *behavior* `Detect` fornisca delle informazioni deducibili dalle immagini catturate dalla telecamea al fine di individuare eventuali oggetti di fronte al robot. Il *behavior* `Forage` utilizzarà poi queste informazioni per decidere di raccogliere un determinato oggetto sfruttando le funzionalità messe a disposizione dal *behavior* `Gripper`. Infine, una volta raccolto un oggetto viene impartito al sistema il comando di raggiungere l'area goal in cui radunare gli oggetti e, una volta lì, si sfrutta nuovamente il *behavior* `Gripper` per rilasciare l'oggetto. Dopo aver effettuato un po' di prove si è optato per aggiungere anche un ulteriore *behavior* `Repulse` con lo scopo di allontanare il robot dall'area goal dopo aver rilasciato un oggetto.

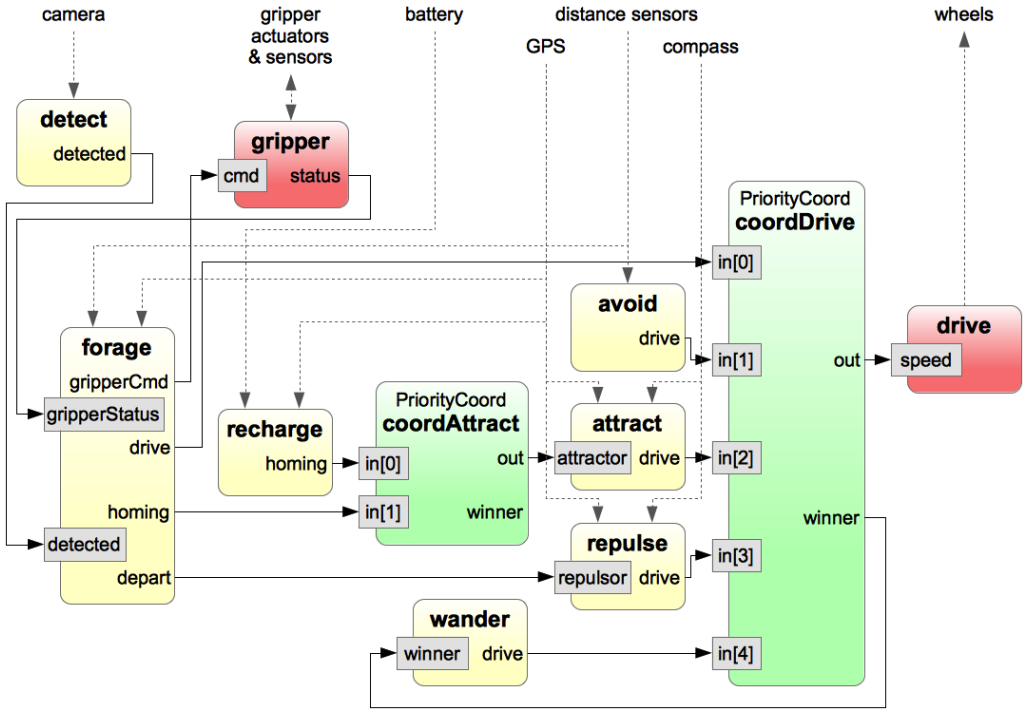


Figura 41: Schema completo del controllore `CtrlWanderRechargeForage`.

Detect

Questo primo *behavior* ha il solo compito di analizzare regolarmente l'immagine proveniente dalla telecamera ed individuare eventuali oggetti di fronte al robot. Si ricorda che gli oggetti d'interesse per il robot sono di colore rosso acceso ed inoltre sono le uniche entità presenti nell'arena ad avere questo caratteristico colore. Per i semplici scopi dimostrativi di questo caso di studio si opta quindi di decretare la presenza di un oggetto d'interesse esclusivamente sulla base del colore assunto dall'immagine. Ci si munisce quindi di una funzione `redRatio` che ha il compito di analizzare una porzione di immagine e restituire la percentuale di pixel "rossi" presenti in tale regione. Si ricorda che l'immagine è codificata come una matrice di pixel ed ogni pixel è modellato come un numero intero contenente le componenti RGB (rosso, verde, blu) del rispettivo colore. In questo caso un pixel viene considerato "rosso" quando il rapporto tra la componente rossa e le altre due è superiore ad una certa soglia (`RED_RATIO`) fissata per il momento a 2.

L'immagine fornita dalla telecamera viene idealmente suddivisa in tre regioni: destra, centro e sinistra. A questo punto il *behavior* non deve far altro che analizzare singolarmente queste tre regioni e fornire le dovute informazioni sugli eventuali oggetti riconosciuti. L'output del *behavior* è rappresentato da un'istanza della struttura `Detected` che riporta le percentuali di rosso riscontrate sulle tre porzioni dell'immagine. Per evitare di generare falsi positivi si ritiene appropriato imporre un valore di soglia (`DETECTED_THRESHOLD`) anche a queste percentuali in modo che non vengano presi in considerazione oggetti troppo lontani dal robot (che quindi appaiono piccoli sull'immagine ripresa dalla telecamera). Questa soglia viene fissata ad un valore empirico del 40%.

Si decide infine di generare messaggi di output solamente se vengono effettivamente riconosciuti degli oggetti.

```
...
public class Detect extends foraging_stub.Detect_stub {
  static public class Detected {
    public final float left, front, right;
    ...
  }
```

```
  ...
  public void step() {
    int[] img = cam.getImage();
    float l = leftRedRatio(img);
    float f = middleRedRatio(img);
    float r = rightRedRatio(img);
    if (l < DETECTED_THRESHOLD) { l = 0; }
    if (f < DETECTED_THRESHOLD) { f = 0; }
    if (r < DETECTED_THRESHOLD) { r = 0; }
    if (l + f + r > 0) {
      _detected.send(new Detected(l, f, r));
    }
  }

  private float leftRedRatio(int[] img) {
    return redRatio(img, w, h, 0, w/3);
  }

  private float middleRedRatio(int[] img) {
    return redRatio(img, w, h, w/3, w*2/3);
  }

  private float rightRedRatio(int[] img) {
    return redRatio(img, w, h, w*2/3, w);
  }

  /** Compute the ratio of almost-red pixels in a portion of an image. */
  private float redRatio(int[] img, int w, int h, int start, int end) {
    int count = 0;
    for (int x=start; x<end; x++) {  // columns
      for (int y=0; y<h; y++) {  // rows
        int r = Camera.imageGetRed(img, w, x, y);
        int g = Camera.imageGetGreen(img, w, x, y);
        int b = Camera.imageGetBlue(img, w, x, y);
        if (r >= g*RED_RATIO  &&  r >= b*RED_RATIO) {
          // the pixel is almost-red
          count++;
        }
      }
    }
    return ((float)count)/((float)((end-start)*h));
  }
}
```

Gripper

Questo *behavior* costituisce un altro caso di "attuatore virtuale" che si fa carico di gestire gli attuatori della pinza del robot e fornisce al sistema due operazioni di alto livello denominate `Pick_Up` e `Release` che consistono rispettivamente nel raccogliere e rilasciare un oggetto. Il *behavior* `Gripper` viene progettato secondo un paradigma propriamente funzionale forse più vicino ai sistemi gerarchici che a quelli *behavior-based*. Si tratta infatti di un modulo che ha il

solo scopo di eseguire il comando che gli viene sottoposto tramite la porta di input `cmd`.

Il funzionamento di questo *behavior* è dettato da un automa a stati finiti composto da otto stati: `Idle`, `Down_Pick`, `Close`, `Up_Pick`, `Carry`, `Down_Rel`, `Open`, `Up_Rel`. Come fatto per il *behavior* `Avoid` si preferisce mostrare il diagramma dell'automa piuttosto che il relativo codice implementativo.

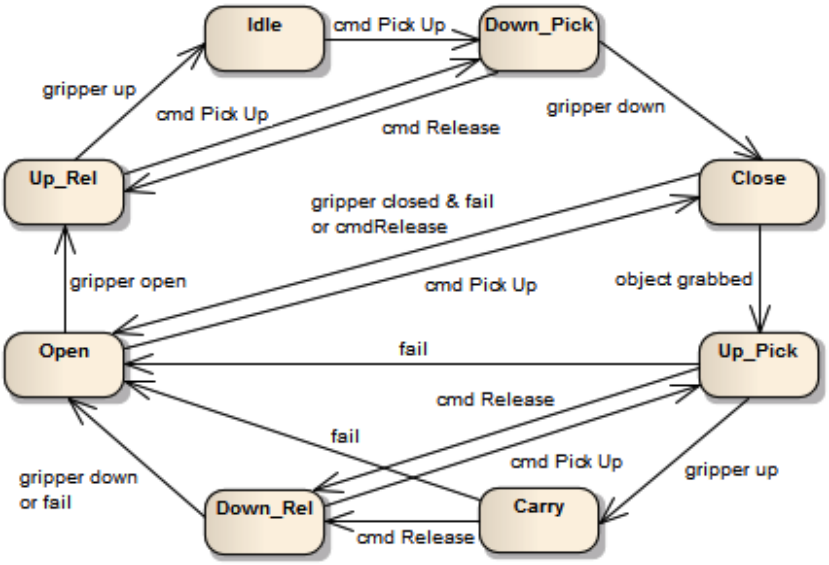


Figura 42: Diagramma degli stati del *behavior* `Gripper`.

Lo stato Idle rappresenta la situazione in cui la pinza è a riposo ed è quindi mantenuta sollevata dal robot senza che stia trattenendo alcun oggetto. Al momento della ricezione del comando `Pick_Up` inizia la fase di raccolta dell'oggetto che segue tutti gli stati `Down_Pick`, `Close` e `Up_Pick` per terminare nello stato `Carry`. Quest'ultimo infatti rappresenta il caso in cui il robot sta trasportando un oggetto e la pinza è mantenuta sollevata mentre stringe l'oggetto raccolto. A seguito di questa operazione si uscirà dallo stato di `Carry` alla ricezione del comando `Release` avviando la fase di rilascio dell'oggetto che attraversa gli stati `Down_Rel`, `Open` e `Up_Rel` per ritornare infine nello stato `Idle`.

L'output `status` del modulo viene costantemente mantenuto aggiornato e riflette lo stato interno del modulo con un livello di granularità meno fine. Gli stati segnalati all'esterno sono quattro e possono essere utilizzati dai *behavior* della rete come dei “sensori” che segnalano una situazione interna al robot:

- `Idle` – mentre il *behavior* è nello stato `Idle`;
- `Picking` – mentre il *behavior* è negli stati `Down_Pick`, `Close`, `Up_Pick`;
- `Carrying` – mentre il *behavior* è nello stato `Carry`;
- `Releasing` – mentre il *behavior* è negli stati `Down_Rel`, `Open`, `Up_Rel`.

Segue una breve descrizione del funzionamento del modulo nei vari stati.

Lo stato `Idle` si limita ad attendere un comando `Pick_Up`.

Lo stato `Down_Pick` impartisce alla pinza il comando di scendere nella posizione inferiore dopo di che, monitorando i sensori di posizione associati agli attuatori, attende che questo evento si verifichi per poi transitare in `Close`.

Lo stato `Close` si occupa di chiudere la pinza nella speranza di afferrare un oggetto. Agli attuatori della pinza viene richiesto di portarsi nella posizione di chiusura per poi iniziare a monitorare i sensori di posizione e forza associati a questi (*force feedback*): se gli attuatori arrivano vicini alla posizione finale senza che sia rilevata un certo riscontro in termini di forza vincolare, significa che non hanno incontrato nessun ostacolo e quindi nessun oggetto è stato effettivamente afferrato. Se invece i sensori di forza segnalano una certa resistenza significa che è stato impugnato un oggetto e si procede nello stato `Up_Pick`. Le letture fornite dai sensori di forza soffrono di cambiamenti repentini nel momento in cui l'attuatore inizia ad incontrare la resistenza dell'oggetto da raccogliere. Per filtrare eventuali letture fuorvianti si mantiene aggiornata una qualche forma di valore medio di forza rilevata nel passato (dando più importanza alle letture recenti) ed ogni valore con uno scarto troppo elevato da questo viene considerato errato. Quando un valore considerato corretto indica un certo livello di resistenza, si considera che l'oggetto sia stato correttamente afferrato e si procede nello stato `Up_Pick`.

```
private final FSM.State CLOSE = new FSM.State() {
  private float avg;
  protected void entry() {
    leftGrip.setPosition(POS_GRIP_CLOSED);
    rightGrip.setPosition(POS_GRIP_CLOSED);
    avg = (float)leftGrip.getMotorForceFeedback();
    _status.send(ST_PICKING);
  }
```

```
  protected void exec() {
    float f = (float)leftGrip.getMotorForceFeedback();
    avg = RATIO*f + (1-RATIO)*avg;
    if (isClosed()  &&  fail()) {  // no object has been grabbed
      transition(OPEN);
    } else if (cmdRel()) {
      transition(OPEN);
    } else if (Math.abs(f - avg) < 2  &&  f < -5) {
      // an object has been properly grabbed
      transition(UP_PICK);
    } else {
      _status.send(ST_PICKING);
    }
  }
}
```

Lo stato `Up_Pick` richiede di sollevare la pinza del robot e attende che questa azione sia terminata per passare poi allo stato `Carry`.

Lo stato `Carry` è in sostanza equivalente allo stato `Idle` con la differenza che attende la ricezione di un comando `Release` per transitare in `Down_Rel`.

Lo stato `Down_Rel` impartisce il comando di far scendere la pinza nella posizione inferiore dopo di che, al completamento di questo movimento, transita nello stato `Open`.

Lo stato `Open` è quello predisposto a rilasciare la presa della pinza e deve quindi semplicemente richiedere l'apertura della pinza. Fatto ciò, per motivi di sicurezza, si attende che la pinza sia completamente aperta prima di passare allo stato `Up_Rel`.

Lo stato `Up_Rel` è quello in cui si conclude il ciclo operativo delle funzioni `Pick Up` e `Release` fornite dal *behavior* e finisce per ritornare nello stato `Idle`.

In tutti gli stati inoltre è possibile annullare l'operazione semplicemente richiedendo al *behavior* di eseguire il comando opposto a quello correntemente in esecuzione (vedi diagramma degli stati in figura 42).

Infine si sottolinea che durante la permanenza negli stati `Up_Pick`, `Carry` e `Down_Rel` viene periodicamente controllato che l'oggetto sia ancora trattenuto nella pinza del robot e che non sia quindi caduto per qualche motivo. A tal fine si implementa una funzione `fail` che verifica questo evento sulla base del valore fornito dal sensore di forza.

```
private boolean fail() {
  return leftGrip.getMotorForceFeedback() > -1f;
}
```

Repulse

Questo *behavior* è sostanzialmente la versione duale del *behavior* `Attract` sviluppato in precedenza e fornisce in output i comandi da impartire alle ruote del robot perché questo si allontani il più possibile da una posizione sorgente fornita al modulo tramite l'input `repulsor`. Non si mostra un'analisi dettagliata di questo *behavior* perché è facilmente deducibile dalla descrizione del *behavior* `Attract` a meno di alcune differenze. Per esempio in questo caso il robot non dovrà mai raggiungere una destinazione puntuale in cui debba fermarsi.

Forage

Questo *behavior* è senz'altro quello più complesso del controllore tant'è che si sarebbe potuto anche scomporre ulteriormente in *behavior* di granularità più fine. Si tratta di un automa con sei stati: `Depart`, `Idle`, `Approach`, `Pick_Up`, `Delivery` e `Release`.

Lo stato `Depart` è quello iniziale e in questo stato il *behavior* si occupa semplicemente di segnalare al modulo `Repulse` l'intenzione di allontanare il robot dalla regione goal fino ad una certa distanza per poi transitare nello stato `Idle`. Lo scopo è quello di spingere il robot ad esplorare il territorio circostante alla regione goal.

```
private final FSM.State DEPART = new FSM.State() {
  ...
  protected void exec() {
    _depart.send(Misc.GOAL);
    double[] pos = gps.getValues();
    if (Misc.dist((float)pos[0], (float)pos[2],
                  Misc.GOAL.x, Misc.GOAL.z) > Misc.GOAL_R*2) {
      // far enough away from the goal area
      transition(IDLE);
    }
  }
}
```

Nello stato `Idle` il *behavior* resta semplicemente in attesa di informazioni provenienti dal *behavior* `Detect` riguardanti eventuali oggetti individuati di

fronte al robot. Se il robot si trova all'interno della regione goal qualsiasi informazione viene ignorata in modo da evitare che vengano raccolti nuovamente oggetti già recuperati in precedenza. Nel caso contrario si procede come segue: se è stato rilevato un oggetto di fronte al robot si transita nello stato `Approach`; se invece è presente un oggetto su di un lato si reagisce semplicemente cercando di ruotare il robot in quella direzione.

Gli output diretti alle ruote del robot devono essere considerati più prioritari rispetto a quelli generati dal *behavior* `Avoid` perché questo tende ad allontanare il robot da ogni tipo di ostacolo, e quindi anche dagli oggetti che devono invece essere raccolti. Perciò l'uscita `drive` di questo modulo deve essere connessa al coordinatore predisposto per i comandi di spostamento del robot in una posizione che prevalga su quella assegnata al *behavior* `Avoid`.

```
private final FSM.State IDLE = new FSM.State() {
  ...
  protected void exec() {
    double[] pos = gps.getValues();
    if (Misc.dist((float)pos[0], (float)pos[2],
                  Misc.GOAL.x, Misc.GOAL.z) <= Misc.GOAL_R*1.3f) {
      // don't consider the sticks inside the goal area
      return;
    }
    if (detected != null) {
      // new or previous detected stick (from the last control step)
      if (detected.front > 0) {  // approach a stick ahead
        transition(APPROACH);
      } else if (detected.left >= detected.right) {
        // slowly turn toward a stick on the left
        _drive.send(new Drive.Speed(-0.4f, +0.4f));
      } else if (detected.right > detected.left) {
        // slowly turn toward a stick on the right
        _drive.send(new Drive.Speed(+0.4f, -0.4f));
      }
    }
  }
}
```

Lo stato `Approach` ha il compito di portare il robot nella posizione ideale per l'operazione di raccolta. Controlla regolarmente che l'oggetto sia ancora di fronte alla telecamera e si avvicina finché non inizia a sentirne la presenza con i sensori di distanza anteriori. A questo punto si assicura che l'oggetto sia esattamente al centro verificando che sia rilevato da entrambi i sensori anteriori. Infine si avvicina (o allontana) molto lentamente fino al momento in cui i sen-

sori anteriori segnalano da distanza ottimale per prelevare l'oggetto: in questo momento la fase di approccio termina e si transita nello stato `Pick_Up`.

```
private final FSM.State APPROACH = new FSM.State() {
  ...
  protected void exec() {
    if (detected == null  ||  detected.front == 0) {
      // the stick has disappeared: why?
      transition(IDLE);
      return;
    }
    // move ahead until the front distance sensors feel the stick
    if (ds2 + ds3 == 0) {
      _drive.send(new Drive.Speed(1, 1));
      return;
    }
    // make sure that the stick is sensed by both front distance sensors
    if (ds2 == 0) {
      _drive.send(new Drive.Speed(+0.2f, -0.2f));
      return;
    }
    if (ds3 == 0) {
      _drive.send(new Drive.Speed(-0.2f, +0.2f));
      return;
    }
    // the stick is ahead in the middle
    float delta = TARGET - (ds2 > ds3 ? ds2 : ds3);
    // move slowly in the right position
    _drive.send(new Drive.Speed(delta/TARGET, delta/TARGET));
    if (Math.abs(delta) < 0.05) {
      // the stick is in the right position -> pick it up
      transition(PICK_UP);
    }
  }
}
```

Lo stato `Pick_Up` si preoccupa semplicemente di mantenere la posizione corrente del robot, impartire il comando di raccolta dell'oggetto al `Gripper`, attendere che tale comando venga preso in carico (cosa indicata dalla segnalazione di stato `Picking` dal `Gripper`) ed infine aspettare il completamento dell'operazione di raccolta che può avvenire con successo o con fallimento (indicati rispettivamente dalla segnalazione di stato `Carrying` o `Idle` del `Gripper`). Se l'oggetto viene correttamente raccolto si passa allo stato `Deliver`, se invece si verifica un fallimento si torna allo stato `Idle`. Durante questa operazione non vengono più presi in considerazioni gli input forniti dalla telecamera e dai sensori di distanza in quanto la pinza del robot durante gli spostamenti finisce per influire sulle informazioni che questi restituiscono rendendone vano l'utilizzo.

```
private final FSM.State PICK_UP = new FSM.State() {
  private boolean sync;
  protected void entry() {
    _gripperCmd.send(Gripper.CMD_PICK_UP);
    sync = false;
  }
  protected void exec() {
    _drive.send(new Drive.Speed(0, 0));  // hold position
    Byte s = _gripperStatus.peek();
    if (!sync) {
      if (s == Gripper.ST_PICKING) {
        // wait for the gripper to start the picking task
        sync = true;
      }
    }
    if (sync) {
      switch (s) {
        case Gripper.ST_CARRYING:  // success
          transition(DELIVER);
          break;
        case Gripper.ST_IDLE:  // failure
          transition(IDLE);
          break;
        }
      }
    }
  }
}
```

Lo stato `Deliver` è quello predisposto a portare il robot a rilasciare l'oggetto recuperato all'interno della regione goal. Per svolgere questo compito deve influenzare un *behavior* di tipo `Attract`. Al momento l'unica istanza di questo *behavior* è controllata da `Recharge` ma dopo una breve riflessione si giunge alla conclusione che entrambi i *behavior* possono influenzare lo stesso modulo dando però priorità a `Recharge`. A tal fine si connette un coordinatore di tipo `PriorityCoord<Location>` all'ingresso di `Attract` per mediare gli output dei due *behavior* in competizione. In alternativa si sarebbe potuto ovviamente introdurre una ulteriore istanza del modulo `Attract` in parallelo a quella già presente. Una volta raggiunta una posizione casuale al centro della regione goal si da il via all'operazione di rilascio transitando in `Release`. Durante tutto il tragitto viene continuamente monitorato lo stato segnalato dal `Gripper` e se questo dovesse indicare che l'oggetto è accidentalmente caduto dalla pinza si ritorna allo stato `Idle`.

```
private final FSM.State DELIVER = new FSM.State() {
  protected void entry() {
```

```
    _homing.send(Misc.GOAL);
  }
  protected void exec() {
    Byte s = _gripperStatus.peek();
    if (s != Gripper.ST_CARRYING) {  // the stick has fallen
      transition(IDLE);
      return;
    }
    _homing.send(Misc.GOAL);
    double[] pos = gps.getValues();
    if (Misc.dist((float)pos[0], (float)pos[2],
                  Misc.GOAL.x, Misc.GOAL.z) <=
                                    Misc.GOAL_R*Misc.RANDOM.nextFloat()) {
      // inside the goal area -> release the stick
      transition(RELEASE);
    }
  }
}
```

Infine `Release` è lo stato che conclude l'intera operazione di recupero di un oggetto e semplicemente segnala al `Gripper` la volontà di rilasciare l'oggetto trasportato e resta in attesa del completamento di questa azione per poi transitare nello stato `Depart` che allontana il robot dalla regione goal.

```
private final FSM.State RELEASE = new FSM.State() {
  protected void entry() {
    _gripperCmd.send(Gripper.CMD_RELEASE);
    exec();
  }
  protected void exec() {
    _drive.send(new Drive.Speed(0, 0));  // hold position
    Byte s = _gripperStatus.peek();
    if (s == Gripper.ST_IDLE) {
      transition(DEPART);
    }
  }
}
```

Anche nel caso del *behavior* `Forage`, in quanto le informazioni provenienti dal modulo `Detect` sono fornite solo se c'è effettivamente un oggetto di fronte al robot, si può verificare l'ormai noto problema dovuto al continuo invio di messaggi tra i due *behavior* paralleli. E anche questa volta come soluzione si ricorre alla tecnica di estendere la validità temporale di un messaggio ad un periodo pari al doppio del *control step* dando ovviamente priorità ad un eventuale messaggio più recente ricevuto nel frattempo.

```
public void step() {
  ...
  Detect.Detected d = _detected.get();
```

```
  if (d != null) {
    detected = d;  // consider the new detected stick
  }
  fsm.exec();
  if (d == null) {  // don't consider the old detected stick any more
    detected = null;
  }
}
```

4.5 Esito delle sperimentazioni

Analizzando la soluzione al caso di studio appena trattato si possono riconoscere facilmente alcune delle peculiarità proprie dei sistemi *behavior-based*. Le fasi di sviluppo che sono state seguite mettono subito in risalto la modularità e la capacità di estensione delle soluzioni ottenute secondo questo approccio. La natura di queste architetture consente infatti abbastanza agevolmente di suddividere una soluzione in più moduli paralleli e in linea di principio indipendenti. L'aggiunta di nuove competenze sotto forma di nuovi *behavior* avviene senza troppe difficoltà e soprattutto senza invalidare la progettazione e l'implementazione già effettuata per le altre parti del sistema. Infine si noti come un controllore di questo genere ben progettato offra una certa robustezza essendo in grado di mantenere un limitato livello di operatività anche a fronte del fallimento di un *behavior* (per fallimento si intende il caso in cui un *behavior* smetta di generare messaggi e perciò di perseguire il proprio obiettivo).

Durante lo sviluppo della soluzione al caso di studio è evidente il fatto che si possono adottare suddivisioni più o meno fini dei *behavior* da mettere in campo. In questo caso ci si appoggia ad una suddivisione abbastanza grossolana che ha portato alla generazione di *behavior* leggermente complessi (come il *behavior* `Forage`). Partizionamenti attuati con granularità più fine porterebbero a generare *behavior* più semplici ma molto più numerosi rischiando così di compromettere la semplicità e la chiarezza del sistema nel suo complesso. D'altro canto però si avrebbero *behavior* più versatili e più facilmente riutilizzabili in altri contesti, senza contare poi che in un sistema più partizionato si aprono una serie di possibilità che permettono, ad esempio, ad una qualunque entità di influenzare porzioni specifiche e limitate del sistema invece che agire su di un livello di competenza nella sua interezza.

Ragionando sulla soluzione mostrata e sul framework in sé ci si può rendere conto però anche di alcune problematiche. Prima di tutto sono individuabili diverse modalità di comunicazione tra i moduli. Una di queste prevede che un *behavior* segnali l'intenzione di agire su di un secondo modulo, sia che si tratti

di un altro *behavior* che di un semplice coordinatore, tramite uno scambio continuo di messaggi che termina solo nel momento in cui tale intenzione cessa. In una situazione del genere il ricevente deve necessariamente consumare i messaggi in arrivo affinché riconosca che il mittente non intende più inviare informazioni nel momento in cui rilevi la mancanza di input. In realtà questa tecnica non è corretta a causa del problema già esposto varie volte nel corso dei capitoli precedenti e quindi all'occorrenza della prima mancanza di un input non si può assumere che il flusso di messaggi sia realmente terminato.

Come soluzione a questo problema è stato sempre proposto di allungare la validità temporale di ogni messaggio ricevuto di un ulteriore *control step* considerando come ancora valido l'ultimo messaggio ricevuto a fronte della prima segnalazione di mancanza di un input[43]. Ciò però determina sempre un ritardo di reazione pari alla durata del *control step* e, collegando in cascata più *behavior* che sfruttano questa stessa tecnica, il ritardo aumenta ad ogni passaggio finché l'informazione non arriva agli attuatori. Ovviamente tutto ciò rischia di minare pericolosamente la reattività del robot.

Se la situazione dovesse diventare insostenibile è necessario adottare una strategia differente. Una alternativa potrebbe essere l'utilizzo di un particolare messaggio per segnalare la fine di un flusso di messaggi che possa essere riconosciuto tempestivamente dal ricevente. Ovviamente questo protocollo deve essere rispettato da entrambi i *behavior* mittente e ricevente che dunque non sarebbero più indipendenti gli uni dagli altri. Un'opzione più corretta potrebbe essere invece l'utilizzo di un flusso continuo di messaggi accompagnati da un livello di attivazione (si ricordi la struttura `MsgAct<T>` vista nella sezione 3.3). Questi messaggi dovrebbero essere trasmessi ad ogni iterazione del *behavior* mittente e se questo non intendesse agire sul ricevente dovrebbe indicare un livello di attivazione nullo. Il ricevente da parte sua dovrà reagire coerentemente con il livello attivazione segnalato. In entrambe queste alternative il

43 Questa soluzione potrebbe essere direttamente incapsulata all'interno di un particolare tipo di `InputPort` che alla prima occorrenza di input mancante restituisca l'ultimo messaggio ricevuto per poi iniziare a segnalare input mancante al *contro step* successivo.

ritardo di reazione di ogni *behavior* coinvolto è ridotto al minimo ma ogni tanto può comunque capitare che si verifichi un ritardo di un *control step*.

Al di là di questo piccolo inconveniente in merito alle tempistiche di ricezione ed elaborazione dei messaggi, la soluzione al caso di studio dimostra chiaramente come il framework proposto, ed in generale l'approccio *behavior-based*, permetta di sviluppare sistemi di controllo molto reattivi ed affidabili. Nell'esempio analizzato il robot riesce infatti a reagire agli stimoli esterni pressoché istantaneamente e la possibilità di integrare *behavior* diversi ed indipendenti permette di inserire all'interno del controllo qualsiasi livello di "intelligenza" desiderato, sempre in un'ottica *behavior-based*. Eseguendo le simulazioni dei tre sistemi di controllo sviluppato si possono notare in realtà situazioni in cui il robot sembra non rispondere tempestivamente alla presenza di un ostacolo che si trovi di fronte al robot ma leggermente a sinistra o a destra. In questo caso in realtà il problema non è dovuto alla prontezza di risposta ma alla posizione e direzione dei sensori di prossimità che è tale per cui se un ostacolo sufficientemente piccolo viene avvicinato con una certa angolazione non viene rilevato finché non si trovi quasi a contatto con il robot: quando poi questo rileva l'ostacolo tende istintivamente a ruotare per evitarlo ma facendo ciò finisce spesso per colpirlo di lato con la pinza che è più larga della base del robot. È chiaro che per evitare questo problema sarebbe sufficiente modificare il *behavior* `Avoid` perché tratti questo caso separatamente dagli altri spingendo il robot ad eseguire una manovra d'urgenza che fermi subito il robot, lo faccia indietreggiare per un breve tratto, lo faccia ruotare su sé stesso in una certa direzione ed infine lo faccia avanzare per allontanarsi. Considerando che questa situazione si presenta praticamente solo nei confronti degli oggetti da raccogliere, si potrebbe in alternativa utilizzare l'output fornito dal modulo `Detect` per portare il robot ad evitare un ostacolo con un certo preavviso, prima che venga rilevato dai sensori di prossimità.

Si può infine affermare con certezza che l'utilizzo dei coordinatori risulta una strategia molto efficiente per l'individuazione della migliore azione da compiere in ogni dato istante. Configurando opportunamente queste entità si

riesce di fatto ad indurre il sistema di controllo ad assumere il comportamento più consono alla situazione corrente soprattutto in funzione dell'urgenza di tale situazione (si pensi ad esempio ai comportamenti per evitare gli ostacoli o raggiungere la stazione di carica).

Nella soluzione al caso di studio sono stati impiegati solamente coordinatori a priorità fissa il che può portare a soluzioni poco flessibili; per di più la scelta del giusto ordine di priorità dei diversi *behavior* diviene un compito importantissimo e per nulla banale. Nel caso poi in cui un modulo sia "controllato" o influenzato da altri *behavior* la priorità che questo dovrebbe possedere potrebbe dipendere dallo specifico *behavior* che ha agito su di esso (nel caso di studio esaminato ciò accade per il *behavior* `Attract`). In situazioni del genere è strettamente necessario ricorrere ad altri tipi di coordinazione, per esempio ad una coordinazione basata su livelli di attivazione propagati da *behavior* a *behavior* fino a giungere ad un coordinatore di tipo *action-selection* (quello che nel framework prende il nome di `ActivationCoord`). Il coordinatore a sua volta riporterebbe in uscita il livello di attivazione associato all'input che può così essere riutilizzato in cascata portando a reti simili a quelle viste nell'architettura iB2C (vedi sezione 2.6.2).

Capitolo 5
Considerazioni finali e sviluppi futuri

Alla luce degli studi e delle sperimentazioni effettuate si può affermare con soddisfazione che l'esito di questa analisi approfondita sul paradigma *behavior-based* e la risoluzione del caso di studio hanno dimostrato come tale strategia di controllo sia particolarmente adatta alla programmazione di robot situati in ambienti dinamici ed incerti. Questo può essere senz'altro considerato un buon punto di partenza per perseguire la visione di "un robot in ogni casa" indicata nell'introduzione.

Il framework proposto si è rivelato essere un buon supporto per la modellazione e l'implementazione di soluzioni modulari e particolarmente reattive alle situazioni che si presentano al robot. Tecnicamente parlando il presente framework è ovviamente in stadio ancora embrionale e in merito ad eventuali sviluppi futuri diverse attività possono essere svolte per estenderlo ulteriormente.

In merito alle prestazioni è evidente il fatto che se si utilizza un metodo di coordinazione competitivo un solo *behavior* in ingresso al coordinatore ottiene il controllo sull'output e di conseguenza tutte le computazioni effettuate dagli altri vengono semplicemente ignorate causando dunque una certa perdita di efficienza per il fatto di aver svolto del lavoro "inutilmente". Questa situazione potrebbe essere migliorata facendo uso di segnalazioni *inter-behavior* che permettano al *behavior* con priorità superiore di inibire direttamente quelli meno prioritari i quali, a seguito di questa segnalazione, possono evitare del tutto di generare un output. Nel framework in esame questa strategia può essere messa in pratica mediante l'uso di un input di tipo `InputInfluence` che riceva le segnalazioni dai *behavior* con priorità maggiore e che sia costantemente con-

trollato dal *behavior* che lo possiede. Come estensione dell'entità base `Behavior` del framework si potrebbe addirittura pensare di introdurre un secondo tipo di *behavior* più specializzato, magari con il nome di `InfluenceableBehavior`, che presenti di default degli input per inibirne od eventualmente stimolarne l'azione, riprendendo in un certo senso la stessa interfaccia standard che presentano i moduli dell'architettura iB2C. Lo *scheduler* potrebbe perfino essere modificato (o esteso) in modo da ignorare completamente eventuali *behavior* con livello di attivazione nullo.

A livello implementativo un'altra estensione potrebbe prevedere di supportare qualche tipo di protocollo di comunicazione real-time per permettere ad un sistema di controllo di essere fisicamente distribuito su più nodi computazionali forniti dall'hardware: questa ipotesi però è strettamente legata agli specifici robot considerati in quanto ciascuno impiega infrastrutture ad hoc sia per quanto concerne i microcontrollori che le reti di comunicazione tra essi.

Si potrebbe inoltre creare una versione del framework compatibile con più linguaggi di programmazione e con i sistemi operativi real-time più diffusi: dato che per il momento è stata utilizzata la tecnologia *Java* ci si potrebbe appoggiare a *Real-Time Java* trasformando i *behavior* in task periodici direttamente affidati alla *Java Virtual Machine* e al sistema operativo real-time sottostante evitando perciò del tutto l'utilizzo dello *scheduler* introdotto per simulare la concorrenza.

Da un punto di vista più teorico ed ingegneristico si potrebbe invece riprendere il linguaggio introdotto per la specifica strutturale di un sistema di controllo e rifinirlo per arrivare a descrivere formalmente un modello computazionale completo. Al momento infatti tale linguaggio è inquadrato solamente come un mezzo per arrivare alla generazione automatica di una serie di classi *Java* di supporto ma potrebbe essere considerato come un punto di partenza per la definizione di un linguaggio che comprenda tutti gli aspetti del sistema di controllo, sia la sua struttura che le specifiche attività eseguite dai vari *behavior*. Per ora infatti quest'ultime vengono specificate direttamente in *Java* appoggiandosi alla semantica operazionale di quest'ultimo, ma l'introduzione di

un linguaggio integrale corredato da semantica operazionale comporterebbe il vantaggio di rimanere del tutto indipendenti dallo specifico linguaggio di implementazione di livello inferiore lasciando al compilatore l'incombenza di creare tutta l'applicazione eseguibile sulla base della specifica dell'utente.

Un altro importante aspetto legato all'ingegneria del software riguarda lo studio e l'analisi di pattern ricorrenti che possano aiutare gli sviluppatori durante la fase di progetto dei loro sistemi per affrontare particolari problematiche che si presentano di frequente. Già diversi tentativi sono stati fatti a riguardo (si vedano per esempio [10] e [21]) e potrebbe essere interessante cercare di testare questo framework su di una lunga serie di casi di studio a seguito dei quali possano essere individuate soluzioni e strategie a cui si possa far comunemente ricorso di fronte a determinate situazioni.

Infine si sottolinea il fatto che il caso di studio presentato qui non è particolarmente complicato, mentre potrebbe essere molto interessante valutare il framework anche sulla base di un insieme di casi in cui si riveli necessario introdurre particolari concetti e tecniche proprie dell'intelligenza artificiale e della pianificazione deliberativa per verificare con quanta semplicità possa avvenire l'integrazione di questi aspetti con l'infrastruttura *behavior-based* fornita dal framework portando magari ad individuare linee guida e design pattern per lo sviluppo di sistemi di controllo con capacità superiori, o suggerendo addirittura una estensione del framework stesso in direzione di una architettura ibrida.

5.1 L'apprendimento nei sistemi behavior-based

Nel corso di questa tesi purtroppo non c'è stato molto spazio per trattare un altro importante aspetto riguardante la robotica e i sistemi artificiali in generale: ci si riferisce al campo dell'apprendimento.

L'apprendimento può essere definito come la capacità di un sistema artificiale di adattare il proprio comportamento e di modificarsi significativamente con lo scopo di ottenere prestazioni migliori. Un sistema in grado di apprendere autonomamente durante la propria esecuzione è potenzialmente in grado di migliorarsi notevolmente col trascorrere del tempo e di adattarsi più o meno velocemente a fronte di eventuali cambiamenti delle condizioni al contorno, come di particolari proprietà dell'ambiente circostante. É evidente che lo sviluppo di un robot situato in una realtà dinamica e non strutturata potrebbe trarre vantaggi rilevanti dall'apprendimento [1].

Svariate tecniche di apprendimento sono state sviluppate in campo scientifico e molte di esse sono state applicate anche al campo della robotica. In genere possono essere suddivise tra tecniche off-line e tecniche on-line. Le prime possono essere usate in fase di sviluppo di un sistema di controllo al fine di determinare il valore pseudo-ottimo di importanti entità del sistema: in questa categoria ricadono ad esempio gli algoritmi euristici a popolazione (come gli algoritmi evolutivi [41]) che prevedono di effettuare un grande numero di tentativi guidati in cerca dei risultati migliori. Le seconde invece possono essere direttamente implementate nel sistema di controllo e influiscono con continuità sul funzionamento del robot: un importante esempio sono gli algoritmi di *reinforcement learning* [44] che intervengono regolarmente modificando il sistema sulla base di un riscontro che indica la correttezza o meno del funzionamento del sistema.

Ovviamente qui ci si concentra sulle opportunità che l'apprendimento offre nel contesto dei sistemi di controllo *behavior-based*.

Le entità principali dell'approccio *behavior-based* sono state definite nel capitolo 2 e ciascuna di esse è in qualche modo candidata ad essere estesa per

introdurre capacità di apprendimento in un sistema [1][6]. Per esempio è lecito pensare di fornire ad un sistema la capacità di variare la tecnica di coordinazione utilizzata al fine di migliorare le prestazioni del sistema. Oppure adattare i parametri più significativi di certi *behavior* così da fornire risposte più consone agli stimoli ricevuti. Nel caso limite si può persino pensare di applicare tecniche di apprendimento in grado di sintetizzare un intero *behavior* componendo entità elementari predefinite che possono assumere diversi livelli di granularità (dalla combinazione di semplici azioni a quella di moduli basilari anche complessi).

La forma più intuitiva di apprendimento può riguardare ad esempio la semplice capacità di un controllore di percepire informazioni durante l'esecuzione e sfruttare tali informazioni per creare una base di conoscenza simbolica distribuita sui vari *behavior* che compongono il sistema [6]. Un esempio in questa direzione è citato in [16] in cui un robot durante l'esecuzione acquisisce informazioni riguardanti entità significative dell'ambiente circostante (corridoi, pareti, zone aperte, etc.) sulla cui base provvede a costruire una mappa interna dell'area sotto forma di processi attivi e indipendenti anziché di un modello statico e centralizzato. Dopo di che utilizza questa mappa virtuale per spostarsi efficacemente nel mondo.

La forma più complessa di apprendimento riguarda invece la capacità di generare nuovi *behavior*, sia in fase di sviluppo del sistema che in fase di esecuzione del sistema stesso. Spesso questo problema è dovuto al fatto che la creazione di un *behavior* che fornisca buone risposte agli stimoli rappresenta in pratica un problema di ricerca in un spazio molto ampio di possibili soluzioni che ha quindi poche probabilità di successo in tempi accettabili.

In reazione a questo problema si possono usare tecniche più semplici basate sull'ipotesi che nel momento in cui si progetta e si sviluppa la struttura fisica di un robot vengono definiti già a priori molti degli aspetti in merito all'interazione tra il robot e l'ambiente (ad esempio come questo debba spostarsi). Tali aspetti sono quindi direttamente incorporati nel robot (si può dire che facciano parte “del suo corredo genetico”) e dunque non è necessario che vengano

"imparati". Come conseguenza si può sfruttare questa conoscenza a priori per definire un insieme di operazioni di base sulla base delle quali costruire lo spazio di ricerca di un eventuale algoritmo di apprendimento [6][16].

Gli approcci che seguono questa idea prevedono ad esempio di costruire il sistema di controllo per mezzo di *behavior* parzialmente specificati in cui determinati parametri (spesso numerici) possano essere stimati e tarati con tecniche di apprendimento per migliorare le prestazioni del robot [1]. Sempre in questa categoria rientra anche una strategia che sembra essere quella più adottata e che si focalizza solamente su come effettuare la coordinazione dei *behavior* di un sistema. In questo caso lo sviluppatore si impegna a predisporre una serie di *behavior* all'interno dei quali codificare la conoscenza a priori che possiede, dopo di che ci si affida ad un algoritmo di apprendimento per allenare i coordinatori del sistema affinché selezionino gli output migliori in merito alla situazione corrente [1]. Esempi di questa strategia sono forniti in [14] e [16].

In [14] è esposto un processo in cui viene sviluppato un sistema di controllo per il robot *Genghis* (mostrato in figura 7) che impara autonomamente ad attivare una serie di *behavior* in modo da permettere al robot di camminare efficacemente. I *behavior* che vengono predisposti compiono una determinata sequenza di azioni[44] e si attivano quando si verifica una certa condizione relazionata ad alcune variabili percettive. I coordinatori del sistema in ogni momento selezionano uno degli output generati dai *behavior* attivi che vi sono connessi. La scelta viene effettuata probabilisticamente sulla base della rilevanza e dell'affidabilità dei *behavior* che viene misurata statisticamente sulla base delle esperienze passate del robot. Si usa infine una speciale tecnica di *reinforcement learning* che si appoggia sul riscontro fornito da alcuni sensori aggiuntivi per stimare la forma pseudo-ottimale delle condizioni di attivazione associate a ciascun *behavior* al fine di massimizzare le prestazioni del robot.

In [16] è descritto un esempio simile in cui vengono individuati dei *behavior* basilari e altri *behavior* di più alto livello vengono ottenuti componendo

44 Ad esempio per ogni zampa del robot è associata un'istanza di un *behavior* che la solleva, la sposta in avanti, la poggia sul terreno e la risposta indietro.

gli output di alcuni dei primi tramite due diverse tecniche di coordinazione alternative: somma degli output o selezione di un solo output. Tale approccio viene poi esteso affinché i robot apprendano autonomamente, tramite *reinforcement learning*, quali tecniche sfruttare e come coordinare i vari *behavior*: lo spazio di ricerca invece che essere esteso a tutte le possibili azioni elementari del robot riguarda il livello dei *behavior* riducendosi quindi notevolmente e agevolando l'apprendimento.

Infine in [22] è mostrato un esempio in cui il sistema di controllo di un robot autonomo viene appreso tramite un algoritmo evolutivo. Il controllore è costituito da un automa a stati finiti che ricorda molto il modello di *Finite State Acceptor* descritto nella sezione 2.2. Ad ogni stato è associata una specifica abilità a disposizione del robot che resta attiva finché non termina in maniera naturale segnalando un successo o un fallimento. Queste abilità possono essere considerate simili al concetto di *behavior* e costituiscono delle procedure implementate manualmente che forniscono un insieme di azioni di base non elementari che il robot è in grado di compiere (quali raggiungere una certa destinazione, avvicinare la pinza ad una precisa posizione, raccogliere un oggetto, e così via). L'algoritmo evolutivo introdotto cerca di massimizzare le prestazioni del robot migliorando nel corso dell'esecuzione il valore di alcuni parametri associati agli stati e la struttura topologica dell'automa stesso che determina perciò le sequenze di attivazione delle abilità. Viene inoltre dimostrato come è possibile definire azioni in maniera gerarchica sulla base di altre abilità più semplici e sintetizzare automaticamente nuove azioni di base per mezzo di una rete neuronale artificiale che viene a sua volta allenata durante l'esecuzione dell'algoritmo stesso.

I riferimenti ad altri esempi possono essere trovati in [1], [6], e [18].

In genere l'impressione che si ha è che queste tecniche di apprendimento per il momento siano state studiate solamente in merito a specifici esperimenti svolti nel contesto di alcuni gruppi di ricerca. In questi casi le soluzioni vengono spesso costruite ad hoc senza appoggiarsi ad una specifica architettura *behavior-based* proprio per permettere agli algoritmi di apprendimento di

modificare o di specificare certe parti del sistema. Cercando un po' in letteratura tuttavia è possibile trovare alcuni esempi di architetture introdotte appositamente per favorire l'utilizzo di apposite strategie di apprendimento[45]. Comunemente però queste architetture sposano un solo determinato algoritmo di apprendimento e non risultano pertanto molto flessibili a riguardo.

Il framework qui sviluppato invece non si occupa minimamente delle tecniche di apprendimento restando dunque neutro a qualsiasi algoritmo. L'utente ha però la possibilità di inserire qualsiasi strategia praticamente in ogni parte del sistema. Ad esempio può definire dei coordinatori in grado di apprendere la funzione coordinatrice migliore per il sistema in esame, o può adottare una qualsivoglia tecnica per la sintesi automatica dei *behavior* o semplicemente per stimare il valore pseudo-ottimo di eventuali variabili su cui questi sono parametrizzati. Inoltre *Webots* fornisce un valido strumento per l'utilizzo di algoritmi di ottimizzazione off-line che richiedano di ripetere un grande numero di prove: si ricordi tal proposito l'entità *supervisor* descritta nella sezione 1.3.1.

45 Alcune di queste sono riportate tra i riferimenti bibliografici.

Riferimenti bibliografici

[1] Arkin R. C.
Behavior-Based Robotics, MIT Press, Cambridge, MA, USA, 1998.

[2] Brooks R. A.
"A Robust Layered Control System for a Mobile Robot"
MIT AI Memo 864, Sept. 1985.

[3] Brooks R. A.
"A Robust Layered Control System for a Mobile Robot"
IEEE Journal of Robotics and Automation, vol. 2 no. 1, Mar. 1986, pp. 14-23.

[4] Brooks R. A.
"A Robot that Walks; Emergent Behavior from a Carefully Evolved Network"
Neural Computation, vol. 1 no. 2, Summer 1989, pp. 253-262.

[5] Brooks R. A.
"The Behavior Language; User's Guide"
MIT AI Memo 1227, Apr. 1990.

[6] Brooks R.A.
"The Role of Learning in Autonomous Robots"
Proceedings of the Fourth Annual Workshop on Computational Learning Theory (COLT '91), Santa Cruz, CA, Morgan Kaufmann Publishers, Aug. 1991, pp. 5-10.

[7] Dorigo M., Schnepf U.
"Genetics-Based Machine Learning and Behaviour Based Robotics: A New Synthesis"
IEEE Transactions on Systems, Man, and Cybernetics, vol. 23 no. 1, 1993, pp. 141-154.

[8] Gates B.
"A Robot in Every Home"
Scientific American, Gen. 2007.

[9] Gottifredi S., Tucat M., Corbatta D., García A., Simari G. R.
"A BDI Architecture for High Level Robot Deliberation"
Inteligencia artificial: Revista Iberoamericana de Inteligencia Artificial, vol. 14, no. 46, 2010, pp 74-83.

[10] Graves A. R., Czarnecki C.
"Design patterns for behavior-based robotics"
IEEE Transactions on Systems, Man, and Cybernetics, Part A, vol. 30 no. 1, Jan. 2000, pp. 36-41.

[11] Gu D., Hu H., Reynolds J., Tsang E.
"GA-based Learning in Behaviour Based Robotics"
Proceedings of IEEE International Symposium on Computational Intelligence in Robotics and Automation, Kobe, Japan, July 2003, pp. 16-20.

[12] Konidaris G. D., Hayes G. M.
"An Architecture for Behavior-Based Reinforcement Learning"
Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems, vol. 13 no. 1, Mar. 2005, pp. 5-32.

[13] Lee S., Suh I. H.
"A Programming Framework Supporting An Ethology-based Behavior Control Architecture"
Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, Beijing, China, Oct. 2006.

[14] Maes P., Brooks R. A.
"Learning to Coordinate Behaviors"
AAAI, Boston, MA, Aug. 1990, pp. 796-802.

[15] Matarić M.
"Behavior-Based Systems: Main Properties and Implications"
Proceedings of IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems, Nice, France, May. 1992, pp 46-54.

[16] Matarić M.
"Behavior-Based Control: Examples from Navigation, Learning, and Group Behavior"
Journal of Experimental and Theoretical Artificial Intelligence, special issue on Software Architectures for Physical Agents, vol. 9 no. 2-3, Hexmoor H., Horswill I., Kortenkamp D. eds, 1997, pp 323-336.

[17] Matarić M.
"Behavior-Based Robotics as a Tool for Synthesis of Artificial Behavior and Analysis of Natural Behavior"
Trends in Cognitive Science, vol. 2 n. 3, Mar. 1998, pp 82-87.

[18] Matarić M.
"Behavior-Based Robotics"
MIT Encyclopedia of Cognitive Sciences, Wilson R. A., Keil F. C. eds, MIT Press, Apr. 1999, pp 74-77.

[19] Medeiros A. A. D.
"A survey of control architectures for autonomous mobile robots"
Journal of the Brazilian Computer Society, vol. 4 no. 3, Campinas, Apr. 1998.
http://dx.doi.org/10.1590/S0104-65001998000100004

[20] Michel O.
"Webots: Professional Mobile Robot Simulation"
International Journal of Advanced Robotic Systems, vol. 1 no. 1, 2004, pp 39-42.

[21] Proetzsch M., Luksch T., Berns K.
"Development of complex robotic systems using the behavior-based control architecture iB2C"
Robotics and Autonomous Systems, vol. 58 no. 1, Jan. 2010, pp. 46-67.

[22] Riano L., McGinnity T. M.
"Automatically composing and parameterizing skills by evolving Finite State Automata"
Robotics and Autonomous Systems, vol. 60 no. 4, Apr. 2012, pp. 639-650.

[23] Suh I. H., Lee S., Kim B. O., Yi B. J., Oh S. R.
"Design and Implementation of a Behavior-Based Control and Learning Architecture for Mobile Robots"
Proceedings of the 2003 IEEE International Conference on Robotics & Automation, Taipei, Taiwan, Sept. 2003.

[24] Vu T., Veloso M.
"Behavior programming language and automated code generation for agent behavior control"
Proceedings of The Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS04), New York, July 2004.

[25] Wahde M.
"A general-purpose method for decision-making in autonomous robots"
Proceedings of IEA-AIE 2009, LNCS (LNAI), 5579, 2009, pp. 1-10.

[26] Wahde M.
"Introduction to Autonomous Robots", retrived Dec 2012.
http://www.me.chalmers.se/~mwahde/courses/aa/2012/Wahde_IntroductionToAutonomousRobots.pdf

[27] Anderson. D. P.
"Subsumption for the SR04 and jBot Robots", 26 Mar. 2007.
http://www.dprg.org/articles/2007-03a

[28] Cyberbotics
Webots, Commercial Mobile Robot Simulation Software, retrieved Jan. 2012.
http://www.cyberbotics.com

[29] Cyberbotics
"Servo"
Webots – Documentation – Reference manual – Chapter 3: Nodes and API Functions, retrieved Jan. 2012.
http://www.cyberbotics.com/reference/section3.42.php

[30] Cyberbotics
"Controller Programming"
Webots – Documentation – User guide – Chapter 6: Programming Fundamentals, retrieved Jan. 2012.
http://www.cyberbotics.com/guide/section6.1.php

[31] Cyberbotics
"Supervisor Programming"
Webots – Documentation – User guide – Chapter 6: Programming Fundamentals, retrieved Jan. 2012.
http://www.cyberbotics.com/guide/section6.2.php

[32] FZI Karlsruhe
"Modular Controller Architecture 2", retrieved Dec. 2012.
http://rrlib.cs.uni-kl.de/mca-kl

[33] Idaho National Laboratory
"Behavior-Based Robotics Tutorial", retrieved Dec. 2012.
https://inlportal.inl.gov/portal/server.pt/community/behavior-based_robotics_tutorial/466

[34] K-Team Corporation
"K-Team Corporation | Mobile Robotics", retrieved Dec. 2012.
http://www.k-team.com

[35] Luksch T., Proetzsch M.
"iB2C - Integrated Behaviour-Based Control", retrieved Dec. 2012.
http://rrlib.cs.uni-kl.de/mca-kl/libraries/ib2c

[36] Scheutz M.
"Behavior-Based Robotics: A Brief Overview", 2004.
http://www.cse.nd.edu/courses/cse471/www/bbr/index.html

[37] Wikipedia
"Autonomous robot", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Autonomous_robot

[38] Wikipedia
"Belief-desire-intention software model", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Belief-Desire-Intention_software_model

[39] Wikipedia
"Control theory", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Control_theory

[40] Wikipedia
"Cybernetics", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Cybernetics

[41] Wikipedia
"Evolutionary algorithm", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Evolutionary_algorithm

[42] Wikipedia
"Industrial robot", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Industrial_robot

[43] Wikipedia
"Reactive planning", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Reactive_planning

[44] Wikipedia
"Reinforcement learning", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Reinforcement_learning

[45] Wikipedia
"Robot", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Robot

[46] Wikipedia
"Robotics", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Robotics

[47] Wikipedia
"Scene graph", retrieved Jan. 2012.
http://en.wikipedia.org/wiki/Scene_graph

[48] Xtext
"Xtext - Language Development Made Easy!", retrieved Jan. 2012.
http://www.eclipse.org/Xtext

[49] *Fast, Cheap & Out of Control*
Errol Morris, Sony Pictures Classics, 1997.

[50] *Rodney's Robot Revolution*
Andrea Ulbrick, Essential Media & Entertainment, 2008.

Appendice

Specifica BCN del controllore Wander & Recharge & Forage

```
package foraging;
controller CtrlWanderRechargeForage;

behavior Detect {
  output {
    Detect.Detected detected;
  }
}

behavior Forage {
  input {
    Detect.Detected detected;
    Byte gripperStatus;
  }
  output {
    Drive.Speed drive;
    Byte gripperCmd;
    Location homing;
    Location depart;
  }
}

behavior Gripper {
  input {
    Byte cmd;
  }
  output {
    Byte status;
  }
}

behavior Avoid {
  output {
    Drive.Speed drive;
  }
}

behavior Recharge {
  output {
    Location homing;
  }
}

behavior Attract {
  input {
    Location attractor;
  }
  output {
    Drive.Speed drive;
  }
}

behavior Repulse {
  input {
    Location repulsor;
  }
  output {
```

```
    Drive.Speed drive;
  }
}

behavior Wander {
  input {
    Integer winner;
  }
  output {
    Drive.Speed drive;
  }
}

behavior Drive {
  input {
    Drive.Speed speed;
  }
}

Detect detect;
Forage forage;
Gripper gripper;
Avoid avoid;
Recharge recharge;
PriorityCoord<Location> coordAttract;
Attract attract;
Repulse repulse;
Wander wander;
PriorityCoord<Drive.Speed> coordDrive;
Drive drive;

connect detect.detected to forage.detected;
connect gripper.status to forage.gripperStatus;
connect forage.gripperCmd to gripper.cmd;
connect forage.depart to repulse.repulsor;
connect recharge.homing to coordAttract.in[0];
connect forage.homing to coordAttract.in[1];
connect coordAttract.out to attract.attractor;
connect forage.drive to coordDrive.in[0];
connect avoid.drive to coordDrive.in[1];
connect attract.drive to coordDrive.in[2];
connect repulse.drive to coordDrive.in[3];
connect wander.drive to coordDrive.in[4];
connect coordDrive.out to drive.speed;
connect coordDrive.winner to wander.winner;
```

Ringraziamenti

Desidero esprimere prima di tutto un sentito ringraziamento ai professori Alessandro Ricci e Andrea Roli per avermi seguito con tanta pazienza e impegno durante lo sviluppo di questa tesi, per avermi dato la possibilità di esplorare un argomento per me nuovo e molto stimolante, e per la loro capacità di trasmettere passione, entusiasmo e coinvolgimento in ogni attività.

Un ringraziamento speciale va ovviamente alla mia famiglia, ai miei genitori e a mia sorella, che non mi hanno mai fatto mancare il loro sostegno, sia morale che materiale, e che hanno sempre contribuito affinché potessi superare ogni ostacolo mi si ponesse di fronte.

Vorrei ringraziare anche tutti i compagni di università che hanno reso indimenticabili questi lunghi cinque anni e che mi hanno permesso di affrontare con leggerezza centinaia di ore di lezione e decine di esami e di estenuanti progetti. Senza di loro tutto sarebbe stato indubbiamente più noioso e faticoso.

Non potrei certamente dimenticare poi tutti gli altri amici grazie ai quali ho sempre potuto scaricare le tensioni e che mi hanno quotidianamente donato momenti di spensieratezza a volte cercando anche di indurmi in tentazione, ma che in fondo in fondo mi hanno sempre spinto a non mollare mai anche di fronte alle corse contro il tempo più impegnative per riuscire a preparare un esame o a terminare un progetto nei tempi previsti.

Infine vorrei ringraziare tutti coloro particolarmente interessati al tema che hanno trovato le forze per leggere per intero questa tesi fino a giungere qui; a quelli fra loro che non si fossero ancora stancati mi sento di consigliare sentitamente di approfondire l'argomento con due appassionanti documentari divulgativi: *Fast, Cheap & Out of Control* [49] e *Rodney's Robot Revolution* [50].