ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

APPLICAZIONE DEL LINGUAGGIO AD AGENTI JASON PER LA PROGRAMMAZIONE DI ALTO LIVELLO DI ROBOT

Elaborata nel corso di: Sistemi Operativi LA

Tesi di Laurea di:
FRANCESCA CIOFFI

Relatore:
Prof. ALESSANDRO RICCI

Co-relatore:
Prof. ANDREA OMICINI

ANNO ACCADEMICO 2008–2009
SESSIONE II

PAROLE CHIAVE

Agente

Modello BDI

Jason

Robotica

Comportamento individuale

Ai miei nonni

Indice

Capitolo 1

Introduzione

Cosa può voler dire specificare il controllore di un robot usando come approccio Jason.

I sistemi software rivestono nel campo dell'informatica un ruolo di fondamentale importanza. Negli ultimi anni una caratteristica richiesta ai sistemi è la *decentralizzazione del controllo*, nell'ottica di un sistema visto come connessioni di parti che possono interagire, e dove ciascuna parte possiede un certo grado di autonomia nella scelta delle attività che devono essere compiute.

In tale contesto si introduce il paradigma degli agenti, in quanto include sia aspetti relativi ai modelli computazionali, sia aspetti relativi ai linguaggi.

Nella tesi si esplora l'applicazione del linguaggio di programmazione **Jason** applicato alla programmazione dei controllori di robot. In particolare, si esplora ciò che riguarda il comportamento individuale dell'agente.

1.1 Un visione d'insieme

Il tema unificante è l'idea di *agente intelligente*. Lo studio degli agenti che ricevono percezioni dall'ambiente ed eseguono azioni, e lo scopo di utilizzare tale paradigma per organizzare i sistemi software con un certo grado di complessità. La robotica non viene trattata come argomento scorrelato, ma nella sua funzione di mettere al servizio dell'agente gli strumenti per il rag-

giungimento degli obiettivi. Viene inoltre posto l'accento sull'importanza dell'ambiente nel determinare l'architettura di agente più appropriata.

1.1.1 Contenuti

La tesi si articola in sei capitoli tra cui i primi tre riguardano argomenti compilativi e i restanti sperimentali.

I sistemi multi-agente: nel secondo capitolo si introducono i concetti di agente e di sistema multi-agente. Aspetto centrale di questo capitolo è il modello BDI ad agenti, fondamentale per lo sviluppo per i linguaggi di programmazione ad agenti. Poi, si presenta una breve panoramica sulle tipologie di ragionamento per il paradigma ad agenti e le motivazioni che hanno portato al loro sviluppo. Il quarto e il quinto paragrafo rappresentano gli aspetti chiave contenuti in questo capitolo. Nel quarto paragrafo si introduce il linguaggio ad agenti Jason e si illustrano i componenti di cui si compone l'architettura ad agente. Nel quinto paragrafo partendo dalle conoscenze acquisite nei paragrafi precedenti (in particolare nel quarto) si descrive l'architettura di un agente Jason e i passi del ragionamento ciclico. Infine, si inserisce il concetto di ambiente, ovvero dove gli agenti sono situati.

Le basi della robotica: nel terzo capito si presenta una breve panoramica storica sull'evoluzione del concetto di robot. Si descrivono, quindi, i componenti principali di un robot, distinguendoli in base alle finalità con cui sono stati progettati. In seguito, si richiamano alcuni concetti di teoria del controllo e principi elementari di fisica per meglio comprendere le caratteristiche del controllore del robot. Dopo una base che riguarda più la robotica che l'informatica, si passa alla definizione del concetto di architettura e si descrivono gli aspetti principali. Poi, quindi, vengono passate in rassegna le varie tipologie di controllo.

Definizione di un controllore di robot tramite l'utilizzo di Jason: nel quarto capitolo si descrive che cosa vuol dire poter programmare il comportamento di un robot servendosi del linguaggio ad agenti Jason. Inizialmente, si descrive l'associazione 1-1 fra agente e robot per poi vedere gli aspetti un po' più in concreto tramite l'utilizzo di un semplice esempio. In fondo, si è inserito un approfondimento sullo studio

del comportamento di un singolo agente. Riflettendo sulle complessità di strutturare il comportamento di un agente per lo svolgimento di una attività complessa.

Caso di studio: nel quinto capitolo si analizza il comportamento del singolo robot per lo svolgimento di una attività complessa ovvero la pulizia di un ambiente. Si pone in luce l'aspetto che il robot è autonomo, ovvero non ha una mappa pre-impostata dell'ambiente che dovrà ripulire. Il robot durante la pulizia riconosce vari livelli di sporco e ha uno stato interno che gli fornisce l'indicazione sul suo stato di carica della batteria. Viene gestito il fatto che il robot durante la pulizia si scarica, poi autonomamente torna alla base per ricaricarsi ed in seguito riprende la pulizia circa dal punto in cui l'ha interrotta.

Conclusioni: nel sesto capitolo si mettono in luce degli aspetti di Jason che possono essere interessanti su cui è interessante riflettere. Possibili esplorazioni che potrebbero essere svolte in futuro prendendo tale tesi come punto di partenza.

1.2 Cenni introduttivi su Jason

Jason è un linguaggio di programmazione ad agenti, basato sul modello Belief-Desire-Intention (BDI), introdotto nell'ambito della ricerca.

In particolare, **Jason** è il primo *interprete* completamente autonomo per una versione molto migliorata di AgentSpeak. Il linguaggio interpretato da **Jason** è un'estensione del linguaggio astratto di programmazione chiamato AgentSpeak(L). **Jason** implementa la semantica operazione di tale linguaggio, e fornisce una piattaforma per lo sviluppo di sistemi multi-agente, includendo numerose funzionalità personalizzabili dall'utente. Una caratteristica di Jason in confronto con altri sistemi di agente BDI è che è stato implementato in *Java* (quindi *multipiattaforma*) ed è disponibile *Open Source* con licenza GNU LGPL.

Oltre ad interpretare il linguaggio AgentSpeak, **Jason** presenta anche:

- la nozione di "strong negation", assunzione del mondo chiuso;
- piano di gestione dei fallimenti, detti anche piani di riparazione;

- possibilità di interagire sulla base della comunicazione fra agenti;
- annotazioni sulle etichette dei piani, che possono essere utilizzate per elaborare le funzioni di selezione;
- supporto per lo sviluppo di ambienti (che normalmente non sono da programmare in AgentSpeak, in questo caso sono programmate in Java);
- la possibilità di eseguire un sistema multi-agente distribuito su una rete;
- completamente personalizzabile (in Java) funzioni di selezione, l'architettura generale dell'agente (percezione, belief-revision, comunicazione inter-agente, azione);
- un IDE[1] in forma di un plugin jEdit, l'IDE include un "*Jason Mind Inspector*" ovvero un ispettore della mente che aiuta l'utente in fase di debug.

A proposito dell'IDE, si introducono due screenshot, il primo in figura 1.1 è l'ambiente di lavoro di Jason (plugin jEdit) , mentre il secondo in figura 1.2 mostra il Jason Mind Inspector.

Alcuni link utili:

- http://jason.sourceforge.net/JasonWebSite/Jason Home.php, homepage del sito di Jason
- http://jason.sourceforge.net, pagina in cui è disponibile Jason
- http://jasonplugin.wikidot.com/, pagina per scaricare il plugin di Jason per Eclipse
- http://jason.sourceforge.net/api/, pagina dove si trova la documentazione API di Jason

[1]Un integrated development environment (IDE), in italiano ambiente di sviluppo integrato, (conosciuto anche come integrated design environment o integrated debugging environment, rispettivamente ambiente integrato di progettazione e ambiente integrato di debugging) è un software che aiuta i programmatori nello sviluppo del codice.

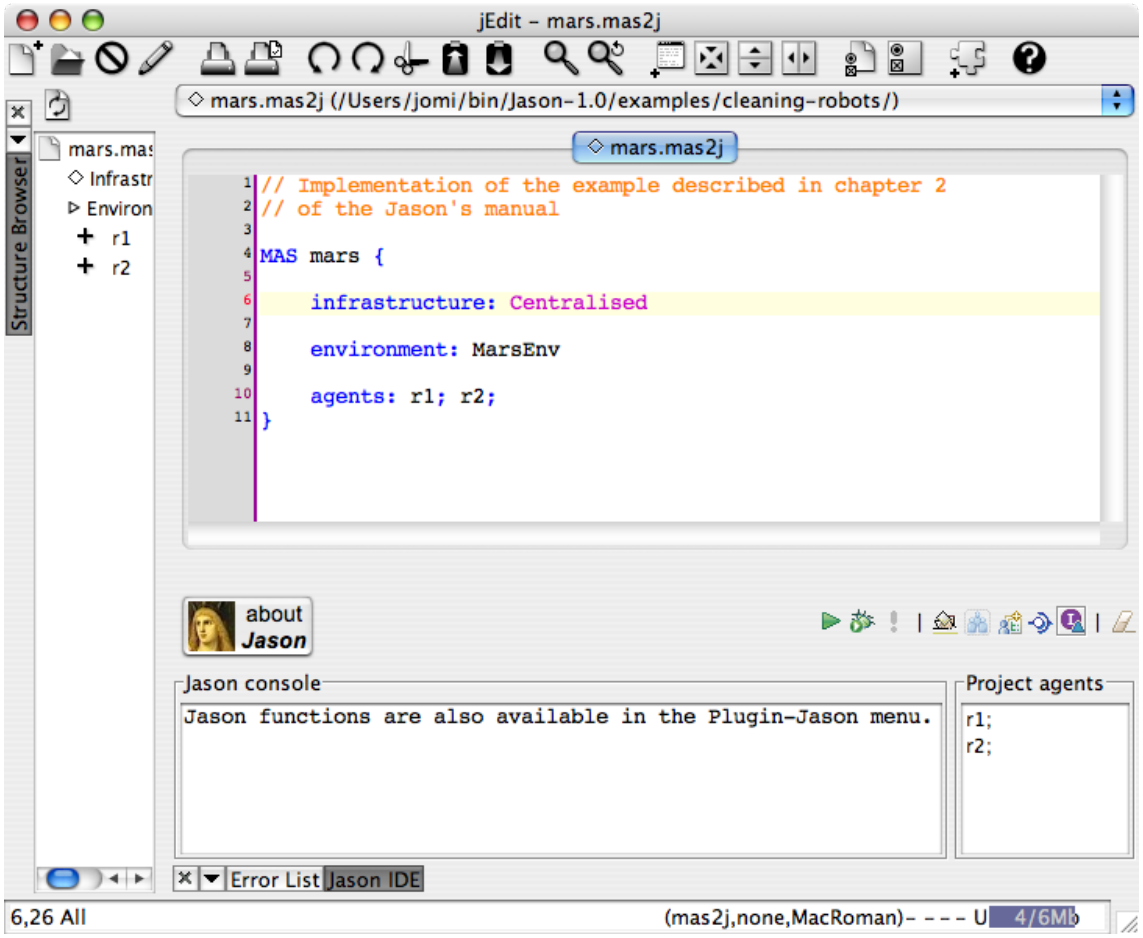

Figura 1.1: Ambiente di lavoro di Jason.

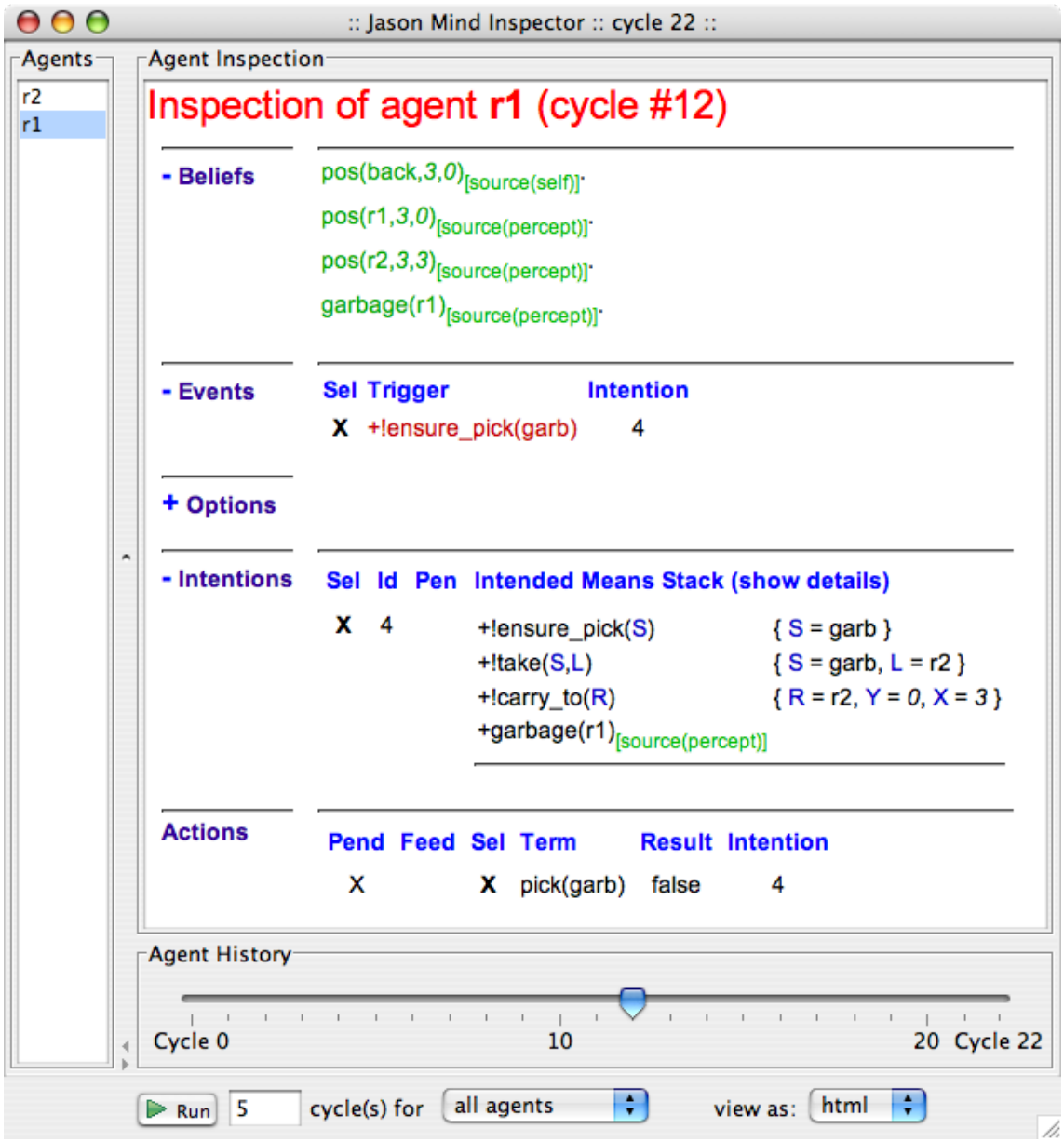

Figura 1.2: Ambiente di debug di Jason.

1.3 Percorso della tesi

La tesi è strutturata in modo tale da introdurre gradualmente il lettore nell'ottica dell'agent oriented, a partire dai concetti di base fino ad arrivare ad un livello di conoscenza che gli permette di riflettere sulle sue possibili applicazioni future.

L'approccio è stato seguire questa semplice relazione:

$$agente = programma + architettura$$

- *programma*, si è studiato il linguaggio di programmazione ad agenti (capitolo 1);
- *architettura*, si è scelto di utilizzare un robot datoti di sensori, perciò si sono studiate la basi della robotica (capitolo 2);
- *agente*, avendo acquisito le conoscenze dai capitoli precedenti (parte compilativa), si descrive come si può mappare il concetto di robot su quello di agente e possibili applicazioni (parte sperimentale) (capitoli 3 e 4).

L'obiettivo della tesi è *programmare* un *agente* attraverso **Jason** per fargli *svolgere attività complesse*.

Capitolo 2

I sistemi multi-agente

2.1 L'architettura ad agenti

Oggi siamo abituati a pensare ad un programma come una esecuzione di un processo di elaborazione nato per realizzare una certa funzionalità.

Un programma ha una semplice struttura:

- accetta un input,
- effettua una elaborazione,
- produce un output.

Sfortunatamente non tutti i programmi sono realizzabili con questa semplice struttura. In particolare alcuni sistemi hanno la necessità di una iterazione continua con l'ambiente che ne determina il comportamento, essi sono detti reattivi. É risaputo, però, che questi sistemi sono più difficili da gestire e meno efficienti rispetto ai programmi (o sistemi) funzionali.

Una classe di sistemi ancora più complessi, sottoinsieme dei sistemi reattivi, sono gli *agenti*. La maggiore complessità è dovuta al fatto che gli agenti hanno un certo grado di *indipendenza* dato che ad essi può essere delegata l'esecuzione di compiti perché capaci di determinare il miglior modo per realizzarli. Agente, quindi, nel senso di un sistema che sa agire ovvero scegliere individualmente le azioni significative per portare a termine un determinato obiettivo.

2.1.1 Caratteristiche

Gli agenti si contraddistinguono perché hanno le seguenti caratteristiche:

- autonomia;
- proattività;
- reattività;
- abilità sociali.

Autonomia : In generale si possono distinguere vari livelli di autonomia da nulla a completa. Un programma con autonomia nulla è definito tale in quanto esso entra in azione solamente per soddisfare le richieste compiute dall'utente come ad esempio la selezione di una voce del menù oppure il click su una icona. Un programma con autonomia completa è definito tale in quanto esso è in grado di prendere iniziative eseguendo azioni senza nessuna richiesta. Si studieranno i programmi che si posizionano in un livello intermedio di autonomia in quanto ritenuti di maggiore interesse per il contesto in cui si opera. Come già precedentemente anticipato, si vuole essere capaci di *delegare* l'esecuzione di compiti agli agenti e che questi ultimi, in modo autonomo, siano in grado di individuare le azioni migliori per realizzarli. L'approccio dell'agente per la gestione dei compiti è la generazione di *piani* e conseguentemente l'adempimento agli obiettivi in essi contenuti. Nel caso in cui, nello stesso tempo, l'agente abbia più piani da realizzare esso compone un piano complessivo per portare a termine i suoi obiettivi e sotto-obiettivi, altresì detti goals e sotto-goals. L'autonomia dell'agente è, dunque, nella scelta e nella gestione degli obiettivi essendo le decisioni per compiere tali azioni proprie, senza nessuna guida o controllo.

Proattività : L'agente esegue una serie di azioni che producono un comportamento atto a realizzare il particolare obiettivo delegato in quel momento. Nel caso in cui ci siano uno o più obiettivi da realizzare un agente non dovrà mai rimanere passivo, in attesa di istruzioni, ma è sempre esso a prendere l'iniziativa per portarli a termine.

Reattività : Essere *reattivo* significa essere *pronto a rispondere* ai cambiamenti dell'ambiente. Nella vita quotidiana è raro che i nostri propositi procedano senza intoppi. Possiamo essere ostacolati da altre cose o persone in modo accidentale o intenzionale. Quando noi ci rendiamo conto che i nostri piani stanno fallendo, noi rispondiamo scegliendo un percorso di azioni alternativo. Possiamo, quindi, distinguere due momenti successivi. Il primo consiste nel riuscire a capire, attraverso la percezione, se l'ambiente è adatto alla realizzazione dei goals. Il secondo consiste nel riuscire a modificare il piano per la realizzazione dei goals nel nuovo scenario dell'ambiente. Creare un sistema che compia uno solo dei due passi risulta semplice. Creare, viceversa, un sistema che compia entrambi i passi risulta estremamente difficile ed è proprio questo uno degli obiettivi del linguaggio di programmazione ad agenti.

Abilità sociali : Oggi milioni di computer distribuiti in tutto il mondo sono collegati fra loro e scambiano informazioni con persone o altri computer. L'abilità che usano può essere definita come scambio di bytes e non è il genere di scambio che si vuole effettuare con gli agenti. L'abilità che si richiede agli agenti consiste nella *cooperazione* e nella *coordinazione* di attività con i suoi simili per la realizzazione dei goals. Gli agenti devono essere in grado di comunicare agli altri le loro *credenze*, i loro *obiettivi* e i loro *piani*. In altre parole, ciò che si vuole ottenere, è una comunicazione non a livello di bytes bensì a *livello di conoscenza*.

2.1.2 Il sistema multi-agente

Si considerano essere agenti sistemi che sono *situati* in un *ambiente* (fisico o virtuale). Questo significa che l'agente deve essere in grado di percepire l'ambiente (attraverso sensori) e avere un repertorio di possibili azioni che esso potrà compiere (attraverso attuatori) per modificare lo scenario circostante.

La chiave della questione è rivolta a come l'agente decide l'azione da compiere in base alle informazioni ottenute attraverso i sensori. Possiamo pensare che tali decisioni siano prese traducendo le informazioni a un meccanismo esterno all'agente tramite una sorta di API.

L'agente, comunque, è in grado di avere solo un controllo parziale sull'ambiente, spesso a causa della presenza di *altri* agenti che operano nello stesso contesto. Nella maggior parte dei casi, infatti, gli ambienti sono abitati da più agenti creando un *sistema multi-agente*.

Ogni agente ha una sfera di influenza nel suo ambiente che può essere unica o sovrapposta con altri. Il secondo caso rende la vita degli agenti più complicata perché questi per realizzare un obiettivo devono tenere conto anche delle probabili azioni degli altri che hanno la stessa sfera di controllo.

Gli agenti situati in uno stesso ambiente possono avere dei rapporti organizzativi fra loro. Questi rapporti possono delinearsi come paritari o no, ovvero tutti gli agenti hanno una pari condizione oppure uno può avere una posizione di autorità sugli altri.

2.2 Il modello BDI ad agenti

Uno degli aspetti più interessanti del linguaggio di programmazione ad agenti o AgentSpeak è che consiste in un modello basato sul comportamento umano sviluppato da filosofi. Questo modello è chiamato *Belief - Desire - Intention* o brevemente BDI.

La prima idea di cui abbiamo bisogno per descrivere tale modello è pensare che il programma abbia una sorta di stato mentale. Il programma possiede, quindi, credenze, desideri ed intenzioni.

Le credenze sono informazioni che l'agente ha sul mondo. Esso considera tali vere ma in realtà potrebbero non esserlo perché imprecise o datate.

I desideri sono tutti i possibili stati di eventi che l'agente *potrebbe* realizzare. Nel momento in cui un agente ha un desiderio, non significa implicitamente che esso deve concretizzarlo ma che le sue azioni future saranno condizionate da esso. Ci accorgiamo che questo è perfettamente ragionevole per un agente in possesso di desideri mutuamente incompatibili fra loro. Si può pensare che i desideri siano delle *alternative* (options) per l'agente.

Le intenzioni sono gli stati di eventi che l'agente ha deciso di realizzare e lavorerà per realizzarli. Esse possono essere goals che sono stati delegati all'agente o possono essere il risultato di una scelta fra i desideri. In particolare, solo i desideri che l'agente decide di concretizzare diventano intenzioni.

2.2.1 Il ragionamento pratico

Il particolare modello di prendere le decisioni nel BDI è conosciuto come il *ragionamento pratico*. Questo processo di valutazione produce in un agente la scelta di intenzioni e di queste ultime possiamo elencare alcune proprietà.

Il ruolo più evidente delle intenzioni è di essere *pro-attitudinali* ovvero tendono a guidare il corso delle azioni dell'agente.

La seconda proprietà delle intenzioni è di essere *persistenti*. Nel caso in cui l'agente segua una intenzione, esso insisterà a svolgerla finché non riesce a realizzarla.

La terza è che una volta adottata una intenzione, questa influenza il ragionamento pratico futuro. Un'altra caratteristica prevedibile del ragionamento è che devono essere escluse tutte le intenzioni contrarie a quella scelta.

In altre parole, le intenzioni sono strettamente collegate alle credenze relative al proprio futuro. Questo perché nel momento in cui l'agente sceglie di realizzare una determinata intenzione, non verranno prese in considerazione tutte le opzioni che sono *inconsistenti* con essa. Viene generato, quindi, un cosiddetto "filtro di ammissibilità" che riduce le possibili intenzioni future che l'agente potrà considerare sulla base di quella scelta. L'agente crede che l'intenzione prediletta in circostanze normali si realizzi ma niente impedisce che essa possa *fallire*.

2.2.2 Analisi mezzi-fini

L'analisi mezzi-fini è il processo di decisione atto a realizzare un fine usando i mezzi a disposizione. L'idea di base è che un programmatore sviluppa parte di un piano per un agente in modalità off-line. Il compito dell'agente è successivamente di assemblare questi piani parziali per l'esecuzione runtime, essendo così in grado di affrontare qualunque obiettivo per cui sta lavorando. A prima vista questo approccio potrebbe sembrare una semplice variazione dei principi di progettazione già esistenti, in quanto invece di assemblare azioni vengono assemblati piani. Inoltre, potrebbe apparire una mancanza di flessibilità riguardante l'abilità di un agente di affrontare le circostanze perché potrebbe risultare dipendente dai piani codificati dal programmatore dell'agente. Tuttavia, l'esperienza dimostra che questo approccio dà buoni risultati ed è, perciò, stato scelto per l'implementazione in AgentSpeak.

Per capire meglio di cosa tratta il ragionamento pratico, di seguito viene riportato un esempio di ciclo di controllo svolto da un agente. L'agente in modo continuo:

- osserva il mondo e aggiorna le credenze su questa base;
- stabilisce quale intenzione realizzare;
- utilizza il ragionamento pratico per cercare un piano per portare a termine l'intenzione scelta;
- esegue il piano.

Il problema di questo ciclo di controllo si identifica nel fatto che una volta scelta una intenzione e il relativo piano per portarla a termine, l'agente lavora finché non lo realizza. Questo significa che l'agente continua a cercare di realizzare una intenzione che potrebbe diventare in un secondo momento inutile o addirittura irrealizzabile. Una soluzione è stata di introdurre un controllo attraverso il quale l'agente riesce a giudicare le intenzioni realizzabili e continui a lavorare solo per quelle.

2.3 Il ragionamento procedurale

Un approccio totalmente differente dai principi di progettazione già esistenti è il ragionamento procedurale, Procedural Reasoning System o brevemente PRS. Questi sistemi sono muniti di librerie di piani pre-compilati che sono stati costruiti precedentemente dai programmatori dell'agente. Tutti i piani in PRS hanno i seguenti componenti:

- un *obiettivo*, la post-condizione del piano;
- un *contesto*, la pre-condizione del piano;
- un *corpo*, la ricetta del piano ovvero la linea d'azione da seguire.

Gli obiettivi di un piano PRS definiscono cosa è significativo per il piano, ad esempio i risultati che vuole ottenere.

Il contesto del piano definisce cosa deve essere verificato nell'ambiente, di modo che il piano possa essere eseguito con successo.

Il corpo di un piano PRS è leggermente diverso da quello visto nel modello BDI.

Nel ciclo di controllo del BDI si sono descritti i piani come una modalità di azione, una semplice lista di azioni eseguite in ciclo. Tali piani sono realizzabili anche nei sistemi di ragionamento procedurale, anzi questi ultimi ne accettano una gamma molto più ampia.

La prima più evidente differenza è che come il BDI ha le azioni individuali primitive in qualità di componenti base dei piani, il PRS ha, invece, gli obiettivi, chiamati anche goals. Dal momento in cui, in un piano viene incluso un particolare goal in un punto, significa che quel goal deve essere realizzato in quella fase prima che il resto del piano venga eseguito.

In principio, un agente PRS ha una collezione di piani, alcune credenze iniziali sul mondo e un obiettivo che deve essere realizzato che verrà inserito in uno stack apposito chiamato *stack delle intenzioni*. Questo stack contiene tutti gli obiettivi che sono in attesa di essere realizzati. L'agente quando cerca nella sua libreria un piano, vede i piani che si trovano in cima allo stack delle intenzioni. Di questi piani vengono selezionati solo quelli aventi le pre-condizioni soddisfatte, ovvero in accordo con le correnti credenze dell'agente. L'insieme di piani che realizzano l'obiettivo e hanno le loro pre-condizioni soddisfatte, diventano possibili alternative per l'agente.

Ci sono diversi modi di scegliere quale alternativa completare nelle architetture PRS. Nel PRS originario, la decisione su quale realizzare veniva presa sulla base di piani ad un meta-livello. Posizionandosi in tale *meta-livello*, si è in grado di modificare le strutture delle intenzioni dell'agente a run-time, in modo da spostare l'attenzione del ragionamento pratico dell'agente. Un semplice metodo, comunque, è di usare *utilità* per i piani. Ad esempio un'utilità potrebbe essere un valore numerico, l'agente sceglie il piano che ha il valore più alto.

Il piano scelto viene eseguito in un certo momento, eliminato dallo stack e in quest'ultimo, eventualmente, ne verrà inserito uno nuovo. L'agente ora dovrà stabilire quale sarà il nuovo piano che deve realizzare secondo le modalità precedentemente descritte. Nel caso in cui un obiettivo del piano eletto fallisca, l'agente è capace di selezionare un differente piano che realizzi quell'obiettivo dall'insieme dei piani candidati.

Il linguaggio AgentSpeak rappresenta un tentativo di estrarre le caratteristiche chiave del PRS in un semplice e unico linguaggio di programmazione da utilizzare anche sui sistemi di larga scala.

2.4 Il linguaggio di programmazione ad agenti Jason

Un agente è caratterizzato da una architettura che deriva dal modello BDI. L'architettura si compone di credenze, obiettivi ed intenzioni. Il *programma ad agenti* definisce il comportamento specifico dell'agente ed è ciò che viene mandato in esecuzione sull'architettura. Il linguaggio interpretato da **Jason** è una estensione di AgentSpeak, linguaggio basato sull'architettura BDI. Questo è visibile da una molteplicità di aspetti, come, ad esempio, dalla gestione delle credenze, degli obiettivi e dei relativi piani. I comportamenti nell' *architettura ad agenti* vengono determinati in base alle credenze. Infatti, una funzione fondamentale dell'interprete sarà percepire l'ambiente, in modo da poter conseguentemente aggiornare la propria base delle credenze.

Di rilievo è, anche, l'obiettivo dell'agente, che viene realizzato tramite l'esecuzione di piani che hanno la caratteristica di essere *reattivi*.

L'interpretazione di un programma ad agenti impiega operativamente il *ragionamento ciclico* di un agente. Il ragionamento viene svolto in accordo con i piani che gli agenti possiedono nella libreria apposita. Inizialmente, questa libreria conterrà esclusivamente i piani inseriti dal programmatore, come in un programma AgentSpeak.

2.4.1 Credenze (Beliefs)

La prima cosa da sapere sugli agenti AgentSpeak è come vengono rappresentate le loro credenze. Un agente possiede una *base di credenze*, che nella sua forma più semplice può essere vista come una collezione di letterali. Nei linguaggi di programmazione ispirati a quelli logici, l'informazione si rappresenta nella forma simbolica di *predicati*. Un letterale è un predicato o la sua negazione.

L'informazione può esprimere una particolare proprietà di un oggetto o individuo oppure una relazione tra due o più oggetti.

In questo contesto si fa riferimento alla *modalità di verità* espressa nella letteratura logica e non in quella classica. Nella letteratura classica le dichiarazioni risultano vere in termini assoluti. L'agente, invece seguendo la letteratura logica, ciò che attualmente possiede nella base delle credenze

lo crede vero, ma non è detto che lo sia, anzi potrebbe essere addirittura il contrario.

2.4.2 Annotazioni

Una importante differenza nella rappresentazione sintattica delle formule logiche in **Jason**, rispetto al linguaggio di programmazione tradizionale, è l'uso delle *note* o *annotazioni*. Esse vengono viste come ulteriori informazioni e per sottolineare ciò, vengono inserite immediatamente in successione ad un letterale e racchiuse tra parentesi quadre.

Si presentano due vantaggi nell'uso delle note in **Jason**. Il primo è semplicemente l'eleganza delle annotazioni. Questo aspetto potrebbe essere irrilevante logicamente, ma per molti programmatori è di grande rilievo. Il fatto che il dettaglio su una particolare credenza sia organizzato con tale ordine, mette in luce il collegamento con essa, rendendo la base delle credenze più leggibile. Il secondo è la facilità di gestione della base delle credenze. Un programmatore, se lo desidera, può con un qualsiasi programma Java modificare sia la base delle credenze, sia la funzione di aggiornamento di quest'ultima, per adattarle alle esigenze del contesto in cui sta lavorando.

L'annotazione che specifica come sorgente l'agente stesso è il motivo della loro creazione, poiché principale fonte di informazioni.

Le sorgenti di informazioni, per gli agenti nei sistemi multi-agente, si classificano in tre tipologie:

informazioni percepibili : un agente acquisisce alcune credenze come conseguenza di sensazioni ovvero rappresentazioni simboliche di proprietà dell'ambiente chiamate *percezioni*;

comunicazione : la modalità di comunicazione degli agenti fra di loro, in un sistema multi-agente, introduce la necessità di rappresentare le informazioni fornite da un altro agente, inoltre, spesso è utile sapere individuare da quale agente sono state acquisite le informazioni;

note mentali : avvenimenti successi nel passato utili per affrontare circostanze future.

Un'altra caratteristica di **Jason** è, quindi, specificare in modo esplicito le sorgenti delle informazioni presenti nella base delle credenze. Per le credenze acquisite da informazioni percepite, l'interprete automaticamente ag-

giunge come sorgente la percezione dell'agente. In altre annotazioni, potrebbe comparire come sorgente l'agente stesso oppure il nome di un altro agente. L'ultimo caso si presenta quando le informazioni sono state apprese tramite un messaggio di comunicazione di credenze da un altro agente. Inoltre, si potrebbe pensare di aggiungere oltre al nome, una valutazione sull'affidabilità dell'agente per determinare quanto siano attendibili le informazioni ricevute. L'agente, quindi, si può annotare tutto quello che crede significativo.

2.4.3 Negazioni

La negazione causa varie difficoltà nei linguaggi di programmazione logica. Un approccio diffuso per gestirla è usare la *closed world assumption* e la *strong negation*.

La closed world assumption può essere vista come una strategia dove qualcosa che né è conosciuto per vero e né risulta ottenibile da realtà conosciute usando le regole nel programma, viene assunto falso. A tale scopo viene utilizzato l'operatore `not`.

La *strong negation* è usata per esprimere che un agente *crede esplicitamente che qualcosa sia falso*. In questo caso viene inserito l'operatore '$\sim$' davanti al predicato.

2.4.4 Regole

Le regole permettono agli agenti di dedurre nuovi aspetti sulla base delle credenze da loro possedute.

Una regola viene suddivisa in due parti da un operatore "`:-`". A sinistra dell'operatore, può esserci solo un letterale. Quest'ultimo viene eseguito solo se la condizione a destra è soddisfatta. Naturalmente, il tutto viene svolto secondo le credenze correnti dell'agente. Inizialmente, le credenze correnti saranno quelle inserite dal programmatore nel codice sorgente. Il codice sorgente, infatti, indica all'interprete quali credenze devono esserci in origine nella base delle credenze dell'agente, ovvero quando quest'ultimo diventa attivo per la prima volta. Dopo che l'agente le ha acquisite, può iniziare a percepire l'ambiente. A meno che non sia specificato diversamente, tutte le credenze, che appaiono nel codice sorgente dell'agente, sono assunte come annotazioni mentali.

2.4.5 Obiettivi (Goals)

Nella programmazione ad agenti, la nozione di *obiettivo* o *goal* è fondamentale. Considerato che le credenze, in particolare quelle di una sorgente percettiva, esprimono proprietà che sono credute vere del mondo nel quale l'agente è collocato; gli obiettivi rivelano le caratteristiche degli stati del mondo che l'agente si prefigge di realizzare. In altre parole, quando si rappresenta un goal in un programma ad agenti, significa che l'agente è impegnato ad agire per cambiare lo stato del mondo da quello in cui crede, in base alle percezioni, a quello desiderato.Questo particolare impiego dei goals nella letteratura della programmazione ad agenti prende il nome di *declarative goal*.

In AgentSpeak, ci sono due tipi di obiettivi: *achievement goals* e *test goals*. Gli "achivement goals" vengono denotati dall'operatore "!" ed includono i "declarative goals". Il fatto che un agente possa scegliere un nuovo obiettivo, conduce all'esecuzione di piani. I piani sono essenzialmente modalità di azione che l'agente suppone portino al successo di un determinato obiettivo.

I "test goals" sono utilizzati per verificare se l'agente crede ad un letterale o ad una unione di questi ultimi, è denotato dall'operatore "?". Nella maggior parte dei casi, tali obiettivi sono usati semplicemente per recuperare informazioni che sono disponibili nella base delle credenze dell'agente.

Le regole in AgentSpeak possono essere utilizzare nella base delle credenze dell'agente per dedurre una risposta agli obiettivi test.

Nel codice sorgente possono essere inseriti, oltre alle credenze iniziali, anche dei goals che l'agente deve tentare tentare di soddisfare subito dopo l'avvio. Questi obiettivi sono chiamati "goals iniziali".

2.4.6 Piani

Un piano AgentSpeak ha tre parti distinte:

- *triggering event* (l'evento scatenante);
- *context* (il contesto);
- *body* (il corpo).

Insieme, l'evento innescato e il contesto sono chiamati la *testa* del piano.

Le tre parti del piano sono separate sintatticamente dagli operatori ":" e "<- " nel modo come segue:

triggering event : context <-body

Si spiegheranno, in seguito, nel dettaglio ognuna delle tre parti. Adesso, brevemente, si descrive l'idea di base per ognuna di esse:

Triggering event. Un agente possiede, fra le altre, due caratteristiche: reattività e proattività. Queste, per l'agente, sono strettamente collegate e viste come aspetti importanti del suo comportamento. L'agente ha obiettivi che cerca di realizzare nel lungo termine, così da determinare il suo comportamento pro-attivo. Tuttavia, gli agenti mentre operano per realizzare i loro obiettivi, necessitano di stare attenti ai cambiamenti nel proprio ambiente. Il motivo è che queste modifiche forniscono un riscontro sul successo effettivo dei loro obiettivi e sul grado di efficienza con cui stanno agendo. Un cambiamento dell'ambiente può anche significare che ci siano nuove opportunità di approccio agli scopi per l'agente e forse occasioni per considerare l'adozione di nuovi obiettivi che precedentemente non sono stati adempiuti o di obiettivi esistenti che per vari motivi sono stati abbandonati.

Per gestire tale situazione, esistono due tipi di modifica: *cancellazione* ed *inserimento*. Queste modifiche possono essere apportate sia nelle credenze, sia negli obiettivi dell'agente. I cambiamenti in entrambi i tipi di attitudini (credenze e obiettivi) creano, nell'architettura dell'agente, eventi che dovranno essere gestiti dall'agente. In questa ottica, i piani sono una modalità di azione che l'agente si impegna ad attuare di conseguenza a tali cambiamenti, ovvero nuovi *eventi*.

Il triggering event ha lo scopo di informare l'agente, su quali siano gli eventi specifici per i quali il piano deve essere usato. Se un evento che si verifica, coincide con l'evento scatenante di uno dei piani presenti nella libreria dei piani dell'agente, quel piano diventa rilevante per quel particolare evento.

Context. Il contesto di un piano è un importante aspetto della programmazione dei sistemi reattivi. Si è visto che gli agenti possiedono obiettivi e piani usati per realizzarli, continuando a stare attenti ai cambiamenti nell'ambiente. Gli ambienti dinamici sono complicati da trattare a

causa delle modifiche nell'ambiente, le quali possono influenzare sia le azioni del futuro, sia la realizzazione dei piani a causa della mutata possibilità di portare a termine gli obiettivi.

Questo è il motivo per cui la pianificazione dei sistemi reattivi prevede che la scelta del piano per la realizzazione dell'obiettivo sia realizzata il più tardi possibile, in modo che l'agente abbia informazioni più recenti, perciò più attendibili, sul mondo che lo circonda, e da cui deriva una maggiore probabilità di successo. Di conseguenza, la scelta del piano per uno dei vari obiettivi viene compiuta da un agente solo quanto sta iniziando ad eseguirlo. Nel processo di scelta del piano entra in gioco il contesto, poiché un agente avrà diversi piani per realizzare lo stesso obiettivo.

Il ruolo del contesto è di controllare la situazione corrente così da determinare se un particolare piano, fra le varie alternative, è possibile che riesca a gestire l'evento (cioè realizzare un obiettivo) date le ultime informazioni in possesso dell'agente sull'ambiente. Di conseguenza, un solo piano viene scelto per essere eseguito a condizione che il suo contesto sia una *logica conseguenza* delle credenze dell'agente. Un piano che ha un contesto valutato come vero in base alle credenze correnti dell'agente è ritenuto *applicabile* in questo momento, perciò è candidato per l'esecuzione.

Body. Il corpo è visto come una sequenza di formule determinanti una modalità di azione. In altre parole, quello che l'agente compie per la gestione dell'evento che ha scatenato il piano. Un importante concetto che appare nel corpo del piano è quello di un *goal*, che permette di sapere quali sono i (sotto)goals che l'agente dovrà portare a termine e che devono essere realizzati in ordine per far sì che il piano, per la gestione dell'evento, abbia successo. Il termine *sotto-goal* è riferito ad una circostanza, in cui un piano vede un altro piano come strumento per realizzare un particolare obiettivo.

Ora si introducono nel dettaglio le varie parti del piano.

Triggering event.

Come si è visto in precedenza, esistono due tipi di attitudini mentali: credenze e obiettivi. Questi ultimi si dividono in due categorie: "test goal" e

"achievement goal". Gli eventi rappresentano variazioni per l'inserimento o la cancellazione di elementi nelle credenze o negli obiettivi.

La combinazione tra credenze e obiettivi con le possibilità di variazioni permettono di esprimere sei differenti tipi di eventi innescati per un dato letterale l. I sei tipi di "triggering events" sono:

- belief addition (+l);
- belief deletion (-l);
- achievement-goal addition (+!l);
- achievement-goal deletion (-!l);
- test-goal addition (+?l);
- test-goal deletion (-?l).

Gli eventi per l'aggiunta, "belief addition" , o la cancellazione di una credenza, "belief deletion", si presentano, per esempio, quando l'agente aggiorna le sue credenze conformemente alle sue percezioni dell'ambiente, solitamente ottenute in ogni ciclo di ragionamento.

Gli eventi originati nell'agente da nuovi obiettivi, "addition goals", sono per lo più una conseguenza dell'esecuzione di altri piani oppure frutto di una comunicazione fra agenti.

I tipi di eventi per la cancellazione di obiettivi, "deletion goals" , sono usati solitamente per gestire il fallimento di un piano.

Context.

Un contesto può essere visto semplicemente come una unione di *default literals* e di espressioni relazionali. Un "literal" è un predicato su alcuni stati del mondo. Inoltre, solitamente un "default literal" può opzionalmente avere diversi tipi di negazione, quella definita "strong negation" e un'altra conosciuta in programmazione logica come *default negation*, che è denotata attraverso l'operatore "`not`".

L'unione dei due tipi di negazione fa nascere quattro combinazioni dei letterali (dato il letterale l):

- l'agente crede che l sia vero (l);

- l'agente crede che l sia falso ($\sim l$);
- l'agente non crede che l sia vero (`not` l);
- l'agente non crede che l sia falso (`not` $\sim l$);

Da notare che, per l'agente non credere che l sia vero (falso) non significa che l sia falso (vero), perché l'agente potrebbe semplicemente non conoscere nulla riguardo ad l.

Le espressioni logiche possono presentarsi in un contesto di un piano come combinazione di letterali con gli operatori:

- and (congiunzione) denotato da "`&`",
- or (disgiunzione) denotato da "|",
- la negazione denotata, come visto in precedenza, da "`not`".

Tale sintassi può essere usata nel corpo delle regole, che sono parte della base delle credenze.

Body.

Il corpo di un piano definisce una modalità di azione, che l'agente intraprende quando: un evento corrispondente all'evento scatenato di un piano avviene ed il contesto di quel determinato piano risulta vero in accordo con le credenze dell'agente, date queste due condizioni soddisfatte, viene selezionato come il piano da eseguire. La modalità di azione è rappresentata come una sequenza di formule, ognuna separata dall'altra da "`;`".
Ci sono sei differenti tipi di formule che possono apparire nel corpo di un piano:

- *azioni*;
- *achievement goals*;
- *test goals*;
- *note mentali*;
- *azioni interne*;

- *espressioni*;

Azioni. Una delle più importanti caratteristiche di un agente è quella di essere capace di *agire* dentro un ambiente. Un tipo di formula, trovata nella maggior parte dei piani, è l'azione ovvero l'attività che l'agente è in grado di compiere. Qualora si progettasse un robot, si devono conoscere le azioni che l'agente è capace di eseguire dato l'hardware che ha a disposizione.In seguito, l'architettura complessiva dell'agente interfaccerà le rappresentazioni simboliche di queste azioni, ovvero il modo con cui ci si riferisce ad esse, con gli esecutori delle azioni, ovvero chi via hardware o via software compie le azioni.

Il linguaggio costruito per riferirsi completamente a tali azioni è detto *ground predicate*. In esso, si necessita che prima dell'esecuzione di una azione, tutte le variabili utilizzate vengano precedentemente istanziate.

Qualcuno potrebbe rimanere stupito di come l'interprete riesce a distinguere le azioni dagli altri usi dei predicati. In realtà, basta semplicemente osservare la posizione dei componenti dentro il piano. Per esempio, i predicati collocati nel contesto del piano saranno controllati in rapporto alle credenze dell'agente, piuttosto che essere visti come azioni. Nel corpo del piano, invece, un semplice predicato sarà interpretato come una azione.

Le azioni vengono eseguite da esecutori dell'agente, che non sono parte dell'interprete AgentSpeak. Di conseguenza, ha senso aspettarsi un certo genere di "feedback" o risposta di ritorno sull'esito della esecuzione dell'azione. Questa risposta di ritorno dimostra, tramite un valore booleano, se per l'esecutore l'azione era impossibile da eseguire (false) e in tale caso il piano fallisce o, viceversa, se è stata eseguita (true). Nel momento in cui un piano durante l'esecuzione richiede un feedback, deve essere *sospeso* finché non verrà a conoscenza dell'avvenuta, o no, esecuzione dell'azione. Avere eseguito una azione e quindi avere ricevuto il feedback "true", non implica necessariamente che siano avvenuti nell'ambiente i cambiamenti attesi. Risulta fondamentale, per questo motivo, avere riscontri concreti continuando a percepire l'ambiente e controllando le credenze successive, così da essere certi del procedere dell'esecuzione.

Achievement Goals. Avere obiettivi da realizzare è una altra essenziale caratteristica di un agente. Il comportamento di un agente è definito complesso, in quanto implica il conseguimento di compiti come requisito (sotto-obiettivo) per adempiere ad una azione e, per questo, non può essere visto come una semplice sequenza lineare di azioni da eseguire. La strategia predefinita è di includere un obiettivo in un corpo di un piano, operare per portarlo a compimento e solo dopo che è stato eseguito con successo riprendere la modalità di azione sospesa. Nel caso in cui, non ci sia la necessità di aspettare la conclusione del goal che si sta eseguendo, si può proseguire con la corrente modalità di azione. Questo può essere espresso anteponendo al letterale che indica l'obiettivo l'operatore "`!!`".

Test Goals. Gli obiettivi test o "test goals" sono normalmente usati per recuperare informazioni dalla base delle credenze o per verificare se qualcosa di atteso è davvero creduto dall'agente durante l'esecuzione del corpo di un piano, ma potrebbero essere impiegati anche per compiti più complessi. Potrebbe risultare lecito domandarsi perché non si utilizzi il contesto del piano per ottenere tutte le informazioni necessarie, prima di iniziare l'esecuzione del piano stesso, così da averle a disposizione in futuro. Concettualmente, il contesto del piano dovrebbe essere usato solo per determinare se il piano è applicabile. Durante l'esecuzione di un piano, non è detto che le informazioni utilizzate nel contesto del piano rimangano invariate, anzi è più probabile che si modifichino data la natura dinamica dell'ambiente. Per questo motivo, è necessario che un agente possieda le informazioni il più possibile aggiornate perché se obsolete potrebbero provocare il fallimento di un piano.

Tali informazioni possono essere ricavate attraverso l'uso di un test goal. Nel caso in cui il test goal recupera le informazioni dalla base delle credenze, esso viene concluso positivamente. Se, viceversa, queste informazioni non sono recuperate perché non incluse nelle credenze, prima che il piano fallisca (come conseguenza del fallimento del test goal), l'interprete **Jason** tenta di creare un evento il cui scopo è di trovare un piano che includa un triggering event per risolvere il test goal. Se nella libreria dei piani non viene trovato nessun piano adatto a gestire il test goal, allora avviene il fallimento del test goal

e, di conseguenza, del piano. Potrebbe essere lecito pensare, inoltre, di recuperare informazioni da un piano servendosi di un achievement goal invece di un test goal. In entrambi i goals, una azione dovrà includere delle variabili non istanziate. L'introduzione di un achievement goal impone che le variabili siano legate all'esecuzione del piano associato per la gestione dell'azione. Per tale motivo viene, necessariamente, generato un evento a cui viene agganciato quel piano, che deve essere scritto appositamente. Anche l'utilizzo di un test goal impone che le variabili siano specificate. Tuttavia, nel caso in cui nella base delle credenze esiste una credenza che corrisponde al test goal, allora automaticamente la variabile viene legata all'informazione nella base delle credenze. Altrimenti, nel caso in cui nessuna credenza corrisponda, viene generato un evento a cui è possibile agganciare un piano specificato dall'interprete **Jason**.

Note mentali. Una importante considerazione quando si programma in **Jason** è di distinguere le cose che l'agente sarà in grado di percepire dall'ambiente. Si può pensare che le credenze di un agente vengono aggiornate in conseguenza dell'azione appena eseguita, però non è così.

In **Jason**, l'agente acquisisce le credenze solo dopo avere percepito l'ambiente. L'aggiornamento provoca la modifica o la creazione di credenze, processo molto utile durante l'esecuzione di un piano poiché permette di controllare la possibilità di realizzazione dello stesso. Tali credenze vengono denominate note mentali ed hanno come sorgente l'agente stesso, in modo da essere distinte nella base delle credenze dell'agente. Un agente può avere bisogno di una nota mentale per ricordare a se stesso qualcosa che esso (o un altro agente) ha compiuto nel passato, informazioni sul compito interrotto che deve necessariamente essere ripreso in seguito oppure un promessa/commissione che ha detto di portare a termine.

In alcuni casi, si potrebbe avere bisogno solamente dell'ultima istanza di una certa nota mentale nella base delle credenze. A tale scopo può essere utilizzato l'operatore `-+` che agisce rimuovendo una eventuale istanza precedente, nel momento in cui ne aggiunge quella nuova. L'operatore "`-+`" è semplicemente una notazione compatta per indicare la rimozione (`-`) di una precedente nota mentale e l'inserimento (`+`) di una nuova nota con valori aggiornati.

Il tentativo di cancellare una credenza che non esiste viene semplicemente ignorato, e non causa il fallimento del piano.

Azioni interne. La caratteristica principale delle azioni esterne è quella di modificare l'ambiente. I *cambiamenti nell'ambiente* sono operati dagli attuatori degli agenti.Le azioni interne, invece, possono essere pensate come eseguite all'interno della mente dell'agente, allo scopo di estendere il linguaggio di programmazione con operazioni altrimenti non disponibili.

Le azioni interne si distinguono da quelle esterne poiché la la loro sintassi include l'operatore ".".

Nel caso in cui, l'operazione interna sia l'utilizzo della libreria, l'operatore "." è usato per separare il nome di una libreria dal nome di una singola azione interna contenuta nella stessa. **Jason** fornisce *azioni interne standard* che implementano operazioni molto importanti per la programmazione BDI.

Una azione interna standard è denotata da un nome di libreria vuoto, un esempio è l'azione `.send` utilizzata per la comunicazione inter-agente. Tale azione interna richiede almeno tre parametri: il nome dell'agente che dovrebbe ricevere il messaggio, l'intenzione del mittente del messaggio (tale intenzione è chiamata *performative*), il contenuto del messaggio.

Espressioni. Ogni formula presente nel contesto o nel corpo di un piano viene valutata attribuendole un valore booleano.
La valutazione avviene tramite operatori logici, i più comuni sono:

- `==` , risulta vero se i termini ai lati sono uguali, viceversa nel caso opposto;
- `/==` , risulta vero se i termini ai lati sono diversi, viceversa nel caso opposto;
- `=` , utilizzato per unire i termini ai lati.

L'utilizzo di espressioni aritmetiche è consentito all'interno di espressioni relazionali e dentro letterali.

Plan labels.

La sintassi di un piano prevede l'introduzione di una specifica etichetta per ognuno denominata "plan label", vista concretamente come un nome per il piano. Tutti i piani hanno una plan label automaticamente generata da **Jason**, nel caso in cui, non ne sia stata assegnata una specifica.
Tale etichetta è importante perché, altrimenti, non si saprebbe come riferirsi al particolare piano, o istanza, che l'agente sta eseguendo in un determinato momento. Le plan labels devono essere utilizzate con astuzia, inserendo etichette significative per associare informazioni di meta-livello sui piani.

La notazione generale per i piani con una specifica etichetta è:

@label te : ctxt <- body

L'etichetta, quindi, non può avere la forma di un semplice termine bensì di un predicato. Per tale motivo, potrebbe includere anche una annotazione all'interno dell'informazione di meta-livello, come, per esempio, dei dati per aiutare l'interprete a scegliere un piano applicabile piuttosto che un altro.

2.5 Il ragionamento ciclico

Un agente opera mediante un ciclo di ragionamento, che nel caso di **Jason** può essere suddiviso in dieci passi principali. Si può paragonare il *ragionamento ciclico* al ciclo di controllo delle decisioni nel modello BDI.

La figura 2.1 mostra l'architettura di un agente **Jason** e le funzioni che sono eseguite durante il ciclo di ragionamento.
I componenti architetturali della figura si distinguono in:

- rettangoli, rappresentano lo *stato dell'agente*;
- rettangoli con vertici arrotondati e rombi, rappresentano le funzioni che possono essere personalizzate dal programmatore. In particolare i rombi denotano *funzioni di selezione* che servono a selezionare un elemento da una determinata lista;
- cerchi, rappresentano parti essenziali dell'interprete che non possono essere modificate.

Il programma AgentSpeak determina lo stato iniziale della base delle credenze, dell'insieme degli eventi e della libreria dei piani.
Le credenze del programma AgentSpeak vengono utilizzate per inizializzare

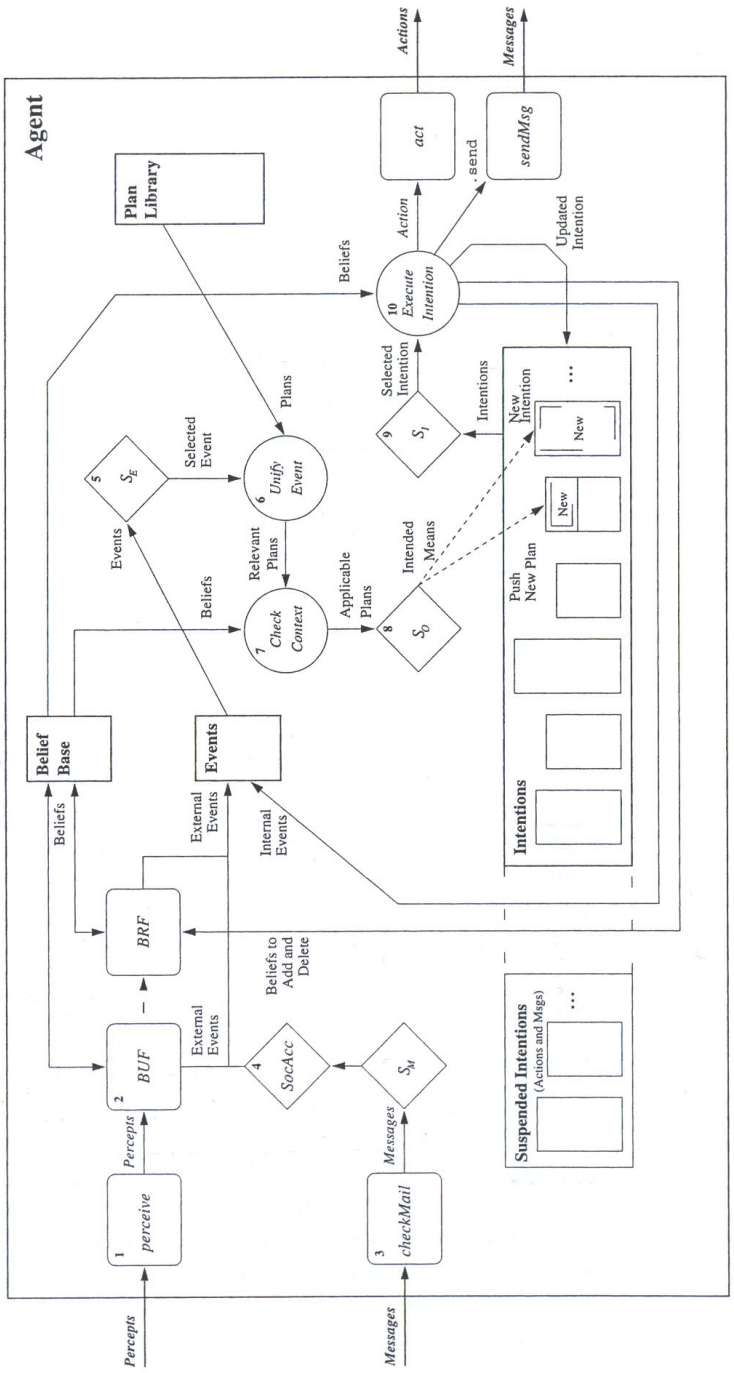

Figura 2.1: Il ragionamento ciclico di Jason.

la base delle credenze. Nell'insieme iniziale degli eventi ci saranno due tipi di eventi: i belief addition event generati dall'inizializzazione della base delle credenze ed i goal addition event provocati dall'aggiunta dei goals iniziali. I piani nel codice sorgente vanno a formare la libreria dei piani iniziale. Quando l'agente viene avviato, l'insieme delle intenzioni è vuoto.

2.5.1 I 10 passi

Ora si descrivono uno alla volta i passi del ragionamento ciclico.

1. Percezione dell'ambiente

   - La prima cosa che un agente compie, entrando nel ragionamento, è percepire l'ambiente ed aggiornare le sue credenze in base allo stato dell'ambiente. Si nota, quindi, che l'architettura ad agenti deve avere un componente capace di percepire l'ambiente e descriverlo in forma simbolica come una lista di letterali. Ogni letterale esprime una percezione, una rappresentazione simbolica dello stato corrente dell'ambiente.

     Il metodo `perceive` è usato per implementare il processo di acquisizione delle percezioni. Nella implementazione disponibile di default da **Jason**, il metodo otterrà la lista di letterali da una simulazione dell'ambiente implementata in Java.

2. Aggiornamento delle credenze

   - Una volta che la lista delle percezioni è stata ottenuta, la base delle credenze necessita di essere rinnovata dati i cambiamenti dell'ambiente. Questo è il compito della funzione di aggiornamento delle credenze, denominata *belief update function* o in breve `buf`. Il metodo `buf` di base assume che ogni cosa correntemente percepibile nell'ambiente viene inclusa nella lista delle percezioni ottenuta nel passo precedente. La strategia utilizzata per l'aggiornamento della base delle credenze risulta semplice. Dati p l'insieme delle percezioni correnti e b l'insieme dei letterali nella base delle credenze che l'agente ha ottenuto l'ultima volta che ha percepito l'ambiente, si ragiona come segue:

- ogni letterale l contenuto in p non correntemente contenuto in b è aggiunto in b;
- ogni letterale l contenuto in b non più in p è cancellato da b.

Ogni cambiamento nella base delle credenze, durante l'esecuzione di una funzione di aggiornamento delle stesse, genera un evento. In particolare, il tipo di evento che è generato perché la credenza è aggiunta o cancellata dalla base delle credenze e l'aggiornamento deriva dalla percezione dell'ambiente, viene definito *evento esterno*.

Gli eventi interni hanno un'intenzione a loro associata ed è quest'ultima a causare la generazione dell'evento. In questo modo, un evento è effettivamente rappresentato come una coppia di componenti: il primo denota il cambiamento avvenuto mentre il secondo è associato all'intenzione. Gli eventi esterni non essendo generati dalle intenzioni, vengono rappresentati aventi una *intenzione associata vuota*.

3. Ricezione comunicazioni da altri agenti

- Altre importanti sorgenti di informazioni per un agente in un sistema multi-agente sono gli altri agenti dello stesso sistema. In questo stadio del ragionamento ciclico, l'interprete controlla che i messaggi siano stati consegnati (dall'infrastruttura di distribuzione servendosi della piattaforma **Jason**) alla casella di posta dell'agente. Tale mansione è svolta dal metodo `checkMail`, che ha la possibilità di essere personalizzato. Questo metodo semplicemente prende i messaggi ricevuti ed immagazzinati dal sistema multi-agente, per poi renderli disponibili a livello dell'interprete AgentSpeak.

  All'interno di un ragionamento ciclico, solo *un* messaggio alla volta è processato dall'interprete AgentSpeak (usato da **Jason**). Nei diversi scenari, gli agenti dovranno attribuire la priorità su alcuni messaggi e per fare ciò si richiede una specifica capacità di svolgere la funzione che permette di assegnare il livello di priorità (procedura di prioritizzazione). Un'altra importante *funzione di*

selezione dei messaggi sceglie fra tutti i messaggi precedentemente ricevuti e che devono essere ancora processati, quello che verrà selezionato per essere svolto nel ciclo di ragionamento corrente. L'implementazione base di **Jason**, semplicemente, sceglie il primo messaggio nella lista ovvero, a meno che il programmatore non abbia disposto diversamente, verranno processati i messaggi in ordine di ricezione.

4. Selezione dei messaggi accettabili socialmente

   - Prima che i messaggi siano processati, essi devono soddisfare un criterio di selezione che determina se possono essere accettati dagli agenti. La funzione dell'interprete che si occupa di questo è chiamata *funzione di accettazione sociale* ed è implementata da un metodo detto `SocAcc`. Questo metodo sarà personalizzabile singolarmente per ogni agente secondo le necessità. L'implementazione di default, semplicemente, accetta tutti i messaggi da tutti gli agenti. Una particolare implementazione del metodo `SocAcc` indica da quali agenti accettare le comunicazioni sotto forma di informazioni, conoscenze o deleghe di obiettivi.

   I passi dall'1 al 4 hanno trattato dell'aggiornamento delle credenze sul mondo e sugli altri agenti. I passi dal 5 al 10, di seguito, descriveranno i passi principali dell'interpretazione dei programmi AgentSpeak.

   Il processo inizia con la base delle credenze e l'insieme degli eventi che sono stati aggiornati. Poi si prosegue con due parti importanti: la prima verte sugli eventi selezionati per essere maneggiati nel corrente ciclo di ragionamento (conducendo ad una nuova o aggiornata intenzione), la seconda sulle intenzioni dell'agente che devono essere ancora eseguite.

5. Selezione di un evento

   - Un agente BDI opera manipolando continuamente *eventi*, che rappresentano o percezioni di cambiamenti nell'ambiente o variazioni negli obiettivi dell'agente. In ogni ciclo di ragionamento,

solo un evento in sospeso può essere affrontato. Questo comporta al fatto, che ci possono essere svariati eventi in attesa, a causa di vari aspetti, come, ad esempio, di recenti cambiamenti dell'ambiente e l'agente, così, non è riuscito a fare abbastanza cicli di ragionamento per gestirli tutti e alcuni sono rimasti in sospeso. Per questo motivo, è necessario selezionare l'evento che dovrà essere maneggiato in un particolare ciclo di ragionamento. Tale mansione è svolta da ogni specifico agente attraverso la *funzione di selezione degli eventi*.

Generalmente, questa funzione di selezione è personalizzabile per poter soddisfare le singolari specifiche, una comune è l'introduzione di un livello di priorità. Solo in semplici applicazioni, dove tutti gli eventi possono essere assunti con la stessa importanza per l'agente, il programmatore potrà usare la funzione predefinita di selezione di un evento. L'insieme degli eventi è implementato in **Jason** come una lista, difatti i nuovi eventi sono aggiunti in coda. Quello che l'implementazione di default della funzione di selezione di un evento compie è, semplicemente, di selezionare il primo evento nella lista. Se il programmatore sceglie di *non* personalizzare la funzione di selezione di un evento, il componente **Evento** può perciò essere immaginato come in una coda di eventi (per esempio un struttura FIFO).

Per fare un esempio, suppongo che nell'insieme degli eventi corrente ce ne siano due. Il primo in cui l'agente percepisce una scatola di colore blu e il secondo una sfera di colore rosso. Nel caso in cui, la funzione di selezione di un evento non venga personalizzata, l'evento per la percezione della scatola blu, essendo la prima in ordine di tempo, sarà quella maneggiata nel corrente ciclo di ragionamento. In caso contrario, in cui l'agente abbia una specifica, nella quale viene indicato un particolare interesse per gli oggetti rossi e perciò, da specifica, devono essere manipolati prima degli oggetti di ogni altro colore (ovvero viene definito un livello di priorità), l'evento per la percezione della sfera rossa sarà quello inserito nel corrente ciclo di ragionamento. Per fare questo, verrà precedente implementata una precisa funzione di selezione personalizzando quella esistente. Si vedrà che il prossimo passo del ciclo di ragionamento considererà che ci sia già

un evento selezionato e che quest'ultimo verrà *rimosso* dall'insieme degli eventi. Diversamente, se l'insieme degli eventi risulterà vuoto, il ciclo passerà direttamente al passo 9, ovvero la sezione di una intenzione da eseguire in futuro.

6. Recupero di tutti i piani rilevanti

   - Ora che si possiede un evento selezionato, si avrà la necessità di trovare un piano che permetterà all'agente di agire per gestire quell'evento. La prima cosa da fare è trovare nella **Libreria dei piani**, tutti i piani che sono *rilevanti* per il dato evento. Questo è svolto recuperando tutti i piani, dalla libreria dell'agente, che hanno innescato l'evento selezionato o che possono essere legati per vari motivi con esso. Se ci sono eventuali annotazioni nel piano, affinché il piano sia rilevante, queste devono coincidere con quelle eventualmente abbinate all'evento.

7. Determinazione dei piani applicabili

   - Nei passi precedenti si è visto che ogni piano possiede un *contesto*, il cui ruolo è di determinare in quali circostanze può essere impiegato. Per questo motivo, anche se ho selezionato diversi piani *rilevanti*, potrebbe succedere che nessuno di essi sia *applicabile*. L'agente, perciò, dovrà seguire una linea d'azione che lo condurrà alla possibilità di gestire l'evento selezionato. Dato l'insieme dei piani rilevanti, occorrerà filtrare quelli che risultano correntemente applicabili di modo che verrà utilizzato un piano che, date le conoscenze dell'agente e le sue credenze correnti, sembra avere una possibilità di successo. Per fare questo, è necessario controllare se il contesto di ognuno dei piani rilevanti è creduto essere vero; in altre parole se il contesto è una *logica conseguenza* della base delle credenze dell'agente. Il prossimo passo del ragionamento ciclico si occuperà della scelta della modalità di azione.

8. Selezione di un piano applicabile

- Date le conoscenze dell'agente come descritte nella sua libreria dei piani e le sue informazioni correnti sul mondo come descritte nelle sue credenze di base, si riesce determinare tutto quello che si trova correntemente nei piani applicabili visti come le alternative adatte a maneggiare l'evento selezionato. Questo significa che, per quanto l'agente possa avvertire, *uno qualunque* di quei piani è attualmente idoneo. In altre parole, qualsiasi sia il piano scelto fra quelli applicabili, se l'esecuzione risulta corretta è sufficiente per la gestione dell'evento selezionato in questo ciclo di ragionamento. Per questo motivo l'agente necessita di scegliere *uno* dei piani applicabili ed impegnarsi ad eseguirlo. In particolare, impegnarsi ad eseguire un piano, significa effettivamente che l'agente ha *intenzione* di seguire la modalità d'azione del piano scelto, di modo che ci si aspetta che a breve finirà nell'insieme delle intenzioni dell'agente. Il piano applicabile scelto è chiamato *"intended means"* perché la linea d'azione definita da quel piano sarà il mezzo con il quale l'agente intenderà manipolare l'evento. Si ricordi che un evento è generalmente o un obiettivo da realizzare o un cambiamento nell'ambiente al quale l'agente deve reagire.

  La selezione del particolare piano dall'insieme dei piani applicabili che sarà incluso nell'insieme delle intenzioni è svolta da una distinta funzione chiamata *funzione di selezione del desiderio* oppure *funzione di selezione del piano applicabile* può essere personalizzabile.
  Gli obiettivi attualmente presenti nell'insieme degli eventi rappresentano diversi *desideri* su cui l'agente potrebbe impegnarsi, ciascun piano applicabile è, per ogni obiettivo, una differente *linea d'azione* che l'agente può percorrere per realizzarlo.

  La funzione di selezione di un piano applicabile predefinita sceglie un piano applicabile in base all'ordine in cui compaiono nelle librerie dei piani. Tale sequenza è fissata dalla successione con cui i piani sono scritti nel codice sorgente, o per essere più precisi dalla comunicazione dei piani all'agente.

  Si possono distinguere due metodi distinti per la selezione di un piano applicabile dall'insieme delle intenzioni, come si può notare nella figura 2.1. Questo perché ci sono due modi diversi per

aggiornare l'insieme delle intenzioni derivanti dalla natura, interno o esterno, dell'evento selezionato. Si ricordi che gli eventi interni sono cambiamenti negli obiettivi mentre gli eventi esterni sono percezioni di modifiche dell'ambiente. Se l'agente acquisisce un nuovo intended means (il piano applicabile selezionato) a causa di un evento esterno questo creerà nell'agente una *nuova intenzione*. Ogni distinta intenzione nell'insieme delle intenzioni rappresenta per l'agente un aspetto da tenere in considerazione e richiede il suo impegno per soddisfarla. Per esempio, se una persona sente suonare il campanello della porta e, contemporaneamente, squillare il telefono, può cercare di gestire entrambi gli eventi rispondendo al telefono mentre si reca verso la porta. In modo simile, un agente può gestire diversi eventi esterni nello stesso momento, tutti relativi a intenzioni che saranno in concorrenza per l'attenzione dell'agente.

Gli eventi interni vengono creati quando, come parte di un piano applicabile, l'agente acquisisce un nuovo goal da realizzare. Questo significa che, prima di continuare la linea d'azione che ha prodotto l'evento, si ha bisogno di trovare e portare a compimento una strategia per realizzare quel goal. Pertanto, per gli eventi interni, invece di creare nuove intenzioni, si inseriscono intended means in cima alle intenzioni esistenti. Questo produce, per una data intenzione, uno *stack* di piani. Tale metodo risulta opportuno perché l'interprete sa che il piano in cima alla pila è quello che *può* essere eseguito.

Si sottolinea che qualunque sia il piano scelto dalla libreria dei piani diventa un intended means, quest'ultimo è solo una *istanza* (ovvero una copia) di quel piano che attualmente è nell'insieme delle intenzioni. La libreria dei piani non viene variata, è l'istanza del piano ad essere manipolata dall'interprete.

9. Selezione di una intenzione da eseguire in futuro

- La prossima attività da svolgere, prima che l'agente possa iniziare ad agire, è selezionare una particolare intenzione fra quelle correntemente pronte per l'esecuzione. Per compiere tale scelta viene utilizzata una specifica funzione chiamata *funzione di selezione di una intenzione*. Di solito, non tutti gli obiettivi hanno lo

stesso grado di urgenza, perciò la scelta della prossima intenzione da eseguire è molto importante per definire l'intervento dell'agente sull'ambiente. **Jason** fornisce una funzione di selezione predefinita che può essere paragonata alla forma dello scheduling round-robin.

La strategia è di vedere la lista delle intenzioni come una coda circolare, ricordandosi che ogni intenzione è formata da uno stack di piani applicabili. Data la lista delle intenzioni, si seleziona quella in cima alla lista. Per l'intenzione selezionata viene eseguita solo *una* azione, più precisamente una formula del corpo di un piano applicabile. Una volta terminata l'azione, l'intenzione verrà rimossa dalla testa ed inserita in coda alla lista delle intenzioni. Questo produce, a meno che il programmatore non modifichi tale funzione di selezione, una garanzia di uguaglianza nell'operato dell'agente poiché si impegna equamente nella realizzazione di tutte le sue intenzioni.

10. Esecuzione di una fase di una intenzione

- Un agente, in ogni ciclo di ragionamento, svolge tre compiti principali:
  - aggiornare le informazioni che possiede sul mondo e sugli altri agenti;
  - gestire uno dei possibili eventi generati;
  - agire sull'ambiente o, più in generale, perseguire una delle sue intenzioni.

  Il momento di cui interessa discutere è quando l'agente sta cercando di compiere un'azione sull'ambiente ovvero il terzo compito. L'approccio che l'interprete possiede, per portare a termine una intenzione, è estrarre la formula nel corpo del piano che si trova in cima allo stack dei piani. Esistono sei differenti tipi di formule che possono presentarsi in tale contesto:
  - *azioni sull'ambiente*,
  - *achievements goals*,
  - *test goals*,
  - *note mentali*,

- *azioni interne*,
- *espressioni*.

Azioni sull'ambiente : Un'azione nel corpo del piano richiede all'agente di compiere qualcosa sull'ambiente. Il processo di ragionamento si occupa della scelta dell'azione da realizzare e poi l'architettura dell'agente, tramite gli *attuatori*, tenta di eseguirla. L'agente aspetta di ricevere dall'architettura un *feedback* booleano che segnala se l'azione è stata eseguita (true) o no (false). La strategia di **Jason** è di *sospendere l'intenzione* fino a che l'esecuzione dell'azione verrà conclusa, in modo che l'agente possa servirsi di questo tempo per svolgere altre operazioni. La formula, in attesa del feedback, viene rimossa dal corpo del piano ed inserita in una struttura speciale dell'interprete accessibile al metodo che si interfaccia con gli attuatori. La causa di tale procedura nasce per la necessità di svolgere il corpo del piano in *sequenza*.
Il notevole vantaggio, che comporta la sospensione di una azione, per l'agente è di evitare lo stato di inattività o idle provocato dall'attesa della ricezione del feedback.

Achievements goals : I nuovi obiettivi da realizzare sono stati definiti *eventi interni*. Facendo riferimento alla gestione delle intenzioni descritta all'ottavo passo si ricorda che nel momento in cui l'interprete raggiunge un obiettivo, genera un evento e la relativa intenzione si pone nell'insieme degli eventi piuttosto che tornare in quello delle intenzioni. Si verifica questo per la necessità di *sospendere le intenzioni*, esse non possono essere eseguite se prima non è stato stabilito un piano per realizzare gli obiettivi. Una intenzione inizialmente, quindi, viene sospesa perché un nuovo evento interno viene posto nell'insieme degli eventi, non in quello delle intenzioni sospese (che è esclusivamente per le intenzioni in attesa di un messaggio o una azione di feedback). Si noti che, diversamente dalle formule, gli obiettivi non sono immediatamente rimossi dal corpo del piano. Essi non sono rimossi quando l'evento interno viene creato, ma solo quando un piano viene attuato completamente realizzando con successo l'obiettivo. In questo modo, l'obiettivo viene rimosso

solo quando l'intenzione è pronta per la prossima formula da eseguire. Si anticipa che non è detto che tutti i piani svolti vengano realizzati con successo, ma esiste la possibilità del loro fallimento.

Test goals : Un test goal è usato per verificare se una certa proprietà è attualmente creduta dall'agente oppure una informazione è già nella base delle credenze dell'agente. Nel caso in cui, in uno dei due scenari sopra descritti, il test goal viene eseguito con successo, si potrà rimuovere l'obiettivo dal corpo del piano e l'intenzione aggiornata verrà messa in coda nell'insieme delle intenzioni. Un'altra attività dei test goals è di recuperare informazioni dalla base delle credenze. Nello stesso ciclo di ragionamento l'interprete cerca di risolvere l'obiettivo test esaminando tutte le credenze della base delle credenze. Se una credenza fornisce le informazioni necessarie, allora il test goal è subito soddisfatto. Di conseguenza l'intenzione viene inserita in coda alla lista delle intenzioni e, se selezionata, potrà procedere l'esecuzione nel prossimo ciclo di ragionamento.
Nel caso in cui, l'informazione non sia presente nella base delle credenze, l'interprete tenterà di realizzare il test goal guardando se esiste un piano *rilevante* nella libreria dei piani, invece di farlo fallire immediatamente.

Note mentali : Nel caso in cui la formula è una credenza che deve essere inserita o rimossa nella/dalla base delle credenze, tutto quello che l'interprete compie è passare le richieste appropriate al metodo `brf`. Il metodo `brf` esegue tutti i cambiamenti necessari alla base delle credenze e di conseguenza genera i rispettivi eventi, inserimento o rimozione di una credenza (belief additions or deletions). In **Jason**, il metodo `brf` svolge tutte le richieste indipendentemente dal mantenimento della consistenza.
Tipicamente, dopo l'invio della richiesta al metodo `brf`, la formula viene rimossa dal corpo del piano e l'intenzione inserita in coda alla lista delle intenzioni. Dato che questo metodo compie qualsiasi variazione nella base delle credenze, significa che verrà provocata la generazione di un evento

di tipo belief addition or deletion che scatenerà a sua volta l'esecuzione di un piano. Se questo piano viene eseguito come parte dell'intenzione o come un diverso centro d'attenzione può essere configurato dal programmatore. Un altro aspetto da tenere in considerazione è che, *a meno che il programmatore inserisca una specifica sorgente*, l'interprete in modo automatico associa come sorgente se stesso, la notazione è `source(self)`, a qualsiasi credenza che viene aggiunta/rimossa dalla/alla base delle credenze come una conseguenza di un tipo di formula di nota mentale nel corpo del piano.

Azioni interne : In questo caso, il codice Java (o linguaggio antecedente) fornito dal programmatore viene eseguito completamente, la formula è rimossa dal corpo del piano e l'intenzione aggiornata posta in fondo all'insieme delle intenzioni pronte per essere eseguite nel prossimo ciclo di ragionamento. Il codice utilizzato per implementare l'azione interna deve essere incluso in un metodo Java con valore di ritorno *booleano*, il quale esprime se l'azione è stata eseguita o no. Come per i goal, le azioni interne successivamente possono istanziare variabili dichiarate in precedenza nel corpo del piano. Questo può risultare utile se i risultati delle azioni interne devono essere usati nel ciclo di ragionamento. Una particolare attenzione dei programmatori è nel codice realizzato per l'esecuzione delle azioni interne dentro un ciclo di ragionamento, dato dal fatto che la computazione pesante potrebbe portare l'agente a non reagire in modo sufficientemente rapido ad importanti cambiamenti nell'ambiente, il che significherebbe avere un sistema multi-agente inefficiente/inadeguato.

Espressioni : L'utilizzo di espressioni è a volte pratico ma bisogna fare particolare attenzione a quando vengono usate nel corpo del piano piuttosto che nel contesto. Nel caso in cui una espressione risulta falsa all'interno del corpo del piano, quest'ultimo fallisce interamente e la gestione di tale situazione potrebbe essere molto costosa.

Prima dell'inizio di un altro ciclo, ci sono un paio di operazioni ausiliarie di routine che devono essere eseguite.

2.5.2 Il passo finale prima di ricominciare il ciclo di ragionamento

Alcune intenzioni potrebbero trovarsi nell'insieme delle intenzioni sospese perché in attesa o di un feedback dell'azione eseguita o di un messaggio di risposta da altri agenti. Prima che il ciclo di ragionamento ricominci, l'interprete controlla se attualmente sono disponibili feedback e/o risposte non processati. Nel caso in cui vi siano, le intenzioni correlate saranno aggiornate ed inserite in coda all'insieme delle intenzioni, in modo che abbiano una possibilità di essere scelte nella prossima esecuzione del successivo ciclo di ragionamento (passo 9).

Un'altra operazione di routine che può essere necessaria è cancellare:

- le intenzioni che possono essere state concluse; oppure
- i piani applicabili (intended means) che possono essere stati portati a termine.

La formula che si sta eseguendo nel corrente ciclo di ragionamento, è l'ultima contenuta nel corpo del piano originale nella libreria dei piani. Nel momento in cui il piano sarà realizzato con successo, verrà rimosso dalla cima dello stack che costituisce questa intenzione. Nel primo caso, questo piano si rivela essere l'ultimo della intenzione ovvero ora lo stack della intenzione è vuoto, l'intera intenzione è stata conclusa e può essere eliminata dall'insieme delle intenzioni. Nel secondo caso, nello stack c'è un altro piano sotto quello terminato. Questo è dovuto al fatto che il piano concluso era relativo all'achievement di un sotto-obiettivo in cui erano specificate delle variabili "unbound" cioè non legate ad alcun valore. L'esecuzione con successo del piano associato al sotto-obiettivo ha come conseguenza il "binding" delle variabili non legate ad un valore che dipenderà dal piano. L'interprete quindi, completato il sotto-piano, prima di eliminarlo dallo stack, verifica se erano state passate delle variabili non istanziate e in tal caso provvede a rendere effettivi i nuovi binding anche per il piano corrente. Il risultato è che il piano corrente vedrà i parametri passati come valori concreti.

L'agente è ora pronto ad iniziare un altro ciclo di ragionamento (passo 1).

2.5.3 Il fallimento di un piano

Gli approcci dei sistemi multi-agente per lo sviluppo software sono rivolti alle aree applicative dove l'ambiente è dinamico e spesso non prevedibile. In tali ambienti, per l'agente sarà frequente fallire un piano portando alla non realizzazione dello scopo che si era supposto di raggiungere. Per questo motivo, nello stesso modo con cui si forniscono piani agli agenti così che essi abbiano delle modalità d'azione alternative, tra cui scegliere nel momento in cui necessitano di raggiungere un obiettivo, nasce il bisogno di dare agli agenti piani che stabiliscono cosa fare quando si verifica il *fallimento* di un piano scelto. Questi piani vengono detti *contingency plans*.
Di seguito verranno descritte tre principali cause che possono portare al non raggiungimento di un piano.

- *Mancanza di piani rilevanti o applicabili per la realizzazione di un achievement goal.*
- *Fallimento di un test goal.*
- *Fallimento di una azione.*

Mancanza di piani rilevanti o applicabili per la realizzazione di un achievement goal.
Un agente si trova, nella circostanza corrente, a non sapere *come* realizzare i suoi desideri. Tale situazione si verifica quando ad un agente, durante l'esecuzione di un piano, viene richiesto di compiere un sotto-obiettivo che non sa come realizzare. Questo può essere provocato da:

- mancanza di piani rilevanti;
- mancanza di piani applicabili.

Nel primo caso l'agente non possiede le conoscenze richieste, poiché non sono presenti piani rilevanti per i sotto-obiettivi. Il motivo è che il programmatore non ha previsto il piano richiesto. Nel secondo caso tutti i modi conosciuti dall'agente per la realizzazione dell'obiettivo non possono essere attualmente impiegati. Ci sono piani rilevanti conosciuti, ma i loro contesti non soddisfano le credenze correnti dell'agente e così risultano momentaneamente inapplicabili.

Fallimento di un test goal. L'agente si aspetta di conoscere una certa proprietà del mondo, ma in realtà tale conoscenza non si trova nella base delle credenze. Tale mancanza è in grado di influenzare il successo del piano. Se l'agente non riesce a rientrare in possesso dalla base delle credenze delle informazioni necessarie a soddisfare il test goal, l'interprete tenta di generare un evento per eseguire un altro piano in grado di recuperarle. Nel caso in cui anche questo approccio ha esito negativo (ad esempio per mancanza di piani rilevanti o applicabili) allora l'obiettivo test verrà ritenuto fallito e di conseguenza il piano dove è contenuto fallirà.

Fallimento di una azione. In **Jason** vi sono due tipi di azioni:

- interne;
- esterne, dette anche di base o di ambiente.

Le azioni interne prevedono l'utilizzo di metodi Java con valore di ritorno booleano, mentre quelle esterne fanno uso di feedback booleani. Entrambi questi strumenti indicano se l'azione è stata eseguita (true) o no (false).

Se una azione, sia interna che esterna, fallisce, di conseguenza fallisce anche il piano dove è contenuta. Indipendentemente dalla ragione per cui un piano fallisce, l'interprete **Jason** genera un evento "goal deletion event", con notazione `-!g`, per la cancellazione dell'obiettivo e, come risultato, tutti i piani che ne prevedono la realizzazione, `+!g`, falliscono.

Il compito di un piano per un goal deletion è di eseguire una "pulizia approfondita", infatti viene detto piano di "clean-up". Pulire significa tornare indietro per capire le conseguenze provocate dalla scelta e dall'esecuzione del piano fallito ed eventualmente eliminarle. Un possibile approccio è quello di eseguire il "backtracking", ovvero tentare un altro piano, detto alternativo o di recupero, per realizzare l'obiettivo per il quale il piano è fallito. I programmi AgentSpeak, durante l'esecuzione di obiettivi e sotto-obiettivi, generano una sequenza di azioni che l'agente compie sull'ambiente esterno per modificarlo. Gli effetti di tali modifiche non possono essere semplicemente cancellati

eseguendo un backtracking, poiché non basta solo annullare le azioni compiute, ma potrebbe essere richiesto di svolgerne sull'ambiente di ulteriori per riuscirci. I piani di recupero, infatti, possono essere strutturati in modo che, prima l'agente operi in modo da annullare le conseguenze di alcune azioni svolte prima del fallimento del piano, e solo dopo tenterà *nuovamente* di realizzare quell'obiettivo. I piani di recupero vengono denotati dal goal deletions (`-!g`) come triggering events.

Nel caso in cui nessun piano di recupero sia previsto, allora l'obiettivo fallirà in maniera definitiva, ovvero senza nessun altro tentativo. Il backtracking , si è visto, che non è un meccanismo automatico, ma se si specifica un piano come:

```
-!g :  true <- !g.
```

si è certi che il piano di recupero viene *sempre* eseguito, essendo il contesto pari a true, con tentativo di backtracking (`!g`).

Quando si verifica un fallimento di un piano, l'intera intenzione è lasciata cadere. Solo nei casi in cui il triggering event del piano in esecuzione è un achievement goal addition o un test goal *addition*, allora si può tentare di recuperare il fallimento attraverso il "goal deletion construct". Nei casi all'infuori dei goal addition (achievement goal o test goal), il fallimento di un piano comporta che l'intera intenzione non può essere compiuta. Nel caso in cui fallisca un piano per un goal addition (achievement o test goal) l'intenzione i, dove quel piano appare, viene sospesa ed inserita nell'insieme degli eventi con il rispettivo evento goal deletion event ≪`-!g, i`≫. Questo può portare al fatto che il test goal addition venga tentato di nuovo come parte del piano per gestire l'evento `-!g`. Quando un piano per `-!g` finisce, non solo se stesso ma anche il piano `+!g` fallisce e viene rimosso dalla intenzione.

Al programmatore spetta la scelta di:

- lasciare o rimuovere il piano dallo stack delle intenzioni. In particolare esso potrebbe preferire mantenerli per esigenze di veri-

fica di quali azioni interne **Jason** ha compiuto in risposta allo specifico piano fallito;

- tentare di soddisfare nuovamente il goal, oppure lasciare cadere l'intera intenzione.

2.6 Il concetto di ambiente

Uno degli aspetti chiave degli agenti autonomi è che sono *situati* in un "environment" o ambiente. Nei sistemi multi-agente, l'*ambiente* è condiviso da una molteplicità di agenti, così che le azioni compiute da un agente sono probabilmente alterate da quelle degli altri agenti.

In molte applicazioni multi-agente, l'ambiente è il mondo reale. Internet è un tipico ambiente dove gli agenti sono situati. In altri casi, si necessita prima di creare un modello computazionale di un mondo reale o artificiale, poi di essere capaci di simulare gli aspetti dinamici di tale ambiente. Dopodiché, si implementeranno gli agenti che opereranno in questo ambiente simulato, agendo su di esso e percependone le proprietà. Questo procedimento è particolarmente importante, per esempio, per la *social simulation*, un'area di ricerca che fa ampio uso delle tecniche dei sistemi multi-agente. Risulta, inoltre, molto utile simulare ambienti per applicazioni che sono finalizzate allo sviluppo di ambienti del mondo reale. I sistemi multi-agente sono normalmente utilizzati per sviluppare sistemi distribuiti complessi, poiché in grado di verificarne e convalidarne il funzionamento.

Molti sviluppatori utilizzano un modello Java dell'ambiente del mondo reale, perché da questo riescono a valutare quanto il sistema è performante. La ragione della scelta di Java è data dal fatto che l'astrazione della programmazione ad agenti è ottima per implementare gli *agenti*. Java, infatti, possiede tutte le astrazioni necessarie per la creazione di modelli ambientali, come gli oggetti e le interfacce grafiche.

Capitolo 3

Le basi della robotica

3.1 L'origine del termine robot

L'introduzione del terimine "robot" si deve allo scrittore ceco Karel Capek, il quale lo usò per la prima volta nel 1921 nel suo dramma teatrale *Rossum's Universal Robots*. In realtà, non fu il vero inventore della parola, la quale, infatti, gli venne suggerita dal fratello Josef. In particolare, questo termine risulta dalla combinazione delle parole ceche *rabota*, che significa lavoro forzato, e *robotnik*, che significa servo.

L'idea di un robot, ovvero di un tipo di macchina capace di aiutare le persone, è molto più antica rispetto ai fratelli Capek. Non è possibile identificare quando si sia originata, perché è probabile che alcuni ingegneri del passato se ne servissero già in qualche forma. La forma si è modificata nel corso del tempo, come è avanzata la scienza e la tecnologia, ma il concetto di robot è rimasto pressoché invariato nel tempo. Oggi molti dei sogni di robot, di fatto irragiungibili in passato, sono diventati realtà o almeno sono entrati nel regno delle possibilità.

Il processo scientifico e tecnologico ha portato ad un concetto di robot sempre più complesso. In origine, le nozioni di robot erano davvero di automi meccanici intelligenti, visti come le macchine di calcolo sviluppate all'epoca (in particolare quando si è riusciti a ristringerle in dimensioni sufficienti da essere immaginati dentro al corpo di un robot). In seguito, le nozioni di robot hanno iniziato ad includere il pensiero, il ragionamento, la risoluzione di un problema (problem-resolving), le emozioni e la coscienza.

In altre parole, si è iniziato a guardare i robot sempre più come creature biologiche, andando dagli insetti agli esseri umani.

3.2 Il concetto di robot

Un *robot* è un sistema autonomo che esiste nel mondo fisico, può percepire il suo ambiente e può agire su di esso per realizzare alcuni obiettivi.

Questa potrebbe sembrare una definizione molto ampia, ma attualmente ogni parte di essa risulta importante e necessaria.

Un robot è un sistema AUTONOMO...

Un robot *autonomo* agisce sulla base di decisioni proprie, non è controllato da un uomo. Le macchine esternamente controllate dall'uomo sono dette *telecomandate*. In particolare, telecomandare significa gestire un sistema a distanza. I robot agiscono autonomamente, in quanto essi possono essere capaci di prendere input e avvisi, o consigli, dagli uomini, ma non sono completamente controllati da essi.

Un robot è un sistema autonomo che esiste nel MONDO FISICO...

Esistere nel mondo fisico, lo stesso in cui sono presenti anche le persone, gli animali, gli alberi, il tempo ed altri oggetti, è una caratteristica fondamentale dei robot. Doversi preoccupare di questo mondo fisico, con le sue rigide leggi fisiche, è ciò che rende la robotica una vera sfida. I robot esistenti nel computer sono simulazioni. Essi non si devono occupare delle reali proprietà del mondo fisico, perché gli ambienti simulati non sono mai complessi quanto il mondo reale. Perciò, sebbene esistono molte simulazioni di robot nel cyberspazio, un robot viene definito tale solo se esiste nel mondo fisico.

Un robot è un sistema autonomo che esiste nel mondo fisico, può PERCEPIRE il suo ambiente...

Percepire l'ambiente significa che il robot ha dei *sensori* di percezione (ad

esempio ascolta, tocca, annusa, ecc..) al fine di prendere informazioni o conoscenza dal mondo. Ad un robot simulato, viceversa, possono essere date semplicemente le informazioni e le conoscenze senza che esso deve compiere nulla, come per magia. Un vero robot può percepire il suo mondo solo attraverso i sensori, come fanno le persone e gli animali. Infatti, se un sistema non percepisce le informazioni ma gli vengono fornite magicamente, non si considera come un vero robot.

Un robot è un sistema autonomo che esiste nel mondo fisico, può percepire il suo ambiente e può AGIRE SU DI ESSO...

Compiere azioni in risposta agli input rilevati dai sensori e realizzare cosa è desiderato, è una parte necessaria dell'essere un robot. Una macchina che non svolge azioni, cioè non si muove o non influisce sul mondo realizzando/cambiando qualcosa, non è un robot.

Un robot è un sistema autonomo che esiste nel mondo fisico, può percepire il suo ambiente e può agire su di esso per REALIZZARE ALCUNI OBIETTIVI.

Finalmente si introduce l'intelligenza, o almeno l'utilità, di un robot. Un sistema o una macchina che esiste nel mondo fisico e lo percepisce, ma agisce a caso o inutilmente, non vale molto come robot. In quanto, non usa le informazioni percepite e le sue abilità per agire in modo da compiere qualcosa di utile per se stesso o per gli altri. Infatti, un vero robot possiede obiettivi e agisce per realizzarli.

Si è stabilito cosa è un robot, ora si definisce cosa è la robotica. La *robotica* è lo studio dei robot, ovvero la ricerca su come rendere le loro percezioni autonome e significative per fare in modo che siano operanti nel mondo fisico.

3.3 I componenti

La definizione di robot fornisce alcuni suggerimenti sulle parti che lo compongono.

UN ROBOT È UN SISTEMA AUTONOMO CHE ESISTE NEL MONDO FISI-

CO, PUÒ PERCEPIRE IL SUO AMBIENTE E PUÒ AGIRE SU DI ESSO PER REALIZZARE ALCUNI OBIETTIVI

Un robot è formato dai seguenti componenti principali:

- un corpo fisico, in modo che esso può esistere e lavorare nel mondo fisico;
- i sensori, in modo che esso sente/percepisce il suo ambiente;
- gli effettori e gli attuatori, in modo che esso può compiere azioni;
- un controllore (controller), in modo che esso può essere autonomo.

3.3.1 Embodiment

Possedere un corpo è il primo requisito per essere un robot. Esso permette al robot di compiere qualcosa: muoversi e agitarsi, spostarsi, conoscere le persone e realizzare il proprio lavoro. Un agente software, non importa quanto realisticamente sia simulato, non è un robot perché non condivide l'universo fisico con il resto delle creature fisiche. Esso non risulta *physically embodied* nel mondo reale. *"Embodiment"* significa possedere un corpo fisico. Tale requisito è necessario, ma ha un prezzo:

- Un robot embodied deve obbedire alle stesse leggi fisiche alle quali rispondono tutte le creature fisiche. Esso non può essere in più posti contemporaneamente; non può cambiare forma e dimensione arbitrariamente; deve usare gli effettori presenti nel suo corpo per spostarsi; ha bisogno di alcune fonti di energia per percepire, pensare e muoversi; quando è in moto, necessita di un po' di tempo per accelerare e decelerare; il suo movimento modifica l'ambiente in vari modi; non può essere invisibile; e così via.

- Avere un corpo significa dover percepire gli altri corpi e oggetti intorno a sé. Tutti i robot fisici si devono preoccupare di non scontrasi con le altre cose nell'ambiente. Questo sembra semplice, ma non lo è. Tipicamente, infatti, la prima cosa che si programma in un robot è evitare le collisioni.

- Ognuno ha i suoi limiti. La forma del corpo del robot, per esempio, condiziona molto il movimento, le sensazioni (perché i sensori sono attaccati al corpo), i lavori che può svolgere e come esso interagisce con gli altri esseri (robot, automobili, tagliaerba, gatti, persone, ecc...) nel suo ambiente.

- Oltre a condizionare le cose che coinvolgono lo spazio e il movimento, il corpo influenza anche quelle che hanno a che fare con il tempo. Il corpo determina quanto veloce il robot può muoversi e reagire nel suo ambiente. Spesso si dice che i robot più veloci sembrano più intelligenti, questo è un esempio di come l'embodiment del robot influisce sulla sua immagine.

3.3.2 Percezione

I sensori sono dispositivi fisici che abilitano un robot a percepire il suo ambiente fisico in modo da ricavare informazioni su se stesso e sull'ambiente. In robotica, i termini *sentire* e *percepire* sono trattati come sinonimi, entrambi vengono riferiti al processo di ricezione delle informazioni sul mondo attraverso i sensori.

Un buon progettista e programmatore di robot inserisce il giusto tipo di sensori sul robot, così da permettere ad esso di percepire le informazioni di cui ha bisogno per svolgere il suo lavoro e per realizzare i suoi obiettivi. In modo simile, gli animali hanno evoluto sensori che sono estremamente adeguati per la loro *nicchia*, vale a dire per l'ambiente in cui vivono e per la loro posizione nell'ecosistema. Una nicchia è posseduta anche dai robot, che consiste nel loro ambiente ed incarico. Inoltre i robot, proprio come gli animali, più si adattano alla loro nicchia, più a lungo sopravvivono.

La percezione consente al robot di conoscere il suo stato. Lo *stato* è una nozione generale dalla fisica, presa in prestito dalla robotica, che riguarda la descrizione di un sistema. Lo stato di un robot è la descrizione di se stesso in ogni istante di tempo. Il robot è detto essere "in uno stato".

Per un robot, uno stato potrebbe essere visibile (formalmente detto *osservabile*), parzialmente nascosto (*parzialmente osservabile*) oppure *nascosto* (inosservabile). Questo significa che un robot può sapere pochissimo o molto su se stesso e sul suo mondo.

Uno stato può essere *discreto* (sopra, sotto, blu, rosso) o *continuo* (100 km/h). Tale attributo è efficace per riferirsi al tipo ed alla quantità di informazione usata per descrivere il sistema.

Lo *spazio degli stati* consiste nell'insieme di tutti i possibili stati in cui può trovarsi un sistema. Il termine spazio, in questo contesto, è collegato a tutti i possibili valori o variazioni di qualcosa.

Risulta spesso utile per un robot distinguere due tipi di stati relativi a se stesso: esterno ed interno. Lo *stato esterno* si riferisce a come il robot può percepire lo stato del mondo, mentre lo *stato interno* si riferisce a come il robot può percepire lo stato del robot, ossia il suo stato.

Gli stati interni possono essere usati per ricordare informazioni sul mondo. Le quali vengono indicate con i termini *rappresentazione* o *modello interno*. Le rappresentazioni ed i modelli hanno molto a che fare con la complessità del cervello di un robot.

In generale, in qualunque modo l'intelligenza di un robot si presenti, risulta fortemente influenzata da in che misura e quanto velocemente esso riesce a percepire il suo ambiente e se stesso, ovvero il suo stato esterno ed interno.

Tutti i sensori del robot, messi insieme, formano lo spazio di tutte le possibili letture sensoriali, il quale è chiamato *spazio dei sensori* del robot o *spazio percettivo*. Un progettista e programmatore di robot ha bisogno di mettere la sua mente nello spazio dei sensori del robot, al fine di immaginare come il robot percepisce il mondo e, di conseguenza, come reagire ad esso. Questo non è facile da fare, perché, anche se i robot esistono nel nostro mondo fisico, essi lo percepiscono in maniera molto diversa rispetto a noi.

3.3.3 Azione

Gli *effettori* abilitano un robot a svolgere le azioni, ovvero compiere gesti fisici. Essi sono la migliore alternativa a gambe, pinne, ali e varie altre parti del corpo che permettono agli animali di muoversi. Gli effettori utilizzano dei meccanismi fondamentali, come muscoli e motori, che sono chiamati *attuatori* ed i quali realizzano concretamente il lavoro per il robot. Come per i sensori, gli effettori ed attuatori robotizzati sono molto diversi da quelli biologici. Questi vengono usati per due attività principali:

1. *Locomozione:* muoversi, spostarsi

2. *Manipolazione:* manipolare oggetti.

Tali attività corrispondono a due dei maggiori sottocampi della robotica:

1. Robotica mobile, riguarda il movimento dei robot, soprattutto sul terreno, ma anche nell'aria e sott'acqua.

2. Robotica manipolatrice, riguarda soprattutto le braccia robotiche dei vari tipi.

I robot mobili usano meccanismi di locomozione come ruote, cingoli o gambe. I robot manipolatori usano braccia e pinze robotizzate, inoltre, possono muoversi in una o più dimensioni. Le dimensioni nelle quali i robot manipolatori possono muoversi sono dette *gradi di libertà*. La distinzione tra robot mobili e manipolatori sta lentamente scomparendo poiché robot più complessi, in quanto sempre più simili agli esseri umani, cioè capaci sia di muoversi che di manipolare oggetti, sono in fase di sviluppo.

3.3.4 Cervello e muscoli

Per quanto riguarda la potenza richiesta dal cervello, i robot e gli animali sono molto diversi. Negli animali, il cervello biologico assorbe una grande quantità di energia rispetto al resto del corpo, soprattutto negli esseri umani. Nei robot, è il contrario, gli attuatori richiedono più potenza del processore che gestisce il controllore, cioé il cervello.

Nel momento in cui inizia la computazione, animali e robot sono simili poiché entrambi necessitano di un cervello per funzionare correttamente. Tuttavia, sia gli umani si pensi al coma, sia i robot si pensi al processore in reset, potrebbero sopravvivere per un certo periodo di tempo senza usare il cervello. In tutti i casi, questo non è una cosa buona, quindi bisogna assicurarsi di proteggere il cervello dei robot ed il proprio.

La potenza è uno dei maggiori problemi nella parte pratica della robotica. I problemi di alimentazione sono:

- fornire l'energia sufficiente al robot in modo che non si trovi con le batterie pesanti scariche

- mantenere efficacemente isolata l'elettronica dai sensori e dagli effettori

- prevenire la perdita di prestazioni, come il calo del livello di potenza dato dall'usura delle batterie logorate o da una richiesta improvvisa di alta potenza
- rifornire la potenza in modo autonomo, ossia dal robot stesso anziché da persone
- e molti altri.

Nella robotica, bisogna tenere a mente che, il cervello e i muscoli sono direttamente collegati e connessi, perciò devono essere progettati e discussi insieme.

3.3.5 Autonomia

I controllori forniscono l'hardware e/o il software che rende il robot autonomo, ovvero capace di utilizzare i sensori di input e qualsiasi altra informazione per decidere cosa fare (le azioni da intraprendere) e, in seguito, controllare gli effettori durante l'esecuzione dell'azione. I controllori ricoprono il ruolo del cervello e del sistema nervoso. Di solito, in un robot è presente più di un controllore, in modo che varie parti di esso possono essere processate contemporaneamente.

L' *autonomia* è la capacità di prendere decisioni proprie e di agire in base ad esse. Per i robot, autonomia significa che le decisioni vengono prese ed eseguite dal robot stesso, non da parte di operatori umani. L'autonomia può essere completa o parziale. I robot completamente autonomi prendono le loro decisioni ed agiscono in base ad esse. Al contrario, i robot telecomandati sono parzialmente o totalmente controllati dagli umani.

3.4 Il controllo retroazionato

3.4.1 Il controllo retroazionato o ad anello chiuso

Il *controllo retroazionato* è uno strumento per ottenere un sistema (un robot) che raggiunge e mantiene uno stato desiderato, di solito chiamato *set point*, attraverso il continuo confronto tra il suo stato attuale e quello desiderato.

La *retroazione* (feedback) si riferisce alle informazioni che vengono inviate indietro, letteralmente "fed back", al controllore del sistema.

Lo *stato desiderato* del sistema, anche chiamato *stato obiettivo*, è dove il sistema vorrebbe essere. Non sorprende che la nozione di stato obiettivo è essenziale per i sistemi finalizzati ad un meta (goal-driven), così viene utilizzata sia nella teoria del controllo, sia nell'intelligenza artificiale (AI).

Lo stato obiettivo di un sistema può essere correlato allo stato interno, esterno o ad una combinazione di entrambi. Esso, inoltre, può essere arbitrariamente complesso e consistere in una serie di requisiti e vincoli. In AI, gli obiettivi (goals) si suddividono in due tipi: achievement e maintenance.

Gli *achievement goals* sono gli stati che il sistema tenta di raggiungere. Una volta che il sistema si trova in tale stato, è realizzato e non ha bisogno di eseguire nessun altro lavoro.

I *maintenance goals* in corso richiedono un lavoro attivo da parte del sistema. La teoria del controllo si è tradizionalmente (ma non esclusivamente) occupata di maintenance goals. In generale, come goals per un robot possono essere utilizzati sia obiettivi realizzabili, che irrealizzabili. Gli obiettivi realizzabili, sono quelli che il robot è in grado di raggiungere e mantenere, mentre quelli irrealizzabili risultano inattuabili date le capacità da esso possedute. In questo ultimo caso, il robot può solo continuare a provare senza mai riuscirci.

Nel caso in cui lo stato attuale del sistema risulta diverso da quello desiderato, il robot continua a sforzarsi per raggiungerlo agendo nel modo dettato dal controllore.

3.4.2 Le molte facce dell'errore

La differenza fra stato corrente e desiderato di un sistema è chiamato *errore*. Qualsiasi sistema di controllo ha come obiettivo minimizzare quell'errore. Il controllo retroazionato esplicitamente elabora e riporta al sistema l'errore, al fine di aiutarlo a realizzare l'obiettivo. Nel caso in cui l'errore è zero (o abbastanza piccolo), lo stato obiettivo risulta raggiunto.

L'informazione più semplice che può essere contenuta nella retroazione, è se il sistema si trova o no nello stato obiettivo. Questa risulta essere insoddisfacente, in quanto se non si è nello stato obiettivo non viene fornito nessun aiuto su come arrivarci.

Si può pensare, quindi, ad una *direzione dell'errore* ovvero una indicazione sulla tendenza dello stesso, che permette di minimizzarlo ed avvicinare l'obiettivo.

Una informazione ancora più precisa è la *grandezza dell'errore*, cioè la distanza dallo stato obiettivo.

Nel momento in cui al sistema viene data sia la direzione, sia la grandezza dell'errore, gli viene detto esattamente che cosa fare per raggiungere l'obiettivo.

Di conseguenza, il controllo è reso più facile se al robot vengono fornite, in modo accurato e frequente, molte informazioni sugli errori.

3.4.3 I tipi di controllo retroazionato

I tre tipi di controllo retroazionato più usati sono:

- controllo proporzionale (P),
- controllo proporzionale derivativo (PD),
- controllo proporzionale integrale derivativo (PID).

Controllo proporzionale. L'idea di base del *controllo proporzionale* è di avere il sistema che risponde in modo proporzionale all'errore, utilizzando sia la direzione, sia la grandezza dell'errore.

Nella teoria del controllo, i parametri che determinano la grandezza della risposta del sistema sono chiamati *guadagni* (gains). Determinare i guadagni corretti è, in genere, molto difficile. L'approccio è per tentativi, ovvero le fasi di test e calibrazione del sistema vengono eseguite più volte finché non si raggiungono valori soddisfacenti. In alcuni rari casi, quando il sistema viene capito veramente bene, i guadagni possono essere calcolati matematicamente.

Se il valore del guadagno è proporzionale a quello dell'errore, viene chiamato *guadagno proporzionale*.

Lo *smorzamento* (damping) si riferisce al processo sistematico di riduzione delle oscillazioni. Un sistema è adeguatamente *smorzato* se non oscilla fuori controllo, ovvero le sue oscillazioni sono completamente evitate o diminuiscono gradualmente verso lo stato desiderato entro un periodo di tempo ragionevole. I guadagni devono essere adeguati al fine di rendere un sistema correttamente smorzato. Tale metodo è un processo di ottimizzazione, che risulta specifico per ogni particolare sistema di controllo (robot o altro).

Quando si regolano i guadagni, si devono tenere in considerazione le proprietà fisiche e computazionali del sistema. Le proprietà fisiche del robot influenzano i valori esatti dei guadagni, perché limitano ciò che il sistema realmente compie in risposta ad un comando.

L' *indeterminazione dell'attuatore* rende impossibile, per un robot, conoscere il risultato esatto di una azione, anche semplice, prima del tempo. Tuttavia, se si conosce abbastanza bene il sistema, si è in grado di stimare la probabilità di successo, così da eseguire una supposizione abbastanza buona sull'esito dell'azione.

Controllo proporzionale derivativo. L'intervento per regolare il guadagno basato semplicemente sull'aumento dello stesso, non risolve i problemi di oscillazione del sistema di controllo. Tale azione può risultare soddisfacente solo per piccoli guadagni, dal momento che con l'aumento del guadagno, le oscillazioni del sistema crescono con esso. Il problema di fondo riguarda la distanza tra il set point e lo stato desiderato: *quando il sistema è vicino allo stato desiderato deve essere controllato in modo diverso rispetto a quando è lontano da esso.* Altrimenti, la quantità di moto, generata dalla risposta del controllore per l'errore, porta il sistema oltre lo stato desiderato causando oscillazioni. Una soluzione a questo problema risulta quella di correggere la quantità di moto quando il sistema si avvicina allo stato desiderato.

La quantità di moto è data dal prodotto della massa per la velocità:

$$\textit{Quantità di moto} = \textit{massa} * \textit{velocità}$$

Dalla formula si deduce che la quantità di moto e la velocità sono direttamente proporzionali. Di conseguenza, si può controllare la quantità di moto attraverso la velocità del sistema. Nel momento in cui il sistema si avvicina allo stato desiderato, si sottrae una quantità proporzionale alla velocità:

$$-(\textit{guadagno} * \textit{velocità})$$

Questa quantità viene definita *termine derivato*, in quanto la velocità è la derivata della posizione. Infatti, un controllore che possiede un termine derivato è chiamato controllore *D*.

Un controllore derivativo produce un output o proporzionale alla derivata del suo input i:

$$o = \mathrm{K}_d \frac{di}{dt}$$

K_d è una costante proporzionale.

Lo scopo del controllo derivativo è di progettare un controllore in grado di correggere la quantità di moto del sistema, appena si avvicina allo stato desiderato.

Controllo proporzionale integrale derivativo. Una ulteriore miglioria per il sistema può essere fatta introducendo il cosiddetto termine integrale o *I*. L'idea è che il sistema tiene traccia dei propri errori, in particolare di quelli ripetibili e fissa gli errori che sono detti *errori a regime stazionario*. Nel corso del tempo il sistema integra (riassume) questi errori elementari ed una volta raggiunta una certa soglia determinata in precedenza (ovvero quando l'errore cumulativo diventa abbastanza grande), il sistema esegue qualcosa per compensarlo/correggerlo.

Un controllore integrale produce un output o proporzionale all'integrale del suo input i:

$$o = \mathrm{K}_f \int i(t)dt$$

K_f è una costante proporzionale.

La maggior parte dei sistemi reali utilizzano le combinazioni dei tre tipi di base: *P, I, D*. I regolatori *P D* e *P I D* sono particolarmente diffusi, tanto che risultano comunemente utilizzati nelle applicazioni industriali.

Il regolatore *P D* è una combinazione, in realtà semplicemente una somma, dei termini del controllo proporzionale (P) e derivativo (D):

$$o = \mathrm{K}_p i + \mathrm{K}_d \frac{di}{dt}$$

Il regolatore *P D* è estremamente utile e applicato nella maggior parte degli impianti industriali per il processo di controllo.

Il regolatore *P I D* è una combinazione dei termini del controllo proporzionale (P), integrale (I) e derivativo (D):

$$o = \mathrm{K}_p i + \mathrm{K}_f \int i(t)dt + \mathrm{K}_d \frac{di}{dt}$$

3.4.4 Il controllo in avanti o ad anello aperto

Il controllo retroazionato è anche definito *controllo ad anello chiuso* per il fatto che chiude il ciclo fra input e output e fornisce il sistema con l'errore come retroazione.

L'alternativa al controllo retroazionato o ad anello chiuso è chiamata controllo in avanti oppure ad anello aperto. Il nome, *controllo ad anello aperto* o *controllo in avanti*, suggerisce che i feedback sensoriali non vengono usati, lo stato non è retroazionato nel sistema e, di conseguenza, il ciclo fra input e output risulta aperto. Nel controllo ad anello aperto, il sistema esegue il comando che gli viene assegnato, in base a quanto è stato previsto, senza interessarsi né dello stato attuale del sistema, né di come procede il suo aggiornamento. Al fine di decidere in anticipo come agire, il controllore determina i suoi set point o sotto-obiettivi prima del tempo. Tale approccio richiede di guardare avanti (looking forward) e prevedere lo stato del sistema, motivo per cui l'approccio è chiamato in avanti (feedfoward).

I sistemi di controllo in avanti o ad anello aperto possono operare in modo efficace se sono ben calibrati e il loro ambiente è prevedibile, cioè che non si modifica in modo da incidere sulle sue performance. Date le caratteristiche possedute, tali sistemi sono adatti per iterazioni e compiti indipendenti dallo stato, ragione per cui non risultano diffusi in robotica.

3.5 Le architetture di controllo

L'incarico del controllore è di occuparsi del cervello del robot allo scopo di renderlo autonomo e capace di realizzare gli obiettivi. Si è visto che il controllo in retroazione risulta un buon approccio esclusivamente per i robot che eseguono un unico comportamento. La maggior parte dei robot, però, possiedono più compiti da realizzare, i quali vanno dalla semplice sopravvivenza (non sbattere contro alle cose oppure esaurire la carica) al conseguimento di un compito complesso (qualunque esso sia). In un determinato momento, iI robot si trova di fronte ad una moltitudine di cose da fare, scegliere su quale impegnarsi non è affatto semplice. La soluzione adottata, anche se complessa, dal controllo del robot è realizzata mettendo insieme i controllori da esso posseduti per produrre il comportamento globale desiderato.

3.5.1 Chi necessita delle architetture di controllo?

Esistono numerosi modi in cui il controllore di un robot può essere programmato (e ci sono infiniti possibili programmi per il controllo di robot), la maggior parte dei quali funzionano piuttosto male, vanno dal completamente sbagliato all'inefficiente. Per trovare un buon (corretto, efficiente, perfino ottimale) modo attraverso il quale controllare un particolare robot durante la realizzazione di un determinato compito, si ha bisogno di conoscere alcuni principi guida per il controllo del robot oltre a modi sostanzialmente diversi in cui i robot possono essere programmati. Questi aspetti sono catturati nelle architetture di controllo del robot.

Una *architettura di controllo* di un robot fornisce i principi guida ed i vincoli per l'organizzazione del sistema di controllo del robot (il suo cervello). Essa aiuta il progettista a programmare il robot in modo tale da produrre in output il comportamento globale desiderato.

Il significato del termine *architettura* in questo contesto è equivalente a quello attribuito all'architettura del computer, dove con esso si intende l'insieme dei principi per la progettazione di computer. Allo stesso modo, nell'architettura dei robot, vi è un insieme di blocchi o strumenti a disposizione per rendere il lavoro di progettazione del controllo di un robot più semplice. Il concetto di architettura risulta adeguato, poiché esprime l'utilizzo di stili specifici, strumenti, vincoli e regole.

Nel momento in cui si vogliono ottenere dei robot capaci di realizzare compiti, anche semplici, non basta conoscere solo la robotica. Bisogna introdurre le architetture di controllo, grazie alle quali è possibile produrre perfino robot complessi (o un gruppo di robot) in grado di compiere qualcosa di utile e robusto in un ambiente complicato.

Il controllo del robot, ovvero il suo cervello, può essere implementato con un programma tradizionale in esecuzione su un microprocessore, può essere integrato nell'hardware o può essere una combinazione di entrambi. Il controllore del robot non ha il vincolo di essere un unico programma su un singolo processore. Nella maggior parte dei casi, infatti, non è la soluzione adottata. La prima ragione è che il controllo centralizzato non risulta robusto al fallimento. La conseguenza nel caso in cui in un robot, controllato in modo centralizzato, si verificasse il fallimento dell'unico processore, sarebbe il suo blocco ovvero il robot smetterebbe di funzionare.

Il controllo del robot può avvenire via hardware o via software. Il con-

trollo via hardware è adatto per impieghi veloci e specializzati, mentre via software ha il vantaggio di essere flessibile. Il cervello di un robot risulta complicato, poiché solitamente coinvolge programmi di varie tipologie in esecuzione in tempo reale sul robot. Solamente a causa dei pregiudizi basati sui sistemi biologici, si tende a realizzare il cervello fisicamente sul robot.

I cervelli, robotizzati o naturali, devono utilizzare i loro programmi per risolvere i problemi che ostacolano il modo di raggiungere i loro obiettivi e portare a termine i loro lavori. Il processo di risoluzione di un un problema utilizzando un insieme finito, o non finito, di procedure svolte passo dopo passo (step by step) è chiamato *algoritmo*. Il campo della scienza informatica dedica un gran numero di ricerche per lo sviluppo e l'analisi di algoritmi per tutti i tipi di usi, dall'ordinamento dei numeri, alla creazione, gestione, e sostegno di Internet. Si può pensare agli algoritmi come alla struttura su cui si basano i programmi per il computer.

3.6 I linguaggi di programmazione per i robot

Il cervello di un robot è un programma per computer scritto in un linguaggio di programmazione. I programmatori di robot conoscono numerosi linguaggi di programmazione, poi di volta in volta ne scelgono uno a seconda di cosa vogliono che il robot realizzi, a quale sono abituati, all'hardware presente sul robot e così via. Per il fatto, dunque, che non esiste un linguaggio di programmazione in assoluto migliore degli altri, i controllori del robot possono essere implementati in vari linguaggi.

I linguaggi che sono stati utilizzati vanno dai general purpose a quelli appositamente progettati (specific purpose). Questi ultimi sono nati con lo scopo di programmare più facilmente lo specifico controllo dell'architettura del robot. Da poco tempo la robotica sta crescendo e maturando come campo, perciò con il passare degli anni ci saranno sempre più linguaggi di programmazione e strumenti specifici per gli scopi richiesti.

3.7 Il concetto di architettura

A prescindere da quale linguaggio di programmazione è stato utilizzato per programmare un robot, ciò che conta è l'architettura di controllo impiegata per implementare il controllore, in quanto non tutte le architetture sono uguali. Al contrario, alcune architetture impongono regole forti e vincoli su come strutturare i programmi dei robot, di conseguenza perfino il software di controllo risulterà molto diverso.

Esistono pochissimi tipi di controllo, essi sono:

1. controllo deliberativo,
2. controllo reattivo,
3. controllo ibrido,
4. controllo basato sul comportamento.

Nella maggior parte dei casi, è impossibile dire, semplicemente osservando il comportamento di un robot, quale architettura di controllo utilizza. Il motivo è che architetture diverse possono realizzare lo stesso lavoro, soprattutto per robot semplici.

Le architetture di controllo differiscono sostanzialmente nel modo in cui trattano i seguenti aspetti importanti:

- il tempo: quanto velocemente succedono le cose? Tutti i componenti del controllore vengono eseguiti alla stessa velocità?
- la modularità: quali sono i componenti del sistema di controllo?
- la rappresentazione: come fa il robot a conoscere e conservare il suo cervello?

3.7.1 Il tempo

Il tempo, di solito chiamato *periodo di tempo* (time-scale), si riferisce a quanto velocemente il robot deve rispondere all'ambiente rispetto a quanto rapidamente può percepire e pensare. Questo è un aspetto chiave del controllo e, quindi, di grande influenza sulla scelta dell'architettura da usare.

I quattro tipi di architettura di base si differenziano significativamente per la gestione del tempo. Il *controllo deliberativo* guarda al futuro, in modo da funzionare su un periodo di tempo lungo. Al contrario, il *controllo reattivo* risponde per il presente, pone le richieste in tempo reale all'ambiente senza guardare nel passato o nel futuro, per farlo funzionare su un breve periodo di tempo. Il *controllo ibrido* unisce il lungo periodo del controllo deliberativo e il breve periodo del controllo reattivo con una certa abilità. Infine, il *controllo basato sul comportamento* lavora per unire i periodi di tempo.

3.7.2 La modularità

La *modularità* si riferisce al modo in cui il sistema di controllo (programma del robot) è suddiviso in pezzi e componenti, chiamati moduli, e come questi ultimi interagiscono tra loro per produrre il comportamento globale del robot. Nel *controllo deliberativo*, il sistema di controllo è costituito da più moduli tra cui la percezione, la pianificazione e l'azione. I moduli svolgono il lavoro in sequenza, visto che l'uscita di un modulo fornisce l'input per il successivo. Nel *controllo reattivo*, le elaborazioni vengono svolte contemporaneamente, non una alla volta. Molteplici moduli sono attivi parallelamente ed in grado di inviare messaggi agli altri in vari modi. Nel *controllo ibrido*, ci sono tre moduli principali del sistema: la parte deliberativa, la parte reattiva e la parte in mezzo. Le tre parti lavorano in parallelo e sono in grado di interagire fra loro. Nel *controllo basato sul comportamento*, di solito ci sono più di tre moduli principali in grado di lavorare in parallelo ed interagire fra loro, ma in un modo diverso rispetto ai sistemi ibridi. Quindi, il numero di moduli che ci sono, il contenuto di ogni modulo, se lavorano in sequenza o in parallelo, la possibile comunicazione fra moduli, risultano le caratteristiche distintive delle architetture di controllo.

3.7.3 La rappresentazione

In molte attività ed ambienti, il robot non può percepire immediatamente tutto ciò di cui ha bisogno. A volte, quindi, è utile ricordare quello che è accaduto in passato per prevedere cosa accadrà in futuro oppure memorizzare le mappe dell'ambiente, le immagini di persone o di luoghi per ottenere informazioni significative da lavori svolti nel passato.

La *rappresentazione* è la forma in cui l'informazione viene memorizzata o codificata nel robot.

In informatica ed in robotica, si pensa alla *memoria* come ad un dispositivo di archiviazione utilizzato per conservare le informazioni. Riferendosi solo alla memoria, non viene detto niente a riguardo di quello che viene memorizzato e su come viene codificato. La rappresentazione ha il compito di codificare le caratteristiche importanti di ciò che è dentro la memoria.

Naturalmente, cosa viene rappresentato e come esso viene rappresentato ha un forte impatto sul controllo del robot.

La rappresentazione del mondo è in genere chiamata un *modello del mondo*. Una mappa è l'esempio più comunemente usato per il modello del mondo. Per spiegare come la rappresentazione di un mondo particolare, la sua mappa, può variare in forma, bisogna considerare il problema di esplorazione di un labirinto.

Cosa il robot può conservare/ricordare per agevolare la sua navigazione in un labirinto?

- Il robot si può ricordare il percorso esatto che ha adottato per arrivare alla fine del labirinto. Tale percorso ricordato è un tipo di mappa per orientarsi nel labirinto. Si tratta di un percorso odometrico.

- Il robot si può ricordare una sequenza di movimenti che ha compiuto per raggiungere un particolare punto di riferimento nell'ambiente. Questo è un altro modo per memorizzare un percorso attraverso il labirinto, in particolare, un percorso basato su punti di riferimento.

- Il robot si può ricordare cosa fare in ogni punto di riferimento nel labirinto. Questa risulta una mappa basata su punti di riferimento, è più di un percorso in quanto dice al robot cosa fare in ogni incrocio, non interessandosi dell'ordine con il quale l'incrocio viene raggiunto. Una raccolta di punti di riferimento collegati con linee viene chiamata *mappa topologica* perché descrive la *topologia*, ossia le connessioni tra i punti di riferimento. Le mappe topologiche sono molto utili. D'altra parte, non bisogna confonderle con le *mappe topografiche*, le quali sono completamente diverse anche se hanno nomi simili.

- Il robot si può ricordare una mappa del labirinto "tracciata" utilizzando le esatte lunghezze dei corridori e le distanze tra le pareti che vede. Questa è una mappa metrica del labirinto ed è molto utile.

Quelli elencati non sono tutti i modi in cui il robot può costruire e conservare un modello del labirinto. Tuttavia, questi quattro tipi di modelli mostrano alcune importanti differenze nel modo in cui le rappresentazioni possono essere utilizzate. Il primo modello, il percorso odometrico, è molto specifico e dettagliato, ma risulta utile solo se il labirinto non cambia mai, ovvero se nessun raccordo si blocca o si apre, inoltre se il robot è capace di mantenere le distanze e gira con estrema attenzione. Il secondo approccio, come il primo, è vincolato al fatto che la mappa deve restare invariata, tuttavia non richiede al robot di prendere misure specifiche, perché si basa sulla ricerca di punti di riferimento (in questo caso, raccordi). Il terzo modello è simile al secondo, ma connette i vari percorsi in una mappa basata su punti di riferimento, creando una rete dei punti di riferimento memorizzati e delle loro connessioni. Infine, il quarto approccio è il più complicato, perché in esso il robot deve effettuare più misure sull'ambiente e, di conseguenza, memorizzare più informazioni. D'altra parte, è anche, in generale, il più utile, in quanto con esso il robot riesce ad utilizzare la sua mappa e pensare ad altri percorsi possibili nel caso in cui degli incroci diventassero bloccati.

3.8 Il controllo deliberativo

La deliberazione si riferisce al concetto di pensare bene a qualcosa, viene definita come "la riflessione nelle decisioni e nelle azioni". Il controllo deliberativo nasce prima di tutto in ambito dell'intelligenza artificiale (AI).

In AI, i sistemi deliberativi sono stati utilizzati per risolvere problemi come ad esempio giocare a scacchi, dove il pensiero fisso è esattamente la scelta della cosa giusta da fare. Nei giochi ed in alcune situazioni nel mondo reale, prendere tempo per valutare i possibili risultati di una azione è un approccio sia praticabile (c'è tempo per farlo) che necessario (senza una strategia, le cose vanno male). Negli anni 1960 e 1970, i ricercatori AI sono stati molto affezionati a questo tipo di ragionamento. In tale periodo hanno teorizzato che il cervello umano funziona con tale logica, di conseguenza hanno sostenuto che anche il controllo del robot avrebbe dovuto seguirla.

Nel 1960, i primi robot erano basati spesso sull'utilizzo di sensori di visione, i quali richiedevano l'elaborazione di una grande quantità di dati, perciò il tempo perso dal robot (dato dalla lentezza dei processori dell'epoca) per riflettere su come agire era giustificato dalla complessità affrontata per decifrare l'ambiente circostante.

3.8.1 Il concetto di pianificazione

La *pianificazione* è il processo in cui si guardano gli esiti futuri delle attività possibili e si ricerca la sequenza delle azioni che realizzano l'obiettivo desiderato.

La *ricerca* è parte integrante della pianificazione. Essa implica di guardare attraverso la rappresentazione disponibile "in ricerca dello" stato obiettivo. A volte, la ricerca della rappresentazione completa è necessaria anche se può essere molto lenta, a causa delle grandi dimensioni della rappresentazione. Altre volte, invece, risulta sufficiente solo una ricerca parziale, in quanto basta fermarsi alla prima soluzione trovata.

Nel caso in cui un robot deve raggiungere un determinato luogo, il percorso più breve, di solito, è considerato il migliore. Questo perché il robot scegliendo il percorso più breve utilizzerà meno tempo e meno batteria per raggiungere il luogo desiderato. Tuttavia, possono essere utilizzati altri criteri, come ad esempio il percorso più sicuro o quello meno affollato. Il processo di perfezionare la soluzione di un problema per trovare quella migliore si chiama *ottimizzazione*. Il problema viene ottimizzato sulla base di determinati valori o proprietà dello stesso, chiamati *criteri di ottimizzazione*. Tali criteri possono entrare in conflitto fra loro, così scegliere cosa e come ottimizzare non è semplice. L' *ottimizzazione delle ricerche* (optimizing search) studia molteplici soluzioni, in alcuni casi tutti i percorsi possibili. In generale, al fine di utilizzare la ricerca e la pianificazione di una soluzione per un determinato problema, è necessario rappresentare il mondo come un insieme di stati. In seguito, viene svolta una ricerca per trovare un cammino che il robot può percorrere per andare dallo stato attuale allo stato obiettivo. Se il robot vuole trovare il percorso migliore deve prima cercare tutti i percorsi possibili, poi scegliere quello che risulta ottimale in base al criterio di ottimizzazione scelto.

3.8.2 I costi di pianificazione

Maggiore è la dimensione dello spazio degli stati, più lenta sarà la pianificazione. Se aumenta il tempo per generare un piano, aumenta anche il tempo necessario a risolvere il problema. Nella robotica, questo aspetto è particolarmente importante in quanto un robot deve essere in grado di evitare un pericolo immediato, come ad esempio le collisioni con gli oggetti. Pertanto, se il processo di pianificazione richiede troppo tempo, il robot può fermarsi ed aspettare che la pianificazione finisca e solo dopo continuare, oppure proseguire senza un piano finito rischiando di incorrere in collisioni o percorsi bloccati. Il fatto che *ogniqualvolta viene coinvolto un ampio spazio degli stati, la pianificazione risulta difficile*, non è un problema presente solo in robotica. Per far fronte a questo problema, i ricercatori dell'intelligenza artificiale hanno trovato varie soluzioni. Un approccio diffuso è di utilizzare delle gerarchie di stati. Tale strategia prevede che in principio, solo un piccolo numero di 'ampi', 'grossolani' o 'astratti' stati vengono presi in considerazione, poi anche gli stati più raffinati e dettagliati vengono utilizzati nelle varie parti dello spazio degli stati. Gli altri metodi, per velocizzare la ricerca e la pianificazione, si basano sulla ottimizzazione della pianificazione stessa ed includono sempre qualche tipo di compromesso.

Per l'intelligenza artificiale, il continuo miglioramento della potenza di calcolo si traduce nell'aumento della velocità di ricerca negli spazi degli stati. Tuttavia, c'è ancora un limite a cosa può essere eseguito in *tempo reale* (real time), ossia il tempo in cui un robot fisico si sposta in un ambiente dinamico.

Le architetture deliberative, basate su piani comprendono tre passi che necessitano di essere compiuti in successione:

1. percezione (S)
2. pianificazione (P)
3. azione (A), esecuzione del piano.

Le architetture deliberative, che sono anche chiamate *architetture SPA (sense-plan-act)*, presentano quattro svantaggi rilevanti per la robotica.

Svantaggio 1: Periodo di tempo I robot hanno tipicamente collezioni di sensori: alcuni digitali, sia semplici che complessi, altri analogici. Gli ingressi formati da questi due sensori compongono un ampio spazio

degli stati. Nel caso in cui tali sensori vengono accoppiati a modelli interni o rappresentazioni, il risultato è uno spazio degli stati ampio e lento da consultare.

Se il processo di pianificazione risulta lento rispetto alla velocità di movimento del robot, quest'ultimo deve fermarsi ed attendere che il piano sia finito. Di conseguenza, per proseguire in modo scorrevole nell'esecuzione, è meglio interporsi fra pianificare il più possibile e muoversi il più raramente possibile. Questo approccio incoraggia il controllo ad anello aperto, ma che non risulta una buona soluzione in ambienti dinamici. Tuttavia, se la pianificazione è veloce, allora l'esecuzione non necessita di essere in anello aperto, dato che essa può essere eseguita ad ogni passo, in genere, purtroppo, questo è impossibile per risolvere i problemi del mondo reale e dei robot.

La generazione di un piano per un ambiente reale può essere molto lenta.

Svantaggio 2: Spazio Può essere necessario molto spazio per rappresentare e manipolare la rappresentazione dello spazio degli stati del robot. La rappresentazione deve contenere tutte le informazioni necessarie per la pianificazione e l'ottimizzazione. Attualmente, la memoria del computer è relativamente a buon mercato, così lo spazio non risulta più un problema, al contrario di un tempo. Tuttavia, tutta la memoria è finita ed alcuni algoritmi possono esaurirla.

La generazione di un piano per un ambiente reale può richiedere un alto impiego di memoria.

Svantaggio 3: Informazione Il progettista ritiene che lo spazio degli stati sia accurato ed aggiornato. Si tratta di una ipotesi ragionevole, perché se la rappresentazione non è accurata e aggiornata, il piano conseguente risulta inulte. La rappresentazione utilizzata dal progettista deve essere aggiornata e verificata con la frequenza necessaria a mantenere sufficientemente accurato il compito. Infatti, più informazioni ci sono meglio è.

La generazione di un piano per l'ambiente reale necessita dell'aggiornamento del modello del mondo, processo che richiede tempo.

Svantaggio 4: Uso dei piani Un piano è accurato solo se:

- durante l'esecuzione del piano, l'ambiente non cambia in un modo che lo influenza
- il robot conosce in ogni momento in quale stato del mondo e del piano si trova
- gli effettori del robot sono sufficientemente accurati per l'esecuzione di ogni fase del piano, al fine di rendere possibile il passaggio successivo.

L'esecuzione di un piano, anche quando risulta disponibile, non è un processo banale.

Come risulta dai prodotti robotici a partire dal 1980, le architetture puramente deliberative non vengono più utilizzate per la maggior parte dei robot fisici, perché la combinazione dei sensori del mondo reale, degli effettori e delle sfide temporali le hanno rese inattuabili. Tuttavia, ci sono eccezioni, in quanto alcune applicazioni richiedono una grande quantità di pianificazione preventiva e non implicano alcuna pressione temporale, mentre allo stesso tempo presentano un ambiente statico ed a basso contenuto di incertezza nell'esecuzione. Tali ambiti di applicazione sono rari, ma esistono.

Nel settore della robotica, l'approccio SPA non è mai stato abbandonato, anzi è stato ampliato. Considerati i problemi con gli approcci puramente deliberativi, sono stati realizzati i seguenti miglioramenti:

- La ricerca/progettazione è lenta, pertanto si salvano o si mettono nella cache le decisioni importanti e/o urgenti.
- L'esecuzione ad anello aperto è inadeguata, così si utilizza il sistema retroazione ad anello chiuso e si sta pronti a rispondere o ripianificare quando il piano fallisce.

3.9 Il controllo reattivo

Il controllo reattivo è uno dei metodi più comunemente usati per il controllo dei robot. Esso si basa su una stretta connessione tra i sensori e gli effettori del robot. I sistemi puramente reattivi non utilizzano alcuna rappresentazione interna dell'ambiente e non guardano i possibili risultati futuri delle

loro azioni, pertanto operano nel breve periodo di tempo e reagiscono alle informazioni sensoriali correnti.

I *sistemi reattivi* utilizzano una corrispondenza diretta tra i sensori e gli effettori, ed, eventualmente, informazioni minimali sullo stato. Essi sono composti da raccolte di norme che associano determinate situazioni ad azioni specifiche. Si può pensare alle regole reattive come simili ai *riflessi*, ovvero risposte innate che non comportano alcun pensiero. I riflessi vengono controllati dalle fibre nervose del midollo spinale, non dal cervello. Al fine di assicurare una reazione rapida, i riflessi non devono andare fino in fondo al cervello, ma solo nel midollo spinale, il quale è in una posizione centrale rispetto alle altre aree del corpo. I sistemi reattivi sono basati sullo stesso principio: il calcolo complesso è rimosso completamente a favore della rapidità e le risposte elaborate precedentemente vengono immagazzinate.

I sistemi reattivi sono costituiti da un insieme di situazioni (stimoli, chiamati anche condizioni) e da un insieme di azioni (risposte, chiamate anche comportamenti). Le situazioni possono essere basate su input sensoriali o sullo stato interno. Le regole reattive possono essere molto complesse, in quanto coinvolgono combinazioni arbitrarie di input esterni e lo stato interno.

Il modo migliore per mantenere un sistema reattivo semplice e diretto è di avere ogni situazione (stato) unica, così da poter essere individuata dai sensori del robot come scatenata da una sola azione del robot. In tale progetto, si dice che le condizioni sono *mutuamente esclusive*, nel senso che si escludono a vicenda, in quanto solo una condizione alla volta può essere vera.

Tuttavia, spesso è troppo difficile dividere tutte le possibili situazioni (gli stati del mondo) in modo esclusivo, anzi si rischia perfino di richiedere codifiche inutili. Al fine di garantire le condizioni mutuamente esclusive, il controllore deve codificare le regole per tutte le possibili combinazioni dei sensori di input. Tutte quelle combinazioni, quando vengono messe insieme definiscono lo spazio dei sensori del robot. L'aumento del numero e della complessità dei sensori di input comporta ad una crescita con legge combinatoria dello spazio di tutti i possibili sensori di input, vale a dire che lo spazio del sensore diventa rapidamente pesante. In informatica, AI ed in robotica, il termine formale per pesante è *intrattabile*. Per codificare e memorizzare uno spazio di sensori così ampio servirebbe una lookup table enorme, così la ricerca di interi in tale tabella enorme risulterà lenta, a meno

che venga utilizzata una tecnica astuta di ricerca parallela.

Di conseguenza, per realizzare un sistema completamente reattivo, l'intero spazio degli stati del robot (tutti i possibili stati interni ed esterni) devono essere unicamente associati o mappati alle azioni adatte, risultanti nello spazio di controllo completo per il robot.

La progettazione del sistema reattivo, dunque, si evolve con l'insieme completo delle regole. Questo significa che è necessaria una lunga riflessione da parte del progettista (come dovrebbe esserci per qualsiasi processo di progettazione del robot), ma non da parte del robot (in contrasto con il controllo deliberativo, dove molta riflessione viene affidata al sistema).

In generale, le mappature complete tra l'intero spazio degli stati e tutte le possibili risposte non vengono utilizzate nei sistemi reattivi progettati manualmente. Il progettista/programmatore individua le situazioni importanti e ne scrive le regole, mentre le restanti sono coperte con le risposte di default.

Si suppone che viene richiesto di scrivere un controllore reattivo che permette al robot di muoversi e di evitare gli ostacoli. La soluzione è nel creare un robot con due semplici baffi (whiskers) o vibrisse, uno a sinistra ed uno a destra. Ogni baffo restituisce 1 bit, "acceso" o "spento". In particolare, "acceso" indica il contatto con una superficie (cioè il baffo è piegato). Un controllore semplicemente reattivo per seguire le pareti utilizza questi sensori in un modo simile a questo:

- Se il baffo a sinistra è piegato, gira a destra.
- Se il baffo a destra è piegato, gira a sinistra.
- Se entrambi i baffi sono piegati, torna un po' indietro e gira a sinistra.
- Altrimenti, continua ad andare diritto.

Nell'esempio ci sono solo quattro possibili sensori di input, quindi lo spazio dei sensori del robot è di quattro, di conseguenza ci sono quattro regole reattive. L'ultima regola è una opzione predefinita (default), anche se copre uno solo dei casi possibili, ovvero nessun baffo piegato.

Un robot che utilizza il controllore indicato sopra potrebbe oscillare nel caso in cui si trovi un angolo in cui i due baffi si alternano a toccare le

pareti. Come si può aggirare questo, piuttosto tipico, problema?

Esistono due modi diffusi:

1. *utilizzare un po' di casualità:* quando si gira, bisogna scegliere un angolo con ampiezza casuale, invece che fissa. Questo introduce una diversità nel controllore che gli impedisce di rimanere definitivamente bloccato in una oscillazione. In generale, l'aggiunta di un po' di casualità evita qualsiasi situazione in cui si rimane permanentemente bloccati. Tuttavia, potrebbe essere richiesto molto tempo per uscire da un angolo.

2. *mantenere un po' di storia:* ricordare la direzione in cui il robot ha girato nel passaggio precedente (1 bit di memoria) e girare nella stessa direzione, ancora una volta, se la medesima situazione si ripete dopo poco tempo. Tale strategia continua a far girare il robot in una direzione ed alla fine si esce dall'angolo. Tuttavia, esistono ambienti in cui questo approccio non può funzionare.

3.9.1 L'azione di selezione

Nel caso in cui nel controllore vengono inserite le condizioni mutuamente esclusive, i loro risultati non entreranno mai in conflitto, perché può essere individuata solo una situazione/condizione alla volta. Nel caso in cui, viceversa, le condizioni mutuamente esclusive non vengono innescate, più di una regola può essere scatenata dalla medesima situazione, di conseguenza agli effettori vengono inviati contemporaneamente i comandi di due o più azioni differenti.

L' *azione di selezione* è il processo di decisione tra più possibili azioni o comportamenti. Si può selezionare una azione in output o si possono combinare le azioni per produrre un risultato. Questi due approcci sono chiamati rispettivamente arbitrale e di fusione.

Il *comando arbitrale* è il processo di selezione di un comportamento o azione tra più candidati.

Il *comando di fusione* è il processo di associazione fra i molteplici comportamenti o azioni candidati atto a produrre un singolo comportamento/azione in output per il robot.

L'azione di selezione è un problema di rilievo in robotica, un motivo è l'introduzione di una grande quantità di lavoro, teorico e pratico, sui metodi per il comando arbitrale e di fusione. Un sistema reattivo, anche se può ricorrere all'arbitraggio e quindi eseguire una sola azione alla volta, ha ancora bisogno di controllare le sue regole in parallelo, ossia contemporaneamente, in modo da essere pronto a rispondere all'eventuale innesco (scatenazione) di una o più di esse.

Il sistema reattivo deve essere in grado di sostenere il parallelismo, vale a dire la capacità di monitorare ed eseguire le regole contemporaneamente. In pratica, questo significa che il linguaggio di programmazione deve avere di base la capacità del *multitasking*, cioè di eseguire in parallelo più processi/regole/comandi. La capacità del multitasking è critica nei sistemi reattivi, perchè se un sistema non può controllare i suoi sensori in parallelo, e quindi li controlla in sequenza, può perdere un evento, o come minimo la manifestazione di un evento, di conseguenza non riesce a reagire in tempo.

La progettazione di un sistema reattivo per un robot può essere molto complicata, in quanto tante regole devono essere associate per produrre un comportamento efficace, affidabile e goal-driven.

3.9.2 Architettura di sussunzione

L'idea di base della architettura di sussunzione è di costruire i sistemi in modo incrementale, dalla parti semplici a quelle più complesse, utilizzando, per i nuovi elementi aggiunti, il più possibile componenti già esistenti.

La sussunzione dei sistemi è composta da un insieme di moduli o strati ognuno dei quali realizza un compito. Tutti i livelli per realizzare i compiti lavorano contemporaneamente, piuttosto che in sequenza. Questo significa che le regole per ciascuno di essi sono pronte per essere eseguite in qualsiasi momento si presenti la giusta situazione.

I moduli o livelli vengono progettati e aggiunti al robot in modo incrementale. Se si numerano gli strati da 0 in avanti, prima si progetta lo strato 0, poi si implementa e, dopo, si ricercano e correggono gli errori (debug) in esso contenuti. Si suppone che lo strato 0 sia dedicato allo spostamento, ovvero permette al robot di continuare ad andare avanti. Poi si aggiunge lo strato 1 il cui compito è di evitare gli ostacoli, ossia ogni volta che viene rilevato un ostacolo il robot si ferma, poi si gira o arretra. Lo strato 1 può usufruire dello strato 0, in modo che i due insieme permettono al robot di

muoversi senza sbattere contro alle cose che è in grado di rilevare. In seguito si aggiunge lo strato 2, il cui compito è di cercare le porte mentre il robot va in giro in modo sicuro. Si continua così, fino a quando tutte le attività desiderate possono essere raggiunte dal robot attraverso l'associazione dei suoi strati.

Un principio della architettura di sussunzione è che, all'interno di un sistema reattivo, gli strati più alti possono disabilitare temporaneamente uno o più strati sotto ad essi. Tale manipolazione viene realizzata in due soli modi.

1. Gli input di uno strato/modulo possono essere soppressi; in questo modo il modulo non riceve input sensoriali, quindi non calcola le reazioni e non invia output agli effettori o ad altri moduli.

2. Gli output di uno strato/modulo possono essere inibiti; in questo modo riceve input sensoriali, ma non può controllare alcun effettore o altro modulo.

Il nome "Architettura di sussunzione" nasce dall'idea di un livello superiore che può presumere l'esistenza di livelli più bassi e di obiettivi che essi devono raggiungere, in modo che i livelli superiori possono utilizzare quelli inferiori per aiutarli a realizzare i loro obiettivi, servendosi di essi mentre sono in esecuzione o bloccandoli selettivamente. In questo modo, i livelli superiori "sussumono" quelli inferiori.

Lo stile della sussunzione genera diversi vantaggi per l'organizzazione di sistemi reattivi. In primo luogo, la progettazione e il debug viene svolto in maniera incrementale, evitando così di perdersi nella complessità totale del compito del robot. In secondo luogo, se gli strati dei livelli superiori di una sussunzione del robot falliscono, quelli del livelli inferiori continuano regolarmente a lavorare.

Il progetto del controllore di sussunzione è definito *bottom-up*, poiché procede dagli elementi più semplici a quelli più complessi ovvero, per essere più precisi, i livelli vengono aggiunti in modo incrementale. Questo approccio è una buona pratica ingegneristica, anche se in realtà la sua nascita è stata ispirata dallo studio in biologia del processo evolutivo. L'aspetto interessante di tale processo è la modalità con cui la natura introduce nuove capacità agli esseri viventi sulla base di quelle già possedute da essi.

Costruire in modo incrementale aiuta sia la progettazione che il debug, mentre l'utilizzo di strati è utile per modulare il controllo del robot.

Altrimenti, se tutto fosse messo insieme, sarebbe difficile da progettare, correggere, modificare ed, in seguito, da migliorare.

Per rendere efficace la modularità, però, è necessario che i moduli non siano tutti collegati fra loro, altrimenti viene vanificata l'intenzione di dividerli. L'obiettivo, dunque, è di avere pochi collegamenti tra strati diversi. Le sole connessioni fra strati disposte sono quelle usate per la chiusura e la eliminazione. All'interno degli strati, sicuramente, ci sono molte connessioni, visto che molteplici regole devono essere associate per produrre un comportamento in grado di realizzare i compiti. Chiaramente, è necessaria più di una regola per ottenere un robot capace di portare a termine il suo incarico, come ad esempio evitare gli ostacoli. Ma a parte le regole per compiere i rispettivi incarichi, se il sistema viene organizzato con tale struttura diventa più gestibile per la progettazione e la manutenzione.

Di conseguenza, nell'architettura di sussunzione, si usano connessioni *fortemente accoppiate* all'interno degli strati, mente connessioni *debolmente accoppiate* per i collegamenti tra gli strati.

Nella fase di progettazione si decide quali attività deve svolgere lo strato di sussunzione e quelli ad esso superiori ed inferiori, sulla base delle caratteristiche specifiche del robot, dell'ambiente e dei compiti. Non esiste uno schema rigido, poiché a seconda delle situazioni alcune soluzioni risultano migliori di altre. Per questo motivo, la maggior parte delle competenze che i progettisti di robot possiedono, le hanno acquisite tramite prove ed errori.

I principi guida dell'architettura di sussunzione sono:

- i sistemi sono costruiti dal basso verso l'alto (bottom up);
- i componenti sono in grado di realizzare azioni/comportamenti (non moduli funzionali);
- i componenti sono organizzati in strati;
- gli strati più bassi gestiscono i compiti più elementari;
- i componenti aggiunti recentemente possono utilizzare quelli esistenti. Ogni componente fornisce e non fa interrompere uno stretto accoppiamento tra la percezione e l'azione;
- non vi è l'uso di modelli interni, "il mondo è il suo miglior modello".

L'architettura di sussunzione non è l'unico metodo per strutturare i sistemi reattivi, ma è molto diffusa per la sua semplicità e robustezza. É stata ampliamente utilizzata in vari robot, in quanto in grado di interagire con successo ad ambienti incerti e soggetti a cambiamenti dinamici.

Le regole necessarie alla costruzione di un sistema reattivo dipendono dal compito, dall'ambiente e dai sensori sul robot. Un completo sistema reattivo può essere determinato per qualsiasi robot, compito e ambiente che possono essere stabiliti *in anticipo*. Tuttavia, tale sistema può risultare eccessivamente grande, in quanto può richiedere un numero enorme di regole.

3.10 Il controllo ibrido

Il controllo reattivo è veloce ma rigido, mentre il controllo deliberativo è intelligente ma lento. L'idea di base del controllo ibrido è di ottenere il meglio dei due mondi: la velocità del controllo reattivo ed il cervello del controllo deliberativo.

Il *controllo ibrido* comporta la combinazione del controllo reattivo e deliberativo all'interno di un unico sistema di controllo del robot. Questa combinazione significa fondamentalmente che diversi controllori, periodi di tempo (corti per i reattivi, lunghi per i deliberativi) e rappresentazioni (nessuna per i reattivi, modelli del mondo espliciti ed elaborati per i deliberativi) devono essere realizzati, per far sì che i due sistemi siano in grado di lavorare insieme in modo efficace.

Al fine di ottenere il meglio dai due mondi, un sistema ibrido tipicamente è formato da tre componenti, che possono essere chiamati strati o moduli (da non confondere con gli strati/moduli utilizzati ne sistemi reattivi):

- uno strato reattivo;
- un progettista;
- uno strato intermedio che collega i due elementi precedenti insieme.

Di conseguenza, le architetture ibride sono spesso dette *architetture a tre strati* ed i sistemi ibridi *sistemi a tre strati*.

Allo strato intermedio è affidato un lavoro impegnativo, perché deve:

- compensare i limiti sia del progettista che del sistema reattivo;
- conciliare i loro diversi periodi di tempo;
- affrontare le loro diverse rappresentazioni;
- conciliare tutti i comandi contrastanti che posso essere inviati al robot.

La sfida principale del controllo ibrido è di raggiungere il giusto compromesso tra le parti deliberative e reattive del sistema.

3.10.1 Occuparsi dei cambiamenti del mondo, della mappa e dell'incarico

Nel momento in cui il sistema reattivo scopre di non poter fare il suo lavoro, può informare lo strato deliberativo su questo nuovo sviluppo. Lo strato deliberativo può utilizzare queste informazioni per aggiornare la sua rappresentazione del mondo, in modo che possa ora ed in futuro, generare piani più precisi e più utili.

Questa strategia risulta valida sotto due punti di vista. Il primo, è la necessità di aggiornare il modello interno quando le cose cambiano. Il secondo, perché l'aggiornamento dei modelli interni e la generazione dei piani richiedono tempo ed elaborazione. L'input dallo strato reattivo fornisce una precisa indicazione sul tempo necessario per effettuare tale aggiornamento.

3.10.2 Pianificazione e ripianificazione

Ogni volta che lo strato reattivo scopre di non poter procedere, può essere usato come un segnale rivolto al sistema deliberativo su cui riflettere al fine di generare un nuovo piano. Questo approccio è detto *ripianficazione dinamica*.

Tuttavia, non tutta l'informazione circola dal basso verso l'alto, dallo strato reattivo a quello deliberativo. In realtà, lo strato deliberativo che fornisce il percorso verso l'obiettivo, dà al robot le indicazioni da seguire, le curve da prendere, le distanze da percorrere. Se il progettista sta utilizzando il computer mentre il robot è in esecuzione, può inviare un messaggio allo strato di navigazione reattiva per farlo fermare, spostare e condurre in una direzione differente perché nella mappa è stata scoperta una via migliore.

In generale, un piano completo, quanto è finito, risulta la miglior risposta deliberatrice in grado di generare. Tuttavia, a volte, il tempo non è sufficiente per aspettare la risposta completa ed ottimale. In questi casi, spesso è meglio per il robot iniziare a percorrere la direzione giusta, ma nel frattempo continuare la generazione di un piano più accurato e dettagliato, oltre all'aggiornamento dello stato di navigazione necessario.

Sicuramente i tempi di questi due processi, la navigazione reattiva e la programmazione deliberativa, non sono sincronizzati. Di conseguenza, si potrebbe verificare che il robot una volta raggiunta una meta non sa come andare avanti, quindi viene fermato e messo in attesa che il progettista gli indichi il passo successivo. In alternativa, il progettista può far aspettare il robot prima che ricominci a muoversi oppure farlo uscire dalla zona affollata, in modo che capisca esattamente dove si trova, per poi essere capace di generare un piano utile.

3.10.3 Evitare la ripianificazione

Una idea utile dei ricercatori nell'ambito della pianificazione è stata quella di ricordare / salvare / memorizzare i piani, in modo che non venissero generati di nuovo in futuro. Naturalmente, ogni piano è specifico per i determinati stato iniziale e stato obiettivo, ma se proprio quelli sono a rischio di ripetizione, conviene conservare il piano per un utilizzo futuro.

Questa idea è davvero importante per le situazioni che si verificano spesso e hanno bisogno subito di una decisione. I progettisti, invece di pensare come affrontare una situazione ogni volta, programmano in anticipo il controllore con alcune risposte.

L'idea di base della memorizzazione e del riutilizzo di mini piani per le situazioni che si ripetono è stata usata nelle "tabelle di contingenza", ossia tabelle di lookup che dicono al robot cosa fare (come nel sistema reattivo), e restituiscono un piccolo piano come risposta (come nel sistema deliberativo). Nel caso specifico dei livelli intermedi, i piani che sono stati calcolati (nella modalità offline o durante la vita del robot) vengono memorizzati per una più veloce ricerca futura.

La priorità assoluta non è sempre posseduta dallo strato deliberativo o reattivo, ma dipende da vari fattori. Ad esempio, dal tipo di ambiente, di incarico, dalle percezioni, dalla tempestività, dalle reazioni richieste. Alcuni sistemi ibridi utilizzano una struttura gerarchica. In alcuni casi, il piano

dello strato deliberativo è "la legge", mentre in quelli reattivi si considera soltanto come dei consigli, i quali potrebbero essere anche ignorati. Nei sistemi più efficaci l'interazione fra il pensare e l'agire è collegata, così che ciascuno può informare ed interrompere l'altro. Ma, per sapere chi dovrebbe essere incaricato al momento, è necessario prendere in considerazione i diversi modi posseduti dal sistema per scoprirlo. Ad esempio, se il progettista ha un nuovo piano sufficientemente migliore di quello attualmente in esecuzione, può interrompere lo strato reattivo. D'altra parte, lo strato reattivo può sospendere il progettista nel caso in cui trova un percorso bloccato e non può procedere, ma ciò deve essere fatto solo dopo che ha cercato di aggirare in vari modi la barriera.

3.10.4 Pianificazione on-line e off-line

La *pianificazione off-line* avviene mentre il robot è in fase di sviluppo, perciò esso non ha molto di cui preoccuparsi. Al contrario, nella *pianificazione on-line* il robot ha molte preoccupazioni mentre lo si sta interrogando sul suo lavoro svolto e sui suoi obiettivi raggiunti.

Un *piano universale* è un insieme dei possibili piani, per tutti gli stati iniziali e tutti gli obiettivi, nello spazio degli stati di un determinato sistema.

Se per ogni situazione del sistema, un robot possiede un preesistente piano ottimale, allora deve solo cercarlo, poi potrà reagire sempre in maniera ottimale, quindi avere sia le capacità reattive che deliberative senza riflettere affatto. Tale robot risulta reattivo, dal momento che la pianificazione è realizzata completamente off-line e non run-time.

Un'altra caratteristica positiva dei piani precompilati è che le informazioni possono essere inserite nel sistema in un modo pulito. Queste informazioni sui robot, gli incarichi e sugli ambienti sono chiamate *dominio di conoscenza*. Tale dominio è compilato da un controllore reattivo, in modo che le informazioni non devono essere ragionate (o pianificate) on-line, ossia in tempo reale, ma diventano, invece, una serie di regole reattive in tempo reale che possono essere cercate.

Questa idea si è così diffusa che anche i ricercatori hanno sviluppato un modo di generare tali piani precompilati automaticamente, utilizzando uno speciale linguaggio di programmazione e un compilatore. I programmi per robot scritti in quel linguaggio hanno prodotto un tipo di piano universale. Per fare questo, il programma ha come input una descrizione matematica del

mondo e degli obiettivi del robot, mentre come output genera un "circuito" di controllo (un diagramma che evidenza i collegamenti fra i componenti) per una "macchina" reattiva. Queste macchine vengono chiamate *automi situati* (situated automata). Essi non sono reali, ossia macchine fisiche, ma sono formali (idealizzate), infatti gli input sono collegati a sensori astratti mentre gli output ad effettori astratti. Essere *situato* significa esistere in un mondo complesso ed interagire con esso, gli automi sono macchine di elaborazione con particolari proprietà matematiche.

Purtroppo, questo è troppo bello per essere vero per i robot del mondo reale. Ecco perché:

- Lo spazio degli stati è troppo grande per la maggior parte dei problemi realistici, così produrre o memorizzare un piano universale è semplicemente impossibile.

- Il mondo non deve cambiare; se lo fa, nuovi piani dovranno essere generati in reazione al cambiamento dell'ambiente.

- Gli obiettivi non devono cambiare; come avviene nei sistemi reattivi dove se gli obiettivi cambiano, almeno alcune delle regole hanno la necessità di cambiare.

Gli automi situati non possono essere collocati nel mondo reale. Non è facile scrivere le specifiche matematiche che descrivono il mondo reale in cui i robot esistono, ma è l'approccio richiesto agli automi situati.

Si torna ad avere a che fare con la deliberazione e la reazione in tempo reale, i sistemi ibridi sono un buon sistema per realizzarli. Comunque essi hanno anche i loro inconvenienti tra cui:

- Lo strato intermedio risulta difficile da progettare ed attuare, esso tende ad essere special purpose, ossia creato per il robot e il proprio compito specifico, in modo che deve essere reinventato per quasi ogni nuovo robot e compito.

- Il meglio dei due mondi può finire per essere il peggiore dei due mondi; se il sistema ibrido viene gestito male può degenerare fino ad avere che il progettista rallenta il sistema reattivo o il sistema reattivo ignora del tutto il progettista, di conseguenza si ha una riduzione dell'efficacia di entrambi.

- Un sistema ibrido efficace non è facile né da progettare, né da collaudare. Tuttavia, risulta così per qualsiasi sistema di robot.

Nonostante gli inconvenienti dei sistemi ibridi, sono la scelta più diffusa per molti problemi nel settore della robotica, in particolare quelli che coinvolgono un singolo robot, il quale è chiamato a svolgere una o più attività che richiedono ragionamenti di qualche tipo, come ad esempio reagire ad un ambiente dinamico.

3.11 Il controllo basato sul comportamento

Il *controllo basato sul comportamento* è un altro diffuso modo di controllare i robot che incorpora il meglio dei sistemi reattivi, ma non comporta una soluzione ibrida.

Il controllo basato sul comportamento (BBC) si è sviluppato dal controllo reattivo, ed è stato allo stesso modo ispirato dai sistemi biologici. In realtà, tutti gli approcci di controllo (reattivo, deliberativo, ibrido e basato sul comportamento) sono ispirati alla biologia. Questo dimostra che i sistemi biologici sono così complessi che possono servire da ispirazione per una varietà di metodi diversi per il controllo. Tuttavia, il mondo naturale risulta ancora più complicato e più efficace di qualsiasi opera artificiale che sia stata fatta finora. Nel momento in cui ci si inizia a sentire soddisfatti dei sistemi artificiali, tutto quello che bisogna fare è uscire fuori all'aperto e guardare tutti gli errori (bugs) commessi, in modo da rendersi conto di quanta strada si deve ancora fare.

L'ispirazione principale per la creazione del controllo basato sul comportamento proviene da diverse sfide fondamentali:

- I sistemi reattivi sono troppo rigidi, incapaci di rappresentanza, di adattamento o di apprendimento.
- I sistemi deliberativi sono troppo lenti e inefficienti.
- I sistemi ibridi richiedono strumenti complessi per l'interazione tra i propri componenti.
- La biologia sembra essersi evoluta in termini di complessità dai componenti semplici e coerenti.

Il BBC risulta più vicino al controllo reattivo rispetto al controllo ibrido e lontano dal controllo deliberativo. In realtà, i sistemi basati sul comportamento hanno componenti reattivi, proprio come i sistemi ibridi, ma non hanno per niente i componenti deliberativi tradizionali.

Il *controllo basato sul comportamento* (BBC) implica l'uso di "comportamenti" come moduli per il controllo. Di conseguenza, i controlli del BBC sono implementati come collezioni di comportamenti. La prima proprietà del controllo basato sul comportamento da ricordare è che tutto riguarda i comportamenti.

Uno dei punti di forza del BBC proviene dai diversi modi in cui le persone hanno codificato e attuato i comportamenti, che a volte sono anche chiamati *moduli per realizzare il comportamento*. Tuttavia, non bisogna dare per scontato che tutto vada bene e che ogni pezzo di codice possa essere un comportamento. Per fortuna, ci sono alcune regole empiriche sui comportamenti, dei vincoli sulla loro modalità di progettazione ed alcune cose da evitare nella loro esecuzione:

- I comportamenti per raggiungere e/o mantenere gli obiettivi specifici. Ad esempio, un *comportamento diretto a casa* (homing behavior) realizza l'obiettivo di portare il robot nella posizione in cui è collocata la sua casa.

- I comportamenti sono estesi nel tempo, non istantanei. Questo significa che ai robot occorre un certo tempo per raggiungere e/o mantenere i loro obiettivi. Dopo tutto, è necessario un po' di tempo per tornare a casa.

- I comportamenti possono ottenere input dai sensori e dagli altri comportamenti. Questo significa che siamo in grado di creare reti di comportamenti in grado di "parlare" fra loro.

- I comportamenti sono più complessi delle azioni. Mentre un sistema reattivo può utilizzare le azioni semplici come fermati e gira a destra, un sistema BBC necessita di un tempo prolungato per realizzare i comportamenti.

I comportamenti possono essere progettati in una varietà di livelli di dettaglio o di descrizione. In modo più formale, viene detto in una varietà di *livelli di astrazione*, in quanto *per astratto* si intende l'allontanamento

dai dettagli al fine di rendere le cose meno specifiche. I comportamenti, a seconda dei casi, possono richiedere diverse quantità di tempo e di calcolo. In breve, sono molto flessibili, questo è uno dei vantaggi chiave del BBC.

La potenza e la flessibilità del BBC non viene solo dai comportamenti, ma anche da parte dell'organizzazione di tali comportamenti, dal modo in cui sono associati in un sistema di controllo. Ecco alcuni principi per un buon progetto di BBC:

- I comportamenti di solito sono eseguiti in parallelo/contemporaneamente, proprio come nei sistemi reattivi, al fine di consentire, se necessario, al controllore di rispondere immediatamente.

- Le reti di comportamenti vengono utilizzate per memorizzare lo stato e costruire i modelli/rappresentazioni del mondo. Quando vengono assemblati in rappresentazioni distribuite, i comportamenti possono essere utilizzati per memorizzare il passato e guardare avanti verso il futuro.

- I comportamenti sono progettati in modo che vengono compiuti in periodi di tempo compatibili. Questo significa che nella progettazione del BBC non è positivo avere alcuni comportamenti molto veloci ed altri molto lenti. La ragione è che questo rende il sistema ibrido in termini di periodo di tempo, risollevando il difficile problema di interfacciamento fra periodi di tempo diversi.

I sistemi basati sul comportamento si differenziano sia dai sistemi reattivi, sia da quelli ibridi. I sistemi basati sul comportamento hanno le seguenti caratteristiche fondamentali:

1. la capacità di reagire in tempo reale;

2. la capacità di utilizzare le rappresentazioni per generare un comportamento efficiente (non solo reattivo);

3. la possibilità di utilizzare una struttura uniforme e la rappresentazione di tutto il sistema (senza strati intermedi).

In alcuni sistemi basati sul comportamento, la struttura interna del comportamento corrisponde esattamente ai comportamenti manifestati esternamente. Tuttavia, non è sempre così, soprattutto per i comportamenti più

complessi, si ha che la maggior parte dei controllori non sono stati progettati in questo modo.

Dopo tutto, avere una corrispondenza intuitiva tra il programma, il controllo interno e quello esterno osservabile dà veramente l'idea di un approccio chiaro e facile da capire. Sfortunatamente, esso risulta essere impraticabile.

I comportamenti osservabili più interessanti non riguardano solo quelli relativi al programma di controllo, ma anche l'interazione dei comportamenti interni fra loro e con l'ambiente in cui il robot è situato. Questa è sostanzialmente la stessa idea utilizzata nei sistemi reattivi: semplici regole reattive possono interagire per produrre un comportamento del robot osservabile ed interessante. Lo stesso vale per i comportamenti del controllo interno.

Si consideri il raggruppamento (flocking), il comportamento in cui un complesso di robot si muove insieme in un gruppo. Un robot che si raggruppa con altri non deve necessariamente possedere un comportamento interno di *raggruppamento*. In realtà, il raggruppamento può essere implementato in maniera molto elegante ed in un modo completamente distribuito.

Questa è una delle filosofie più intelligenti dei sistemi basati sul comportamento. Tali sistemi sono, in genere, progettati in modo che gli effetti dei comportamenti interagiscano con l'ambiente, piuttosto che internamente ad esso, allo scopo di trarre vantaggio dalle *dinamiche di interazione*. In questo contesto, le dinamiche fanno riferimento ai modelli (patterns) e alla cronologia (history) delle interazioni e dei cambiamenti. L'idea che le regole e i comportamenti possono interagire per produrre risultati più complessi è chiamato il *comportamento emergente*.

In generale, nella definizione di un sistema basato sul comportamento, il progettista inizia elencando i comportamenti visibilmente desiderabili manifestati esternamente. Successivamente, il progettista stima il modo migliore per programmare questi comportamenti attraverso comportamenti interni. Tali comportamenti interni possono o non possono realizzare gli obiettivi direttamente osservabili. Per semplificare questo processo, vari compilatori e linguaggi di programmazione sono stati sviluppati.

Di conseguenza, i sistemi basati sul comportamento sono strutturati come collezioni di comportamenti interni, che, quando si spostano in un ambiente, producono un insieme di comportamenti manifestati esternamente. L'insieme dei comportamenti interni e quello dei comportamenti manifestati esternamente non sono necessariamente lo stesso. Inoltre, tale sistema pro-

cede bene solo fino a quando il robot mantiene comportamenti osservabili per raggiungere i propri obiettivi.

In generale, i controllori basati sul comportamento sono reti di comportamenti interni che interagiscono (inviando messaggi agli altri), al fine di produrre il comportamento esterno desiderato, osservabile e manifestato del robot.

Come si è visto in precedenza, le differenze principali tra i vari metodi di controllo si basano sul modo in cui vengono trattati: la modularità, il tempo e la rappresentazione. Nel caso del BBC, l'approccio alla modularità è di utilizzare un insieme di comportamenti, i quali siano relativamente simili fra loro in termini di tempo di esecuzione. Ciò significa che, avendo un comportamento che contiene un modello centralizzato del mondo e svolgendo un ragionamento su di esso, come nei sistemi ibridi, non si adatta ad un comportamento basato sulla filosofia, quindi non è in grado di realizzare un buon controllore.

Poiché il BBC si basa sulla filosofia reattiva (ma non solo su di essa), i comportamenti devono essere aggiunti al sistema in modo incrementale ed eseguiti concorrentemente, in parallelo, anziché in sequenza, ossia uno alla volta. I comportamenti vengono attivati in risposta alle condizioni interne e/o esterne, input sensoriali e stati o messaggi interni ricevuti da altri comportamenti. Le dinamiche di interazione sono presenti sia all'interno del sistema stesso (nell'interazione tra i comportamenti), sia all'interno dell'ambiente (nell'interazione dei comportamenti con il mondo esterno). Tale approccio è simile a quello dei principi e degli effetti dei sistemi reattivi, ma può essere sfruttato in maniera più ricca ed interessante, perché:

I comportamenti sono più espressivi rispetto alle semplici regole reattive.

Si ricorda che i sistemi basati sul comportamento possono avere componenti reattivi. Pertanto, questi sistemi non possono utilizzare affatto le rappresentazioni complesse. La rappresentazione interna non è necessaria per tutti i problemi, quindi per alcuni può essere evitata. Questo non significa che il controllore risultante è reattivo, fintanto che il controllore è strutturato con i comportamenti si crea un sistema di BBC con tutti i benefici che derivano da tale approccio, anche senza l'uso della rappresentazione.

Siccome, i comportamenti più complessi e più flessibili delle regole reattive possono essere utilizzati in modo intelligente per i programmi dei robot, se si hanno comportamenti che interagiscono fra loro all'interno del robot, possono essere utilizzati per la memorizzazione e la rappresentazione. Pertanto, possono servire come base per l'apprendimento e la previsione, e questo significa che i sistemi basati sul comportamento possono ottenere le stesse cose dei sistemi ibridi, ma in modo diverso. Una delle differenze principali è nel modo in cui la rappresentazione viene utilizzata.

3.11.1 La rappresentazione distribuita

La filosofia dei sistemi basati sul comportamento impone che le informazioni utilizzate come rappresentazione interna non possono essere centralizzate o manipolate a livello centrale. Questo è in diretto contrasto con l'approccio dei sistemi ibridi e deliberativi, che in genere utilizzano una rappresentazione centralizzata (come ad esempio una mappa globale) e un meccanismo di ragionamento centralizzato (come un progettista che utilizza la mappa globale per trovare un percorso).

La sfida chiave utilizzata nella rappresentazione del BBC è nel modo in cui la rappresentazione (di qualsiasi forma di modello dello mondo) può essere efficacemente distribuita sulla struttura del comportamento. Al fine di evitare le insidie del controllo deliberativo, la rappresentazione deve essere in grado di agire su un periodo di tempo che si avvicina (se non lo stesso) al tempo reale dei componenti del sistema. Allo stesso modo, per evitare le sfide dello strato intermedio del controllo ibrido, la rappresentazione deve utilizzare la stessa struttura del comportamento di base come il resto del sistema.

Capitolo 4

Un modello di controllore di robot in Jason

4.1 Il mapping

In seguito allo studio di Jason e dei tipi di controllo esistenti in robotica, si è iniziato a pensare ad una applicazione del linguaggio di programmazione ad agenti sui controllori dei robot. La prima scelta è stata di mappare un robot su un agente. Il motivo è la forte relazione tra il concetto di robot e di agente. Dal momento che gli aspetti che caratterizzano un agente, ovvero il ciclo di ragionamento e le percezioni, si mappano molto bene in quelli del robot. Il ragionamento ciclico di un agente potrebbe essere visto come una sorta di approccio SPA dell'architettura deliberativa del robot, dove però la pianificazione del modello BDI di Jason non è creare un piano ma sceglierlo, in base al triggering event ed al contesto, nella libreria dei piani, come avviene nell'architettura reattiva. A questo punto si può affermare, quindi, che Jason per sua natura non è riconducibile completamente né ad una architettura puramente deliberativa né ad una architettura puramente reattiva, così si è introdotto un tipo di architettura che mette insieme le due creando una nuova forma di ibridazione.

Questa nuova tipologia di architettura permette di specificare il comportamento del robot in termini di percezioni, credenze ed azioni.

I sensori sono l'interfaccia percettiva tra il robot e l'ambiente. Il compito dei sensori è di raccogliere le informazioni dall'ambiente e trasmetterle al controllore del robot. Data l'associazione 1-1 fra robot ed agente, i percetti

rilevati attraverso i sensori del robot risultano belief per l'agente e perciò vengono memorizzate nella base delle credenze sottoforma di letterali. Di conseguenza, viene spontaneo immaginare di mappare la base delle credenze dell'agente con la rappresentazione, in quanto quest'ultima è vista come il modello del mondo per il robot. La rappresentazione motiva i tratti di architettura deliberativa presenti nella nuova forma di ibridazione. Si ricorda che nell'architettura reattiva non vi è alcuna sorta di rappresentazione. La rappresentazione nell'architettura deliberativa è utile per costruire il modello del mondo ai fini della pianificazione, in quanto permette al robot di pianificare on-line. Per gli agenti BDI, viceversa, la pianificazione è totalmente off-line. In particolare, un agente possiede una libreria dei piani. Poi, nel momento in cui esso dovrà gestire un evento cercherà un piano, nella libreria dei piani, che si adatta alla situazione corrente, in particolare con triggering event e contesto adeguato. In altre parole, la pianificazione per un agente non è altro che la scelta di un piano fra piani preconfezionati. Nel caso in cui l'agente si trova di fronte ad una situazione che non si abbina a nessuno dei suoi piani, esso non riesce a gestirla perché non sa cosa fare. Si verifica, dunque, il fallimento del piano. Bisogna sottolineare che a fronte del fatto della possibilità di avere più piani rilevanti per eventi diversi, l'agente potrebbe possedere dei piani che in qualche modo potrebbero interagire oppure interferire fra loro. Per la caratteristica di essere off-line, la pianificazione per l'agente risulta essere un processo poco costoso, così da fornire un buon grado di efficienza e di reattività a discapito della flessibilità. In questo aspetto la nuova tipologia di architettura è molto più vicina a quella reattiva (pianificazione totalmente off-line), piuttosto che alla deliberativa.

Il robot ha la possibilità di interagire con l'ambiente oltre che con i sensori, anche attraverso gli attuatori. Si possono definire gli attuatori come i dispositivi che permettono al robot di agire sul mondo. In questa chiave di lettura, le azioni degli agenti possono essere mappate con gli effettori dei robot. Un agente, dunque, può essere visto come un sistema che percepisce il suo ambiente attraverso i sensori e agisce su di esso mediante attuatori. Il comportamento del robot può essere descritto come una sequenza di reazioni provocate da ciò che ha percepito, le quali vengono realizzate attraverso gli effettori. Queste singole reazioni, a livello di agente possono essere mappate sottoforma di azioni e la loro realizzazione in sequenza come un piano.

4.2 Sperimentazioni

Un intento di questa tesi è di riuscire a realizzare un modello software attraverso il quale sarà possibile realizzare sperimentazioni all'interno della facoltà perciò lo scenario che è stato considerato è *Lego-Mindstorms-like*. Questo significa operare attraverso sensori ed attuatori poco costosi perciò con prestazioni modeste, ma che ci permettono di produrre un comportamento significativo.

4.2.1 Un ambiente di simulazione

L'ambiente anche se realizzato esclusivamente via software, cerca di rispondere il più possibile ad un ambiente reale. A questo scopo, l'ambiente non è stato implementato come un semplice modellino a griglia, dove i robot possono solo muoversi in modo discreto ma è stato simulato il fatto che i Lego-Mindstorms possono spostarsi non solo in modo orizzontale o verticale ma anche in obliquo, ovvero facendo riferimento ad un determinato angolo.

Rispettando le caratteristiche degli agenti, i robot si muovono nell'ambiente in maniera autonoma. Nel modello non è presente una entità esterna che li pilota, ma il controllo è proprio su ognuno di essi. Il robot, infatti, non riesce a percepire tutto ciò che avviene nell'ambiente in quel determinato momento ovvero anche gli eventi che si verificano lontano da esso. Nella simulazione è presente il *principio di località*, cioè il robot riesce a percepire quello che è il suo intorno, secondo le capacità dei sensori definiti in sede di progettazione. In altre parole, l'agente ha delle percezioni e reazioni che riguardano una certa località, a seconda però delle prestazioni dei sensori e degli attuatori specificati, potrebbe essere che effettivamente riesce ad eseguire delle azioni che hanno effetti anche a distanza.

Un aspetto che non è stato ritenuto approfondire ma di cui non si può non tenere conto è l'*imprevidibilità dell'ambiente*. Si potrebbe realizzare il comportamento del robot in grado di far fronte alle imperfezioni ed errori, ovvero il non determinismo dell'ambiente. Ad esempio, le ruote del robot potrebbero slittare provocando un avanzamento di un valore diverso da quello atteso. Questo aspetto risulta legato all'approssimazione presente, in generale, nel software.

4.3 Esempio applicativo

La nuova tipologia di architettura viene sperimentata al momento su un semplice esempio, in questo capitolo, ed in seguito sul caso di studio, precisamente nel capitolo successivo.

Ora si espongono nel dettaglio le caratteristiche dell'esempio preso in considerazione, partendo da una sua descrizione.

Si considera un *ambiente* rappresentato come un'arena delimitata da quattro *ostacoli*, essi possono essere immaginati come muri dal momento che creano una sorta di stanza di forma quadrata. All'interno dell'arena possono essere collocati altri ostacoli, i quali possono assumere esclusivamente una posizione orizzontale oppure verticale. Nell'arena è situato un *robot*, in grado di muoversi in qualsiasi direzione (orizzontale, verticale oppure obliqua) all'interno di essa. Il comportamento del robot è di perlustrare in modo autonomo e casuale l'arena, sapendosi districare fra i vari ostacoli e con l'obiettivo di trovare il *target*. Nel momento in cui il target viene trovato il robot arresta la sua marcia.

L'esempio è stato modellato, come già anticipato, attraverso una simulazione, ma si è fatto riferimento ad uno scenario Lego-Mindstorms-like. Tale scelta ha influenzato sia la progettazione dell'arena, sia del robot.

4.3.1 Ambiente

Continuando nell'associazione 1-1 fra robot e agente, risulta spontaneo mappare il concetto di arena per il robot con quello di ambiente per l'agente. Il linguaggio di programmazione ad agenti Jason fornisce un sostegno per l'attuazione di un modello di ambiente in Java. L'architettura in generale di un agente includerà i metodi Java che implementano l'interazione di un agente con il suo ambiente.

Jason fornisce la classe `Environment` che supporta la individualizzazione delle percezioni, così da facilitare il compito di associare le percezioni all'agente. Tale classe possiede due metodi: `init` ed `executeAction`. Il metodo `init` può essere utilizzato per ricevere i parametri per la classe `Environment` di configurazione del sistema multiagente. In generale, è opportuno utilizzare tale metodo per inizializzare la lista delle percezioni con le caratteristiche dell'ambiente che saranno percepibili dall'agente appena

viene avviato. Per ciascuna richiesta di esecuzione di una azione, l'architettura dell'agente invoca il metodo `executeAction` dell'ambiente, e riprende la rispettiva intenzione quando il metodo ritorna; questo mentre il ciclo di ragionamento procede.

Nella progettazione dell'esempio, la classe che estende `Environment` di Jason prende il nome di `ArenaEnv`. I metodi `init` ed `executeAction` che vengono ereditati vengono riscritti (override) per adattarli alle specifiche necessità. In particolare `ArenaEnv` si basa sulle classi `ArenaModel` ed `ArenaView`, ed implementa l'interfaccia `ArenaModelListener`. Le classi `ArenaView`, `ArenaModel` e `ArenaEnv` sono state realizzate rispettando il pattern architetturale Model-View-Control, pietra miliare nell'ingegneria del software.

Nel metodo `init` vengono creati gli oggetti che costituisco la parte grafica dell'ambiente tanto che utilizza le classi `ArenaModel` ed `ArenaView`. In particolare, `init` richiama un metodo privato `initModel`. Tale scelta fa sì da suddividere la gestione della sola arena in `init` e degli oggetti in essa contenuti in `initModel`. Il metodo `executeAction` implementa le azioni di base dell'agente, ovvero quelle permesse dati gli attuatori presenti sul robot. In questo esempio sono per la maggior parte relativi al movimento, ovvero fanno riferimento alle ruote. All'interno di questo metodo viene utilizzato `getFunctor` che viene utilizzato per restituire in forma di stringa l'entità atomica che è la testa (cioè il funtore) della struttura che rappresenta l'azione. Inoltre, viene inserito il metodo `notifyStateUpdated`, in cui vengono aggiunte le percezioni alla lista di percezioni esclusive di ciascun robot. Le percezioni vengono create utilizzando il metodo `parseLiteral` della classe `Literal`.

Il punto di vista grafico dell'ambiente gestito da `ArenaView` ed `ArenaViewPanel`, rispettivamente il primo si occupa del frame e il secondo del pannello.

Avendo come scopo la creazione di un ambiente il più realistico possibile, è stato necessario aggiungere un thread, ovvero un processo, il cui ruolo è di controllore della simulazione. Il suo compito è di aggiornare continuamente l'ambiente in base al comportamento del robot, poiché esso stesso scandisce il tempo logico. In particolare, tale aspetto è legato alla progettazione dell'azione `move_forward(Speed)`, ovvero un robot che ha una certa posizione e che continua ad avanzare ad una determinata velocità. La classe che si occupa del controllore della simulazione è `SimlationController`, si nota

che estende la classe `Thread` di Java.

L'ambiente è stato realizzato in chiave Lego-Mindstorms, per cui l'arena viene raffigurata di colore bianco ed in un punto della stessa viene situato il target rappresentato da una traccia sul pavimento della stanza. All'interno della stanza, oltre al robot ed agli ostacoli sono stati introdotti dei tag RFID[1]. Tali sensori sono disposti sul pavimento dell'arena con una struttura a griglia. La disposizione e il numero di RFID utilizzati nell'ambiente sono un parametro di progetto di rilevo.

Si pone in luce che la configurazione (setup) dell'ambiente tocca diversi aspetti dell'ICT[2] poiché si utilizzano attuatori, sensori oltre alla tecnologia RFID per la localizzazione indoor.

Dettagli sulla realizzazione dell'ambiente in Java

Per maggiore chiarezza, si inseriscono delle parti di codice Java relative alle classi sopra menzionate.

Il metodo `notifyStateUpdated` della classe `ArenaView` si è implementato come segue.

```
package env;

import javax.swing.*;
import java.awt.*;

public class ArenaView extends JFrame
implements ArenaModelListener {

   \\  .......

   public void notifyStateUpdated(double dt) {
      model.acquireReadLock();
```

[1]L'RFID (o Radio Frequency IDentification o Identificazione automatica di oggetti, animali o persone basata sulla capacità di memorizzare e accedere a distanza a tali dati usando dispositivi elettronici (chiamati tag o transponder) che sono in grado di rispondere comunicando le informazioni in essi contenute quando "interrogati". In un certo senso sono un sistema di lettura wireless ovvero"senza fili". Il sistema RFID si basa sulla lettura a distanza di informazioni contenute in un tag RFID usando dei lettori RFID.

[2]Information and communication technology, in sigla ICT, è l'insieme delle tecnologie che consentono di elaborare e comunicare l'informazione attraverso mezzi digitali.

```
        long ms =
            Math.round(model.getCurrentTime()*1000);
        time.setText(""+ms);
        this.repaint();
        model.releaseReadLock();
    }

    \\  .......

}
```

Il metodo `updateState` della classe `ArenaModel` si è implementato come segue.

```
package env;

import javax.swing.*;
import java.awt.*;

public class ArenaView extends JFrame
implements ArenaModelListener {

     \\  .......

    public void updateState(double dt){
       rwLock.writeLock().lock();
       try {
        for (RobotInfo ri: robots.values()){
           P2d oldPos = ri.getPos();
           ri.updatePos(dt);

           boolean colliding = false;
           for (ObjectInfo oi: obstacles){
              if (oi.foundCollision(ri.getPos(),
                 ri.getRadius())){
              ri.setPos(oldPos);
              ri.setColliding(0);
              colliding = true;
              }

           }

           if (!colliding){
              for (ObjectInfo oi: robots.values()){
```

```
        if (!oi.getId().equals(ri.getId()) &&
          oi.foundCollision(ri.getPos(),
          ri.getRadius())){
          ri.setPos(oldPos);
           ri.setColliding(0);
          colliding = true;
       }
    }
}

if (!colliding){
   ri.setNotColliding();
 }

double dist = ri.detectObject(obstacles);
ri.setDetectedObjectDistance(dist);

P2d readerPos = ri.getPos();
double readerRadius =
  ri.getRFIDRadius();
boolean detected = false;
for (RFIDInfo rfid: rfids){
   if (rfid.isDetectable(readerPos,
    readerRadius)){
      ri.setDetectedPos(rfid.getPos());
      detected = true;
      break;
   }
}

if (!detected){
   ri.setDetectedPos(null);
}

double radius = ri.getRadius();
detected = false;
for (TargetInfo target: targets){
   if (target.isDetectable(readerPos,
    radius)){
      ri.setDetectedTarget(target.getId());
      detected = true;
      break;
    }
 }
```

```
            if (!detected){
               ri.setDetectedTarget(null);
            }

         }

        time += dt;
        notifyStateUpdated(dt);

    } finally {
          rwLock.writeLock().unlock();
    }
}

    \\   .......

}
```

La classe `ArenaEnv` si è implementata come segue.

```
package env;
import jason.asSyntax.*;
import jason.environment.Environment;
import java.util.*;
import java.util.concurrent.*;

public class ArenaEnv extends Environment
implements ArenaModelListener {

  ArenaModel model;
  ArenaView view;
  SimulationController controller;
  double arenaSize;

   @Override
    public void init(String[] args) {

    arenaSize = 10;

    model = new ArenaModel(arenaSize, arenaSize);
    initModel();

    view = new ArenaView(model, 400, 400, arenaSize);
    model.addListener(this);
    model.addListener(view);
```

```
controller = new SimulationController(model);
controller.start();

view.setVisible(true);
log("initialized.");
}

private void initModel(){

    // parametri: nome, posizione, raggio,
    //direzione iniziale, raggio RFID,
    //raggio ultrasuoni, colore
    RobotInfo ri = new RobotInfo("alfa",
    new P2d(5,5), 0.30, new V2d(0,1), 0.030, 2,
    new java.awt.Color(0,0,255));
    model.addRobot(ri);

    RobotInfo ri2 = new RobotInfo("beta",
    new P2d(2,2), 0.20, new V2d(1,0), 0.020, 1,
    new java.awt.Color(255,0,0));
    model.addRobot(ri2);

     ObstacleInfo wall1 = new ObstacleInfo("w1",
     new P2d(0,0),10,0.50);
     ObstacleInfo wall2 = new ObstacleInfo("w2",
     new P2d(0,9.50),10,0.50);
     ObstacleInfo wall3 = new ObstacleInfo("w3",
     new P2d(0,0),0.50,10);
     ObstacleInfo wall4 = new ObstacleInfo("w4",
     new P2d(9.50,0),0.50,10);
     ObstacleInfo block1 = new ObstacleInfo("b1",
     new P2d(2,7),3,1);
     ObstacleInfo block2 = new ObstacleInfo("b2",
     new P2d(7,3),1,2);

     model.addObstacle(wall1);
     model.addObstacle(wall2);
     model.addObstacle(wall3);
     model.addObstacle(wall4);
     model.addObstacle(block1);
     model.addObstacle(block2);

     double dx = 1.0;
     double dy = 1.0;
     for (int i = 0; i < 10; i++){
```

```
        for (int j = 0; j < 10; j++){
            RFIDInfo rfid = new RFIDInfo(
            "rfid"+i+j,new P2d(dx*i,dy*j));
            model.addRFID(rfid);
        }
    }

    TargetInfo target = new TargetInfo("target",
    new P2d(1,1));
    model.addTarget(target);
}

/**
 * Implementation of the agent's basic actions
 */
@Override
public boolean executeAction(String ag, Structure act){

 if (act.getFunctor().equals("move_forward")){
    double vel =
    ((NumberTerm)act.getTerm(0)).solve();
     model.setRobotMovingFwd(ag,vel);
  } else
   if (act.getFunctor().equals("move_backward")){
        double vel =
        ((NumberTerm)act.getTerm(0)).solve();
        model.setRobotMovingBackward(ag,vel);
  } else
  if (act.getFunctor().equals("stop_moving")){
        model.setRobotStopped(ag);
  } else
  if (act.getFunctor().equals("change_dir")){
         int angle =
         (int)((NumberTerm)act.getTerm(0)).solve();
          //log("ACTION change dir "+ag+" "+angle);
          double angle2 =
          ((double)angle)*Math.PI/180;
          model.changeDir(ag,angle2);
   }
    return true;
}

public void notifyStateUpdated(double dt){

  model.acquireReadLock();
```

```
Literal colliding =
    Literal.parseLiteral("touch_on");
long time =
    Math.round(model.getCurrentTime()*1000);
Literal timeLit =
   Literal.parseLiteral("time("+time+")");
Collection<RobotInfo> robots =
    model.getRobotsInfo();

for (RobotInfo robot: robots){

String name = robot.getId();

try {
  clearPercepts(name);

  if (robot.isMovingFwd()){
    addPercept(name, Literal.parseLiteral(
        "moving("+robot.getSpeed()+")"));
  }

  if (robot.isColliding()){
    addPercept(name,colliding);
  }

  P2d pos = robot.getDetectedPos();
  if (pos != null){
    addPercept(name,
    Literal.parseLiteral(
    "position("+pos.x+","+pos.y+")"));
  }

  double dist = robot.getDetectedObjectDistance();
  if (dist >= 0){
    addPercept(name,
    Literal.parseLiteral(
    "detected_surface("+dist+")"));
   }

  String target = robot.getDetectedTargetName();
  if (target != null){
     addPercept(name,
     Literal.parseLiteral(
     "detected_target("+target+")"));
  }
```

```
            addPercept(name, timeLit);

            } catch (Exception ex){
              ex.printStackTrace();
            }

            informAgsEnvironmentChanged(name);

            }
            model.releaseReadLock();
}
    public void log(String msg){
        System.out.println("[ARENA - ENV] "+msg);
    }

}
```

La classe `SimulationController` si è implementata come segue.

```
package env;

public class SimulationController extends Thread {

   private ArenaModel model;
   private SimulationState state;

   public SimulationController(ArenaModel model){
      this.model = model;
      state = new SimulationState();
    }

     public void run(){
     state.setStopped(false);

     double dt = 0.01;
     try {
        log("Waiting one second for agents' setup completion...");
        Thread.sleep(1000);
        log("Starting.");
        while (!state.isStopped()) {
           model.updateState(dt);
           Thread.sleep(10);
           //System.out.println("updated.");
         }
     } catch (Exception ex){
```

```
            ex.printStackTrace();
        }
    }

    private void log(String msg){
        System.out.println("[SIM CONTROLLER] "+msg);
    }
}
```

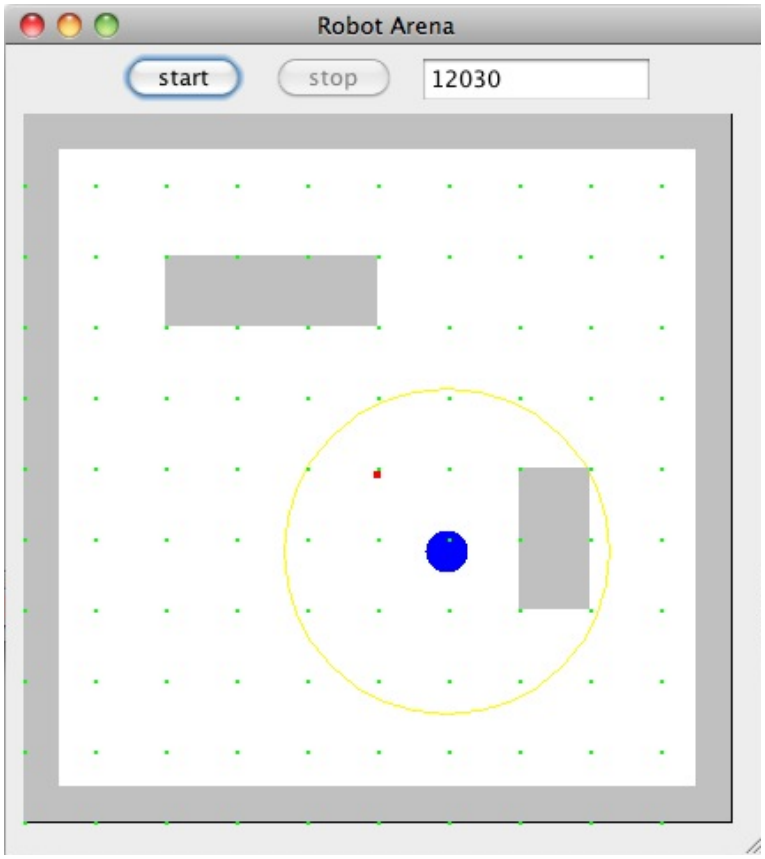

Figura 4.1: Interfaccia grafica realizzata in Java per la rappresentazione dell'arena dell'esempio applicativo.

Nella figura 4.1, in particolare, si nota che:

- il *robot* è rappresentato di forma circolare e di colore blu;

- il *target* è rappresentato dal puntino rosso;
- i *tag* RFID sono i puntini verdi.
- il cerchio giallo intorno al robot rappresenterebbe il *campo di visione* del robot nel caso in cui abbia molteplici sensori ad ultrasuoni posizionati sulla propria circonferenza, in questo esempio il robot è equipaggiato di un solo sensore ad ultrasuoni perciò la sua visione è unidirezionale lungo la direzione di marcia.

4.3.2 Robot

I robot sono agenti fisici che eseguono compiti agendo sull'ambiente. In questo esempio le attività distinguibili che il robot deve essere capace di svolgere sono tre, tra cui due di base, muoversi e districarsi tra gli ostacoli, mentre la terza è specifica poiché introduce la capacità di realizzare il goal, ovvero trovare il target. Per fare questo il robot è equipaggiato di sensori e di attuatori. Tali dispositivi sono stati scelti sempre in uno scenario Lego-Mindstorms-like, sia per motivi economici ma anche per avere un riferimento sui parametri fisici dei robot.

Il punto fondamentale che interessa sottolineare e chiave di lettura per la discussione successiva, non è spiegare il funzionamento dei sensori, aspetti più rilevanti per l'elettronica, ma riguarda l'informazione che restituisce in risposta il sensore e come possiamo esprimerla dal punto di vista informatico, in particolare in riferimento agli agenti. Il motivo, dunque, dell'inserimento dei principi di funzionamento dei sensori è con lo scopo di comprendere al meglio il tipo di valori in ingresso e in uscita ovvero una visione ai morsetti.

I sensori che si mettono in campo sono:

Sensore ad ultrasuoni: I sensori a ultrasuoni detti anche sonar, appartengono alla categoria dei rilevatori a distanza, che misurano appunto la distanza tra i robot e gli oggetti più vicini. I sensori ad ultrasuoni emettono onde direzionali, una parte delle quali è riflessa dagli oggetti e ritorna verso il sensore. Il ritardo e l'intensità del segnale di ritorno contiene forniscono informazioni sulla distanza degli oggetti più vicini. Il sensore ad ultrasuoni installati sul robot rileva gli ostacoli solamente se si trovano lungo la direzione di marcia. Nel momento in

cui l'ostacolo entra nel campo visivo del robot, quest'ultimo inizia un graduale rallentamento fino a quando non raggiunge una distanza di mezzo metro, dopodiché manterrà una velocità costante fino ad urtare l'ostacolo. La percezione generata quando l'ostacolo entra nel campo visivo del sensore ad ultrasuoni e che verrà inserita nella base delle credenze è `detected_surface(Dist)`. La percezione viene costantemente aggiornata man mano che il robot si avvicina all'ostacolo, e viene cancellata dalla base delle credenze quando l'ostacolo non è più rilevato dal sensore ad ultrasuoni.

Sensore di contatto: Il sensore di contatto è un dispositivo capace di rilevare il contatto con una superficie. Tale dispositivo è dotato di un pulsante che in base al proprio stato restituisce una informazione. Se il pulsante è premuto, l'informazione restituita è "sono in contatto con la parate", altrimenti se il pulsante è rilasciato (posizione di default), l'informazione restituita è "non sono in contatto con la parete". La percezione relativa al contatto fra il sensore e un ostacolo è stata definita `touch_on(SensorId)`. Tale percezione viene inserita nella base delle credenze nel momento in cui viene generata, permane finché il sensore resta a contatto con l'ostacolo. Nel momento in cui il contatto viene meno, la percezione viene eliminata dalla base delle credenze. Il robot è dato di un unico sensore di contatto che ricopre la sua intera circonferenza. Il parametro *SensorId* in questo semplice esempio non è significato, in quanto il robot possiede solo un sensore di contatto. Il parametro è stato inserito per rendere agevole una futura espansione del comportamento del robot.

Sensore di luce: Il sensore di luce è in grado di riconoscere variazioni di luminosità (scala di grigi). Nell'esempio la percezione relativa al sensore di luce è `detected_target(TargetId)`. Il parametro `TargetId` è stato inserito, così da poter eventualmente realizzare in futuro un comportamento per il robot finalizzato al ritrovamento di più target.

Tachimetro: Il tachimetro è uno strumento per misurare la velocità di marcia. Il tachimetro rileva una velocità solo se il robot sta andando avanti. Altrimenti, se il robot è fermo o sta facendo retromarcia, il tachimetro non rileva nulla. La percezione relativa è `moving(Speed)`.

Lettore RFID: il lettore RFID emette un campo elettromagnetico/elettrico che tramite il processo della induzione genera nell'antenna del tag una corrente che alimenta il chip. Il chip così alimentato comunica tutte le sue informazioni che vengono irradiate tramite l'antenna verso il lettore. La tecnologia RFID potrebbe essere utilizzata per la localizzazione del robot. In questo semplice esempio il robot si muove in maniera casuale, perciò non necessita di informazioni perciò che riguardano la localizzazione. Di conseguenza, anche se il robot passa vicino ad un tag RFID non si crea alcuna percezione.

Le percezioni del robot sono:

- `detected_surface(Dist)`
- `touch_on(SensorId)`
- `detected_target(TargetId)`
- `moving(Speed)`
- `time(T)`

L'unica percezione da commentare è l'ultima in elenco, ovvero `time(T)`. Tale percetto non è stato discusso finora perché riguarda esclusivamente la simulazione e perciò non ha sensori associati. Nel software, in generale, il tempo si differenzia in tempo logico e in tempo fisico. Il tempo logico viene scandito dal controllore della simulazione, e gli agenti lo percepiscono attraverso la belief `time(T)`, dove il parametro `T` è espresso in millisecondi. Il tempo fisico, invece, fa riferimento al tempo scandito dal processore. Quindi poiché il tempo fisico viene stabilito in modo indipendente dall'agente, la creazione di un tempo simulato, logico, permette di regolare il passo con cui avviene la simulazione.

Si stabiliscono gli attuatori presenti sul robot:

Due servomotori nelle due ruote laterali Da soli sensori ed attuatori non bastano: un robot completo necessita anche di fonti di energia per azionare gli attuatori. Le due ruote laterali permettono al robot di spostarsi avanti, indietro e di girare.

Una ruota posteriore sferica La sfericità della ruota permette di seguire i movimenti del robot senza la necessità di un motore. Il suo ruolo è semplicemente di una ruota d'appoggio, ovvero fornisce maggiore stabilità al robot.

Dati gli attuatori di cui è equipaggiato il robot, esso può svolgere determinate azioni:

- `move_forward(Speed)`
- `move_backward(Speed)`
- `stop_moving`
- `change_dir(Angle)`

Si noti che tutte queste azioni riguardano il movimento. In particolare, le prime due risultano molto simili, in quanto rispettivamente la prima permette di andare avanti, la seconda di andare indietro, ma entrambe ad una determinata velocità indicata dal parametro `Speed`. Queste due azioni sono state progettate con l'intento di essere fedeli alla realtà dei LegoMindstorms, ovvero una volta che viene dato il comando di avanzare (retrocedere), il robot continuerà a proseguire finché non gli viene detto espressamente detto di fermarsi. L'idea è quindi di fornire al robot la sola velocità di marcia e non il valore di quanto deve avanzare (retrocedere). In particolare, fare retromarcia per il robot significa procedere all'indietro. Siccome nell'esempio il robot ha i sensori ad ultrasuoni solo sulla parte anteriore del suo corpo, muoversi avanti oppure indietro risulta differente in quanto se ha un ostacolo dietro di sé non lo rileva.

A questo punto sono stati descritti tutti gli elementi necessari per definire il comportamento del robot attraverso il linguaggio di programmazione ad agenti.

Il codice **Jason** del robot, chiamato *explorer1*, progettato per essere inserito nell'arena dell'esempio è il seguente:

```
!explore.

+!explore
 <-     Angle = math.random(120) - 60;
        change_dir(Angle);
```

```
        move_forward(1);
        Dt = 750+math.random(2000);
        !wait_for(Dt);
        stop_moving;
        !!explore.

+touch_on
<-      .suspend(explore);
        .println("suspended.");
        stop_moving;
        move_backward(1);
        !wait_for(500);
        change_dir(45);
        .resume(explore);
        .println("resumed.").

+detected_surface(Dist) : moving(Speed) & Dist > 0.5
<-      move_forward(Dist*0.9).

+detected_target(Target)
<-      .println("Ho individuato un target");
        stop_moving;
        .drop_all_desires.

+!wait_for(DT) : time(T1)
<-      T2 = T1+math.round(DT);
        !wait_until(T2).

-!wait_for(DT)
 <-     .wait(10);
        !wait_for(DT).

+!wait_until(T2) : time(T) & T > T2.

-!wait_until(T2)
 <- .wait(10);
    !wait_until(T2).
```

Il goal iniziale del robot è `!explore`. Nel momento in cui il robot inizia la sua esecuzione l'evento `+!explore` viene generato. Questo piano semplicemente permette al robot di navigare in modo non prevedibile la stanza.

La funzione matematica `math.random(x)` restituisce in modo casuale un numero reale compreso fra 0 e x. Di conseguenza, il robot si gira di un angolo compreso tra -60° e +60°, inizia ad avanzare per un certo periodo di tempo, poi si ferma ed infine richiama `explore` stessa, ovvero fa generare di nuovo l'evento `+!explore`.

Nel caso in cui il robot tocca una superficie e, di conseguenza, viene generato l'evento `+touch_on`, il piano `explore` viene sospeso. Questo permette al robot di fare per un determinato periodo di tempo retromarcia, per poi riprendere il piano sospeso.

Nel momento in cui il robot rileva una superficie e le condizioni del contesto sono verificate, esso inizia a diminuire la sua velocità di marcia. Nel caso in cui il robot rileva una superficie ma le condizioni del contesto non sono verificate, il robot semplicemente con gestisce l'evento.

I piani `wait_for(DT)` e `wait_until(T2)` gestiscono il tempo logico, facendosi aiutare dal percetto `time(T1)`.

Il problema è introdurre il concetto di attendere una certa quantità di tempo. Per il tempo fisico Jason mette a disposizione l'azione interna `.wait(T1)` ma per il tempo logico non è stata predefinita nessuna azione. Per questo motivo sono stati creati i piani `wait_for(DT)` e `wait_until(T2)` con i relativi piani di riparazione.

4.4 Aspetti chiave del modello di programmazione

4.4.1 "Un robot ovunque": generalità del modello di programmazione

Un computer in ogni casa sembrava un diktat irraggiungibile solo pochi decenni fa e adesso essi risultano essenziali per svolgere numerose attività in vari settori di lavoro e di ricerca. Essi sono diventati uno strumento indispensabile, vengono utilizzati ormai dalla maggior parte della popolazione per una quantità di tempo considerevole durante le loro giornate. In modo analogo è possibile pensare che nel prossimo futuro anche i robot entrino a

far parte della vita quotidiana delle persone, svolgendo in modo autonomo compiti come - ad esempio - la pulizia di ambienti, lo spostamento di oggetti pesanti, etc...

Si può pensare perciò che i futuri ingegneri informatici quotidianamente non programmeranno più applicazioni, bensì comportamenti per i robot ovvero piani per gli agenti.

Un agente è semplicemente qualcosa che agisce, fa qualcosa. Tuttavia si suppone che gli agenti abbiano caratteristiche particolari, che li distinguono dalle semplici applicazioni: ad esempio potrebbero operare con controllo autonomo, essere in grado di percepire l'ambiente, adattarsi al cambiamento ed essere capaci di scambiarsi gli obiettivi a vicenda. Un agente agisce in modo migliore in modo da ottenere il miglior risultato, o in condizione di incertezza, il miglior risultato atteso.

Gli agenti, in linea teorica, offrono delle potenzialità che nessun linguaggio di programmazione ha mai avuto prima. Gli ingegneri informatici però non possono fermarsi solo alla teoria, ma devono riuscire a realizzare concretamente i concetti, altrimenti sarebbe come dire, usando una espressione popolare, "tutto fumo e niente arrosto".

Modularità e flessibilità

I programmi software possono avere una natura e complessità profondamente diversa a seconda del problema. Problemi di pura natura algoritmica, in cui il programma, dato un certo input produce un certo output, e termina. Questa programmazione viene definita "in the small". Altri problemi richiedono, invece, lo sviluppo di vere e proprie applicazioni e sistemi (software) che possono includere varie forme di interazione. Questa programmazione, in contrapposizione con la precedente, viene chiamata *"in the large"*. Dal punto di vista dell'ingegneria, l'obiettivo è di riuscire a fare, in prospettiva futura, un robot di una certa complessità, ovvero come sistema in the large.

Una riflessione, a tale proposito, riguarda la possibilità di servirsi dei paradigmi di programmazione esistenti. In particolare, i concetti fondamentali della programmazione ad oggetti di: modularità, ereditarietà, estensione e riuso.

Per le considerazioni seguenti viene preso come riferimento Jason, per il linguaggio ad agenti, e Java, per quello ad oggetti.

Dal punto di vista del linguaggio ad agenti, la *modularità* è vista come la possibilità di strutturare il comportamento del robot, sia per quanto riguarda le caratteristiche di proattività, che quelle di reattività.

Il concetto di comportamento proattivo permette di strutturare attività complessa in goal e sottogoal, pensando anche ad una gerarchia di piani con sottopiani associati ai sottogoal. Jason, inoltre, fornisce la possibilità di creare piani eseguiti a fronte di fallimenti di piani stessi (piani di riparazione). Nel caso in cui si verifica una situazione che porta al fallimento di una azione, causa il fallimento del piano dove quest'ultima è contenuta. In uno scenario ipotetico dove i piani non interagiscono fra loro, il fallimento di una azione provoca il fallimento di un solo piano non influenzando l'esecuzione degli altri. Se in futuro all'agente, si richiederà di nuovo di eseguire un piano precedentemente fallito, questo non determina che fallirà di nuovo ma, ad esempio per variazioni dell'ambiente, potrebbe essere portato a termine correttamente. In Java un evento anomalo/problematico che non permette la continuazione del metodo o del contesto di esecuzione in cui si verifica è indicato con il termine eccezione. Nel caso di eccezioni, non è possibile continuare la computazione corrente dal momento che non ci sono informazioni sufficienti per fare fronte al problema nel contesto corrente. Java fornisce costrutti per gestire le eccezioni: ovvero l'occorrenza di un'eccezione come evento anomalo a runtime che non provoca la terminazione del programma in esecuzione, ma può essere opportunamente gestita all'interno dello stesso. La cattura dell'eccezione concerne la specifica delle azioni da fare lato utilizzatore del metodo per gestire le eccezioni generate da oggetti con cui ha interagito. La generazione di una eccezione e relativa cattura comportano una interruzione e trasferimento del flusso di controllo, dal punto in cui viene lanciata l'eccezione al punto in cui vene gestita. Si nota, dunque, che un supporto di base per la gestione delle anomalie è presente in entrambi i linguaggi. La differenza sostanziale tra i due linguaggi, che si nota anche affrontando questo aspetto, è che Java ha un flusso di controllo continuo (sequenziale) mentre Jason non possiede questo concetto in quanto il comportamento dell'agente dipende dal modo con cui esso si imbatte nei componenti dell'ambiente. In Jason, un aiuto in fase di debug, che sottolinea questo approccio, è il Jason Mind Inspector, la cui interfaccia grafica è mostrata in figura 1.2 nell'introduzione.

Il concetto di comportamento reattivo permette di sfruttare una forma di modularizzazione avvalendosi del contesto di un piano. Questo compo-

nente, presente in Jason, permette di specificare condizioni che permettono di mettere in esecuzione piani diversi a fronte di una stessa percezione dell'ambiente, a seconda del task per cui l'agente sta operando. Il contesto che per gli agenti è una pre-condizione per il piano, in quanto se soddisfatto rende il piano applicabile, in chiave generale può essere visto sia come la struttura di controllo alternativa (if-then), mentre in chiave object oriented può essere accostato ad una sorta di polimorfismo. Per quanto riguarda la corrispondenza con l'if, le considerazioni risultano molto semplici. Parafrasando il significato dell'if-then con "se vale la condizione C, esegui l'istruzione (blocco) I", l'analogo per il contesto sarà "se vale il contesto C, esegui il piano P". La corrispondenza con il polimorfismo è per certi versi meno naturale. In Java, i metodi che vengono ridefiniti in una sottoclasse sono detti polimorfi, in quanto lo stesso metodo di comporta diversamente a seconda del tipo di oggetto su cui è invoca. L'idea, quindi lato Jason, è dire lo stesso evento provoca l'esecuzione di un piano diverso a seconda del task che l'agente sta svolgendo, ossia del contesto di esecuzione.

Se analizziamo il concetto di modularità da un punto di vista più vicino alla robotica un collegamento che si estrae è con il controllo basato sul comportamento. In tale controllo, i comportamenti vengono usati come moduli per il controllo.

Il linguaggio ad oggetti prevede un meccanismo molto importante, l'*ereditarietà*, che permette di derivare nuove classi a partire da classi già definite. L'ereditarietà permette di aggiungere membri in una classe, e di modificare il comportamento dei metodi, in modo da adattarli alla nuova struttura della classe. Sintatticamente, una classe può essere definita come derivata da un'altra classe esistente. In Java la classe derivata, o sottoclasse, eredita tutti i metodi e gli attributi della classe "genitrice", e può aggiungere membri alla classe, sia attributi che metodi, e/o ridefinire il codice di alcuni metodi. L'ereditarietà può essere usata come meccanismo per gestire l'evoluzione ed il *riuso* del software: il codice disponibile definisce delle classi, se sono necessarie modifiche, vengono definite sottoclassi che adattano la classe esistente alle nuove esigenze. Spostandosi al linguaggio di programmazione ad agenti, questo significherebbe possedere le possibilità di creare inizialmente un robot in grado di raggiungere certi obiettivi, per poi, in un momento successivo, avere un approccio per estendere le sue capacità correnti tramite l'inserimento di nuovi piani. A tale scopo, il sistema reattivo può essere strutturato tramite l'architettura di sussunzione. Si ri-

corda che un completo sistema reattivo può essere determinato per qualsiasi robot, compito ed ambiente che possono essere stabiliti in anticipo. Tale architettura permette di costruire sistemi in modo incrementale utilizzando per i nuovi elementi aggiuntivi, parti già esistenti. Tale architettura è molto diffusa per la sua semplicità e robustezza, ma se l'obiettivo è realizzare una attività complessa il sistema diventa facilmente grande, in quanto può richiedere un numero enorme di regole. Bisogna inoltre sottolineare che l'aggiunta di nuovi piani, richiede anche una gestione delle possibili interferenze fra i vari piani che si potrebbero creare.

Queste considerazioni vertono sul fatto di pensare a quale sia il supporto più adeguato per un approccio agent oriented per l'ingegneria del software, partendo dalle basi dell'object oriented. Questo non significa che è obbligatorio mappare i paradigmi del linguaggio ad oggetti in quello ad agenti, ma vuole essere semplicemente un confronto così da stabilire in modo chiaro similitudini e differenze tra i due, oltre a vantaggi e svantaggi di entrambi.

Il messaggio che si vuole trasmettere in questo paragrafo è che il linguaggio di programmazione ad agenti Jason è estremamente interessante come approccio di alto livello per programmare robot, ma ci sono diversi aspetti che necessitano di lavoro ulteriore in ambito di ricerca, in quanto hanno il bisogno di essere studiati e approfonditi con maggiore dettaglio.

Capitolo 5

Caso di studio

5.1 Comportamento del singolo robot per lo svolgimento di una attività complessa

A partire dal semplice esempio descritto nel capitolo precedente, si analizza come applicare le scelte di architettura discusse su un robot il cui compito è svolgere una attività complessa.

Ora si espongono nel dettaglio le caratteristiche del caso di studio preso in considerazione, partendo da una sua descrizione o in linguaggio tecnico dalle specifiche di progetto.

Si considera un *ambiente* rappresentato come un'arena delimitata da quattro ostacoli, essi possono essere immaginati come muri dal momento che creano una sorta di stanza di forma quadrata. All'interno dell'arena possono essere collocati altri ostacoli, i quali possono assumere esclusivamente una posizione orizzontale oppure verticale. Nell'arena è situato un *robot*, in grado di muoversi in qualsiasi direzione (orizzontale, verticale oppure obliqua) all'interno di essa.
Il robot durante la sua marcia può riconoscere vari tipi di oggetti presenti nella stanza. Tali oggetti sono gli ostacoli, la base di carica e la polvere. Gli ostacoli impediscono la marcia del robot in quella direzione. La base di carica e la polvere sono gli oggetti caratteristici dell'ambiente. Il robot possiede uno stato interno che rappresenta il suo livello attuale di *carica della batteria* e conosce la posizione della base di carica. Nel momento in cui il robot si sente scarico, torna alla base di carica. Per poi riprendere la

pulizia solo dopo che si è ricaricato. Lo sporco, in particolare la polvere, è visto come un oggetto dotato di un valore. Il robot riconosce se livello di polvere sul pavimento da pulito ad estremamente sporco. Il comportamento di robot è di navigare l'intera stanza in modo non lasciare zone non perlustrate. L'obiettivo è di pulire l'intera stanza dalla polvere.

Il caso di studio è stato modellato attraverso una simulazione, ma si è fatto riferimento ad uno scenario Lego-Mindstorms-like. Tale scelta ha influenza sia la progettazione dell'arena, sia del robot.

5.1.1 Ambiente

Per modellare l'ambiente del caso di studio, si è partiti da quello costruito per l'esempio applicativo.

La figura 5.1 rappresenta l'organizzazione delle cartelle in cui sono contenuti i sorgenti Java dell'ambiente. Nella sottocartella `common` della cartella `env` sono presenti le classi tratte dall'esempio applicativo. Alcune classi sono state modificate e poi rinominate aggiungendo il prefisso `Basic` davanti al rispettivo nome, quelle con lo stesso nome invece sono rimaste uguali. L'idea è appunto che tramite l'utilizzo di solo questa cartella, si riescano a svolgere le azioni di base. La cartella `room_env` contiene i sorgenti che specializzano quelli contenuti in `common` per permettere al robot di compiere l'attività di pulire la stanza. La terza cartella, ovvero `action_lib`, contiene i sorgenti che aiutano il robot nei calcoli per l'orientamento. In realtà, è presente un'altra cartella che in figura non è stata rappresentata denominata `team_env`. Tale cartella di riferisce ad un altro caso di studio fatto sempre a partire dal precedente esempio. Nel caso sia di interesse per il lettore, quest'altro caso di studio è stato realizzato da Seri Laura nella tesi dal titolo *"Programmazione di agenti situati in ambienti fisici mediante la piattaforma Jason"* discussa nel medesimo anno accademico e sessione di questa tesi. Lo scopo di quest'altro caso di studio è di approfondire gli aspetti di collaborazione fra più agenti.

Nelle figure 5.2 , 5.3 , 5.4 catturate in diversi momenti durante il movimento del robot, si nota che:

- il *robot* è rappresentato di forma e di colore blu. Nel caso in cui il robot stia pulendo sullo stesso comparirà una lettera C di colore bianco, se il robot invece è completamente scarico la lettera visibile sarà una D di

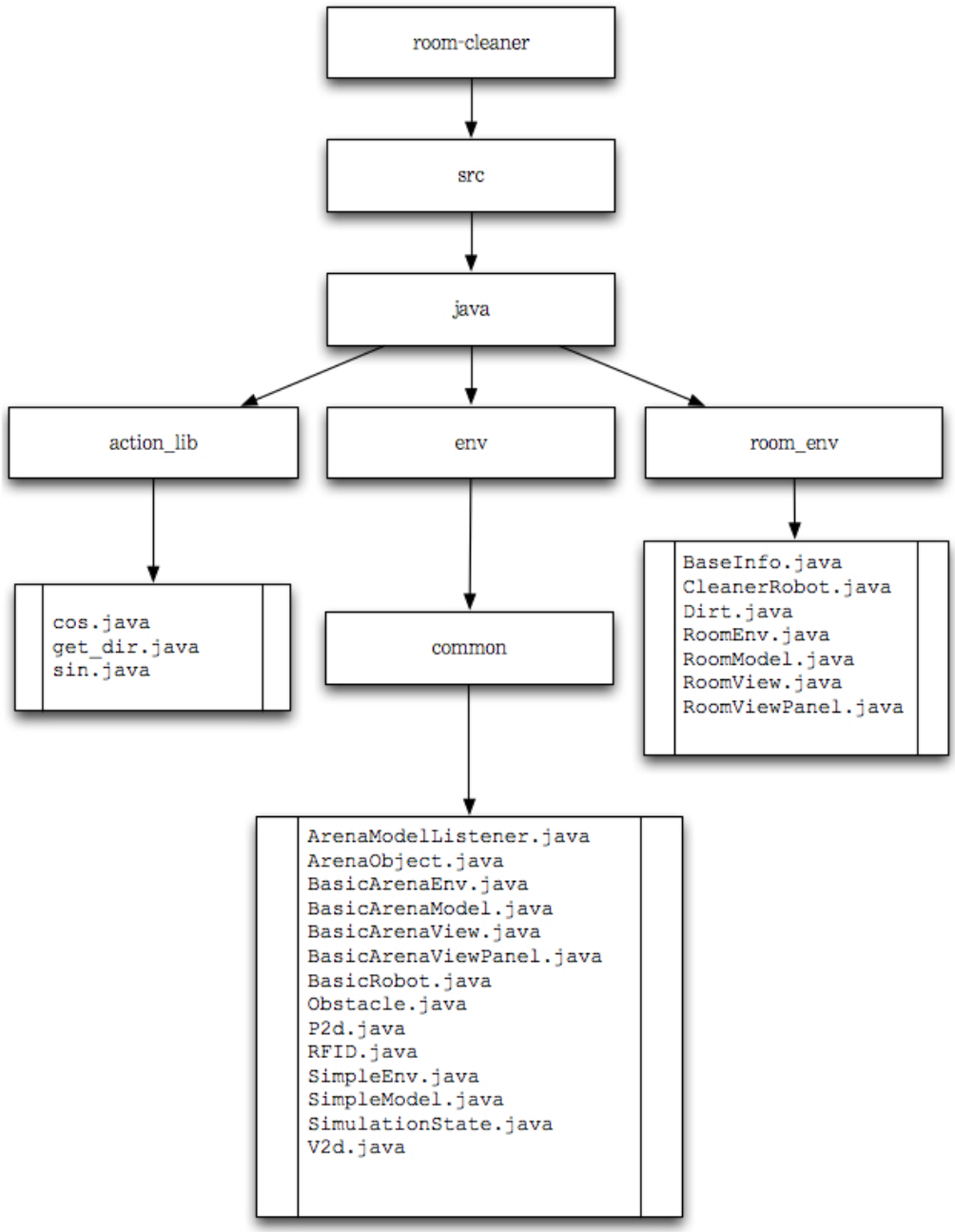

Figura 5.1: Organizzazione dei file sorgenti dell'ambiente realizzato in Java del caso di studio.

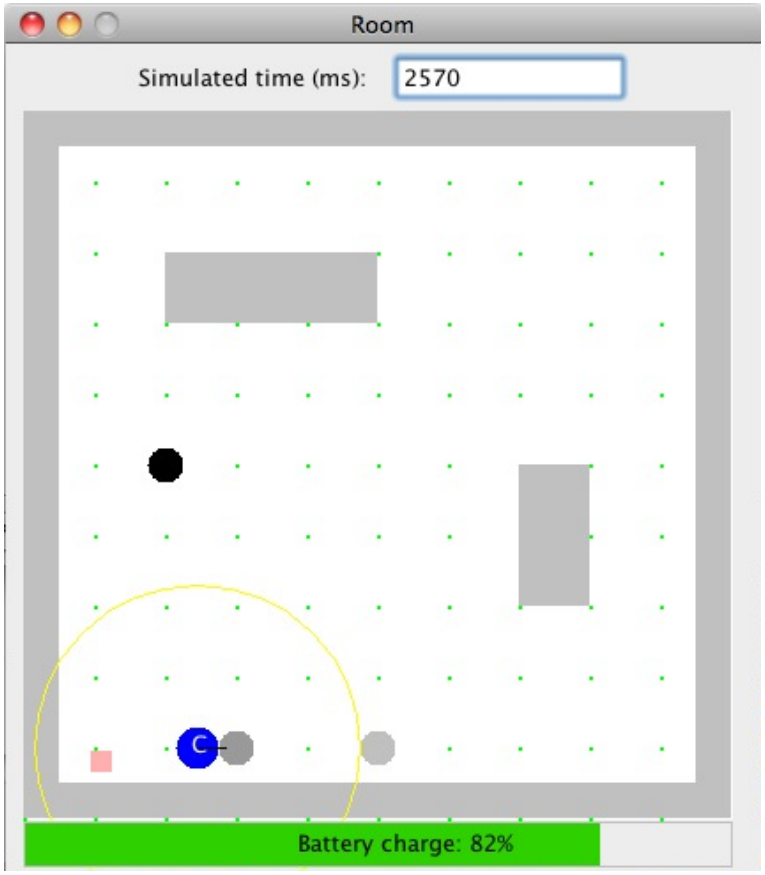

Figura 5.2: Interfaccia grafica realizzata in Java per la rappresentazione dell'arena del caso di studio catturata in un momento in cui il robot sta pulendo la stanza.

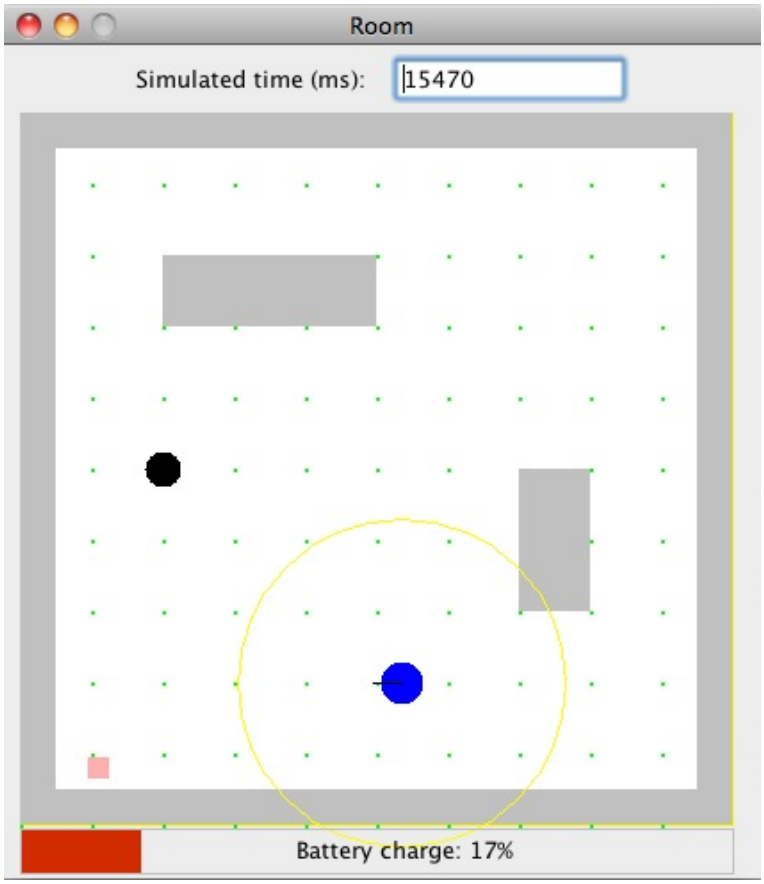

Figura 5.3: Interfaccia grafica realizzata in Java per la rappresentazione dell'arena del caso di studio catturata in un momento in cui il robot inizia a percepire di avere un livello di carica della batteria basso.

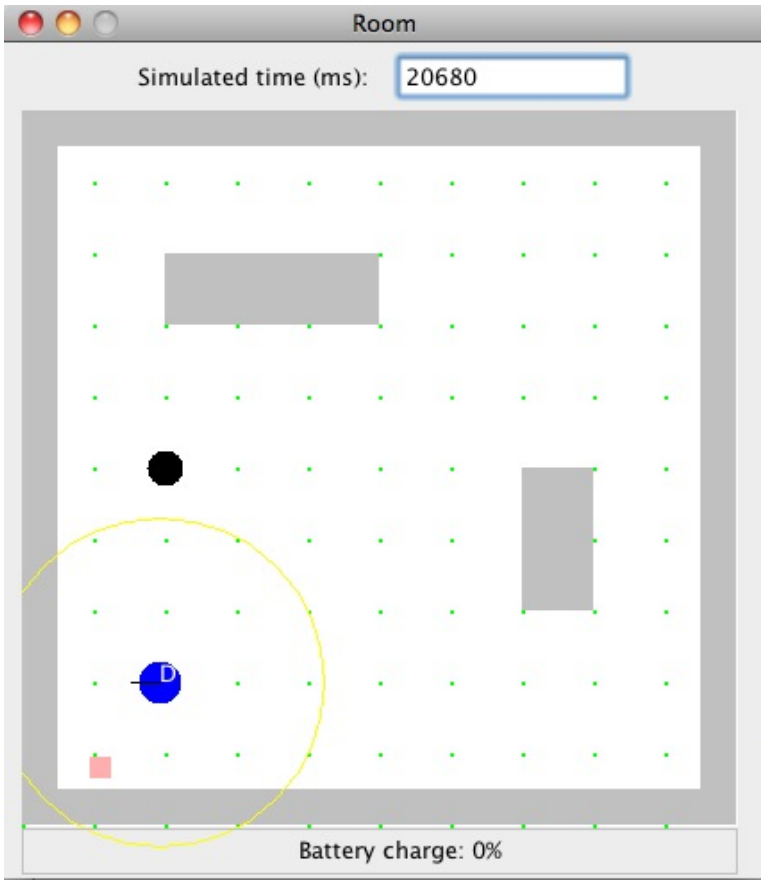

Figura 5.4: Interfaccia grafica realizzata in Java per la rappresentazione dell'arena del caso di studio catturata in un momento in cui il robot è totalmente scarico, in quanto il livello di batteria che possedeva è stato sufficiente per tornare alla base di carica.

colore bianco. Sul robot è stata applicata una lineetta nera che indica la parte anteriore del robot, in particolare dove è collocato il sensore ad ultrasuoni. la direzione in cui si sta muovendo il robot.

- i *tag* RFID sono i puntini verdi;
- lo *sporco* si identifica da puntini con scale di grigio diverse a seconda del livello di sporco. Maggiore il punto sul pavimento è sporco, più scuro sarà il suo colore fino ad arrivare ad essere nero.
- il livello di *carica della batteria* è reso visibile una barra di progressione, in particolare si è inserita l'espressione "Battery charge: ≪valore≫%". In fase di cleaning il robot consuma una maggiore quantità di carica rispetto allo svolgimento di tutte le altre azioni, tale aspetto è visibile in quanto il livello di batteria cala più velocemente.

Note sulla barra di progressione

La percentuale indicata sulla barra di progressione è calcolata in base allo stato interno della carica del robot. Oltre alla graduale diminuzione di valore della carica, la barra cambia anche di colore. Inizialmente quando la carica del robot è al 100% la barra viene rappresentata di colore verde, man mano che il livello diminuisce il colore passa da verde a rosso. Il colore della barra è espresso dalla tripla di valori R, G, B (Red Green Blue). Quando la carica è al 100% la tripla è (0,255,0), poi in base alla diminuzione della carica del robot si aumenta il valore del red e si diminuisce quello del green, per arrivare a (255,0,0).

Dettagli sulla realizzazione dell'ambiente in Java

Per maggiore completezza, si inseriscono delle parti del codice Java specifiche del caso di studio.

Il metodo `updateRobotState` della classe `RoomModel` si è implementato come segue.

```
package room_env;

import java.util.*;
import java.util.concurrent.locks.*;
import env.common.*;
```

```
public class RoomModel extends BasicArenaModel {

   \\  .......

   protected void updateRobotState(BasicRobot ri, double dt){
      super.updateRobotState(ri, dt);
      CleanerRobot cleaner = (CleanerRobot)ri;

      boolean found = false;
      Iterator<Dirt> it = dirts.listIterator();
      while (it.hasNext()){
          Dirt oi = it.next();
         if (oi.isDetectable(cleaner.getPos(),
            cleaner.getRadius())){
            if (cleaner.isCleaning()){
               oi.clean(dt);
             }
             if (oi.getLevel() <= 0){
                it.remove();
             } else {
                cleaner.setDetectedDirt(oi.getLevel());
                found = true;
                break;
                }

            }
       }

       if (!found){
          cleaner.setDetectedDirt(0);
       }
   }
}
```

La classe `RoomEnv` si è implementata come segue.

```
package room_env;
import jason.asSyntax.*;
import env.common.*;

public class RoomEnv extends BasicArenaEnv {


   protected BasicArenaModel makeModel(){

       RoomModel model =  new RoomModel(10, 10);
```

```
// parametri: nome, posizione iniziale, raggio,
//direzione iniziale, raggio RFID, raggio ultrasuoni,
//colore
CleanerRobot ri = new CleanerRobot("alfa",
  new P2d(1,1), 0.30, new V2d(1,0), 0.30, 2,
  new java.awt.Color(0,0,255));
model.addRobot(ri);

//RobotInfo ri2 = new RobotInfo("beta", new P2d(2,2),
//0.20, new V2d(1,0), 0.020, 1,
//new java.awt.Color(255,0,0));
//model.addRobot(ri2);

Obstacle wall1 = new Obstacle("w1",
   new P2d(0,0),10,0.50);
Obstacle wall2 = new Obstacle("w2",
   new P2d(0,9.50),10,0.50);
Obstacle wall3 = new Obstacle("w3",
   new P2d(0,0),0.50,10);
Obstacle wall4 = new Obstacle("w4",
   new P2d(9.50,0),0.50,10);
Obstacle block1 = new Obstacle("b1",
   new P2d(2,7),3,1);
Obstacle block2 =
   new Obstacle("b2", new P2d(7,3),1,2);

model.addObstacle(wall1);
model.addObstacle(wall2);
model.addObstacle(wall3);
model.addObstacle(wall4);
model.addObstacle(block1);
model.addObstacle(block2);

double dx = 1.0;
double dy = 1.0;
for (int i = 0; i < 10; i++){
   for (int j = 0; j < 10; j++){
     RFID rfid = new RFID("rfid"+i+j,
        new P2d(dx*i,dy*j));
     model.addRFID(rfid);
      }
}

BaseInfo base =
   new BaseInfo("base", new P2d(1,1));
```

```
        model.addBase(base);

        Dirt dirt1 = new Dirt(new P2d(2,1), 0.25, 1);
        Dirt dirt2 = new Dirt(new P2d(2,5), 0.25, 1);
        Dirt dirt3 = new Dirt(new P2d(3,1), 0.25, 0.5);
        Dirt dirt4 = new Dirt(new P2d(5,1), 0.25, 0.25);

         model.addDirt(dirt1);
         model.addDirt(dirt2);
         model.addDirt(dirt3);
         model.addDirt(dirt4);

        return model;
    }

    protected BasicArenaView
       makeView(BasicArenaModel model){
        return new RoomView((RoomModel)model, 400, 400);
    }

    /**
     * Implementation of the agent's basic actions
    */
    @Override
    public boolean executeAction(String ag, Structure act) {
       RoomModel rmodel = (RoomModel)model;
        if (act.getFunctor().equals("start_cleaning")){
            rmodel.setRobotCleaning(ag);
            return true;
        } else if (act.getFunctor().equals("stop_cleaning")){
         rmodel.stopCleaning(ag);
          return true;
        } else {
         return super.executeAction(ag, act);
        }
     }

     protected void
        generateRobotPercepts(BasicRobot robot, double time){
        super.generateRobotPercepts(robot, time);
        String name = robot.getId();
        CleanerRobot cleaner = (CleanerRobot)robot;

        if (cleaner.detectedDirtLevel() > 0){
           addPercept(name, Literal.parseLiteral("dirt"));
```

```
        }
      }

    public void log(String msg){
       System.out.println("[STANZA-ENV] "+msg);
     }
}
```

5.1.2 Robot

La prima caratteristica da sottolineare è che il robot è completamente *autonomo*. Il robot non è dotato di una mappa pre-impostata o di un programma che gli fornisce nel dettaglio la sequenza delle azioni da svolgere. Rispettando il principio di località, il robot scopre l'ambiente un po' alla volta durante la sua marcia. Il robot dinamicamente in base alle percezioni e al contesto in cui si trova in ogni momento sceglie l'azione da eseguire.

Il robot per realizzare il suo obiettivo, a fronte delle considerazioni precedenti, è equipaggiato di sensori e di attuatori.

I sensori che si mettono in campo sono:

- sensore ad ultrasuoni
- sensore di contatto
- sensore di luce
- lettore RFID
- bussola
- indicatore di carica

Alcuni di questi sensori sono già stati illustrati nel capitolo precedente, in particolare nel paragrafo dell'esempio applicativo, perciò di seguito verranno solamente descritti i restanti o precisazioni nell'utilizzo del determinato sensore nel caso di studio.

Sensore di luce: Il sensore di luce è in grado di riconoscere variazioni di luminosità (scala di grigi). Nel caso di studio la percezione relativa al sensore di luce è `dirt`. Tale percezione possiede un livello da 0 a 1, 0 significa che il pavimento è pulito e quindi la percezione viene rimossa,

1 significa che il pavimento è estremamente sporco. Nel momento in cui il robot inizia a pulire il pavimento, lo percepisce sempre meno sporco e continua la sua marcia solo dopo che sente che quel punto è pulito.

Lettore RFID: Nel caso di studio la percezione relativa alla lettura di un tag RFID è `pos(X,Y)`, dove `X` e `Y` sono relative alle coordinate sulla griglia dei tag RFID, in seguito verrà discusso con maggiore dettaglio questo concetto

Bussola: la bussola restituisce un valore da 0° a 359°, ovvero l'angolo rispetto al Nord. Nel caso di studio la percezione relativa alla bussola è `compass(Angle)`

Indicatore di carica: l'indicatore di carica forniscono un valore circa lo stato di consumo della batteria. Nel caso di studio la percezione relativa all'indicatore di carica è `battery_charge(B)`. Il parametro `B` è un valore intero da 0 a 100. Il valore 0 significa che il robot è completamente scarico, mentre il valore 100 significa che il robot è completamente carico. Il livello della batteria cala a seguito dell'esecuzione di azioni del robot.

Nel caso di studio l'utilizzo della tecnologia RFID e della bussola è estremamente collegato. Da soli i tag e i lettori RFID e la bussola non bastano al robot per potersi orientare, bisogna introdurre il concetto di sistema di riferimento. Per prima cosa nella stanza si deve rilevare la direzione del Nord. Noto il Nord si ricava, di conseguenza, la direzione dell'Est. La direzione dell'Est e del Nord si pensi che traccino rispettivamente l'asse delle ascisse (x) e quello delle ordinate (y) in modo da creare un piano cartesiano che si assume sia il sistema di riferimento assoluto. I tag RFID vengono disposti a griglia sul tale piano con intervalli di spazio uguali fra gli elementi dispostivi in modo orizzontale e verticale (ovvero lungo la coordinata x uguale e coordinate y diverse) e anche fra loro (ovvero lungo coordinata y uguale e coordinate x diverse). In ciascun tag RFID si scrivono le coordinate del punto rispetto al piano cartesiano. Si sottolinea l'esigenza di vincolare la disposizione degli RFID con il Nord in quanto, altrimenti, per il robot l'informazione della bussola sarebbe non significativa. Il robot non ha conoscenza del sistema di riferimento assoluto, per questo motivo sia le coordinate che ha percepito riguardanti la sua posizione, sia le coordinate

che possiede nella base delle credenze risultano essere informazioni che da sole non hanno nessun valore. In tali circostanze, anche se il robot fosse equipaggiato della bussola, ancora le coordinate possedute gli risulterebbero inespressive. L'ottica è che la bussola indica al robot dove si trova il Nord, ma esso, in base alle sue conoscenze, non sa come mettere in relazione tale punto cardinale con le coordinate a disposizione. In altre parole, il robot conosce che lui si trova in posizione (5,6), dove si trova il Nord e che deve raggiungere la coordinata (1,1). Ma per andare in (1,1) il robot che direzione deve prendere, dato che non conosce dove si trova? Il robot per raggiungere la coordinata desiderata, si deve muovere verso Nord, Sud, Est, Ovest o con un certo angolo rispetto a uno di questi punti cardinali? Di conseguenza, il robot per orientarsi ha bisogno della creazione di un sistema di riferimento assoluto, di utilizzare la tecnologia RFID strutturata come una griglia orientata, ad esempio, secondo l'Est (asse x) e il Nord (asse y), di una bussola e di saper utilizzare le informazioni restituite da questi strumenti nel modo che segue. Il robot nel momento in cui rileva un tag RFID percepisce la posizione in cui si trova, ricava la direzione in cui sta andando attraverso la bussola, dalla base delle credenze ricava le coordinate del punto in cui si trova la base di carica e così ha tutti gli elementi per calcolare la direzione che deve prendere per muoversi (orientamento) verso la base di carica. Di seguito si mostrano dei piani, in codice **Jason**, che premettono al robot di orientarsi nella stanza.

```
!do_reach.

+!do_test
  <-  action_lib.sin(0.5,R);
      .println(R);
      !discover_position(X,Y);
      .println("located in ",X," ",Y).

+!do_test2
  <-  !align_to(1,4).

+!do_reach
  <-  !reach(8,8).

+!discover_position(X,Y) : position(X,Y)
```

```
  <-  stop_moving.

+!discover_position(X,Y) : not position(X,Y)
        <-      !align(90);
      +searching_pos;
                !search_pos;
      -searching_pos;
      -location_found(X,Y).

+!search_pos : not position(X,Y)
  <-  move_forward(2);
      !wait_for(500);
      stop_moving;
      change_dir(90);
      move_forward(2);
      !wait_for(20);
      stop_moving;
      change_dir(90);
      move_forward(2);
      !wait_for(500);
      stop_moving;
      change_dir(270);
      move_forward(2);
      !wait_for(20);
      stop_moving;
      change_dir(270);
      !search_pos.

+!search_pos : position(X,Y)
  <-  +location_found(X,Y).

@found_pos_plan [atomically]
+position(X,Y) : searching_pos
  <-  stop_moving;
      +location_found(X,Y);
      .succeed_goal(search_pos).
```

```
+!align(Angle) : compass_value(Angle).
+!align(Angle) : not compass_value(Angle)
  <- change_dir(1);
     .wait(10);
     !align(Angle).
-!align(Angle) <-
     .wait(10);
     !align(Angle).

+!align_to(TX,TY)
  <- ?position(X,Y);
     action_lib.get_dir(TX-X,TY-Y,A1);
     ?compass_value(A0);
     change_dir(A1-A0).

+?compass_value(A)
  <- .wait(10);
     ?compass_value(A).

+?position(X,Y)
  <- .wait(10);
     ?position(X,Y).

+!reach(Tx,Ty)
  <- !discover_position(X,Y);
     !align_to(Tx,Ty);
     move_forward(1);
     .wait({+position(Tx,Ty)});
     stop_moving.

+!test_compass <-
        move_forward(1);
        !wait_for(500);
        change_dir(90);
        !wait_for(500);
        change_dir(90);
        !wait_for(500);
```

```
change_dir(90);
!wait_for(500);
change_dir(90);
!wait_for(500);
stop_moving.
```

A proposito della carica, in sede di progetto sono state fatte alcune considerazioni. Il livello della carica diminuisce a fronte del movimento del robot, ma nel momento in cui inizia a pulire impiega più risorse e perciò il livello della batteria cala più rapidamente. Nel momento in cui il robot è completamente scarico, non esegue più alcuna azione e compare una lettera D di colore bianca sul robot. Il livello della batteria è monitorato a livello grafico da una barra di progressione posta in fondo all'interfaccia grafica.

Si nota che i parametri per la gestione della carica della batteria risultano rilevanti dal punto di vista delle prestazioni. La definizione del valore del livello di carica ottimo per tornare alla base è impossibile da stabile. Bisognerebbe sapere ogni volta il valore della carica necessario per tornare alla base di ricarica. Per la strategia di orientamento adottata è possibile calcolare la distanza tra il punto in cui è posizionato il robot e il punto in cui è posta la base di ricarica, ma non è possibile prevedere se lungo la strada si incontreranno ostacoli. Bisogna fare un compromesso tra lo scegliere un valore non troppo elevato, altrimenti parte della carica potrebbe non essere mai utilizzata ma si è sicuri che il robot raggiunga la base, ma neanche troppo basso, altrimenti se eventualmente lungo la marcia il robot incontrasse vari ostacoli si rischierebbe che rimanga scarico in mezzo alla stanza senza aver potuto terminare il suo lavoro, situazione mostrata in figura 5.4. In tale circostanza sarà necessario l'intervento di una entità esterna che riporta fisicamente il robot alla base di ricarica. Il codice **Jason** per la percezione del livello di carica si inserisce di seguito.

```
+battery_charge(C) : C < 40
  <-  .println("Warning - Low Battery: ",C).
```

Si noti che la percezione ha il contesto verificato solamente quando il livello di carica della batteria del robot è inferiore al 40%.

A livello grafico viene mostrato anche la diversa densità di sporco. Maggiore l'area è sporca, più scuro sarà il colore sull'interfaccia da grigio chiaro a nero. Il robot dunque impiegherà più risorse in termini di tempo e di ener-

gie (carica della batteria) a pulire una parte estremamente sporca rispetto ad una leggermente sporca.

In sintesi, le percezioni del robot sono:

- `detected_surface(Dist)`
- `touch_on(SensorId)`
- `moving(Speed)`
- `detected_surface(Dist)`
- `dirt`
- `position(X,Y)`
- `compass(Angle)`
- `battery_charge(B)`

La percezione `detected_surface(Dist)` si usa per decrementare la velocità del robot, nell'ottica di conservare pressoché integro il robot nel tempo. Il codice **Jason** che realizza la decelerazione, si inserisce di seguito.

```
+detected_surface(Dist) : moving(Speed) & Dist > 0.5 <-
    move_forward(Dist*0.9).
```

Si stabiliscono gli attuatori presenti sul robot:

- due servomotori nelle due ruote laterali
- una ruota posteriore sferica
- una spazzola

L'unico attuatore da commentare è l'ultima in elenco, ovvero la spazzola. Tale attuatore permette fisicamente al robot di pulire il pavimento dallo sporco. Quando il robot esegue l'azione di pulire (`start_cleaning`) questa spazzola viene attivata, e, di conseguenza, nel momento in cui non pulisce più (`stop_cleaning`) , la spazzola viene disattivata. L'utilizzo di tale spazzola comporta un consumo ulteriore di carica da cui deriva l'idea di un consumo maggiore di carica nell'intervallo di tempo in cui il robot pulisce.

Dati gli attuatori di cui è equipaggiato il robot, esso può svolgere determinate azioni:

- `move_forward(Speed)`
- `move_backward(Speed)`
- `stop_moving`
- `change_dir(Angle)`
- `start_cleaning`
- `stop_cleaning`

Si nota che le prime quattro azioni riguardano il movimento e sono state già descritte nell'esempio applicativo, mentre le altre due riguardano la pulizia della stanza.

Le azioni riguardanti il movimento sono già state descritte nell'esempio applicato, si sottolinea solo che il parametro `Angle` dell'azione `change_dir(Angle)` risulta positivo se l'angolo viene misurato in senso orario, altrimenti è negativo.

La strategia per la pulizia del robot è di iniziare a spazzolare le sporco, ma mentre sto eseguendo questa azione, continuo ad avere la percezione sul grado di sporco del pavimento di modo che proseguo la marcia, ovvero fermo le spazzole e proseguo, solamente nel momento in cui ho rimosso del tutto lo sporco che ho iniziato a pulire. In particolare il grado di sporco, inizialmente sarà ad un determinato livello per poi decrementare gradualmente fino ad arrivare a zero. A livello grafico, lo stato di cleaning del robot viene espresso da una lettera C di colore bianco sul robot, situazione mostrata in figura 5.2. Il codice **Jason** relativo alla gestione della percezione dello sporco viene inserito di seguito.

```
+dirt
  <-
      .println("DIRT! ");
      .drop_all_desires;
      stop_moving;
      start_cleaning.

-dirt
  <- .println("NO MORE DIRT.");
     stop_cleaning;
```

```
!explore.
```

La strategia di navigazione della stanza è pensato nell'ottica di realizzazione di una "serpentina". Riutilizzando il concetto di piano cartesiano per la spiegazione dell'orientamento, la strategia di navigazione è di percorrere una intera riga (coordinata y costante), ovvero finché non si sbatte contro il muro, per poi spostarsi alla riga successiva (si incrementa la y di uno) e percorre tutta questa diversa riga e così via.

Il problema è che nell'ambiente che si vuole gestire non sono presenti solo gli ostacoli in forma di muri, ma essi possono essere anche collocati dovunque all'interno della stanza. Per risolvere tale situazione, l'approccio è molto semplice: il robot non cerca di capire quale sia il movimento miglior ma subito cerca di allontanarsi dall'ostacolo in una direzione che potrebbe essere plausibile, cioè non lo si fa tornare indietro dritto per dritto ma lo si muovere in una certa direzione per una certa quantità di tempo piccola e poi continuare la sua marcia. In particolare i casi da gestire quando un robot incontra un ostacolo sono tre: il primo è quando l'ostacolo è un muro e perciò non può raggirarlo in alcun modo, il secondo è che lo incontra quando sta navigando la stanza al fine di pulirla, il terzo è, invece, quando il robot lo incontra nel momento in cui sta cercando di tornare alla base di ricarica. Nel primo caso l'approccio è quello di utilizzare la *nota mentale* `old_pos(X,Y)` e la bussola. Si ricorda che le percezioni per un agente durano solo per un ciclo di ragionamento. Se un agente ha bisogno di ricordare le cose dopo che sono state percepite, nel piano deve essere aggiunte le noti mentali nel momento in cui la percezione si è generata. La nota mentale viene ricordata dall'agente, finché non viene esplicitamente cancellata. Per interrogare una nota mentale si usano i test goal. La strategia è quella di controllare l'ultima posizione percepita e la direzione con cui il robot si sta muovendo così da essere in grado di sapere l'angolo di rotazione. Ad esempio, nel caso in cui il robot si sta muovendo verso Est e l'ultima posizione percepita è (9,1) allora il robot dovrà girarsi prima di 270°, proseguire per un secondo alla velocità 1 e poi girarsi di nuovo di 270°. In generale, si nota che se l'ultima posizione rilevata è (1,Y) allora il robot è vicino al muro di sinistra, riferendosi all'interfaccia grafica, viceversa che è (9,Y) allora il robot è vicino al muro di destra. Nel secondo caso il robot deve ricordarsi la riga che stava pulendo ed aggirare l'ostacolo in modo da proseguire lungo la stessa linea ma dall'altro lato della stanza. Nel terzo caso il robot dovrà allontanarsi dall'ostacolo, una volta che nella sua direzione di marcia non rileva più un

ostacolo allora deve trovare un tag RFID per localizzarsi e ricalcolare la direzione in cui deve muoversi per andare verso la base.

Funzionalità attuali del robot

L'insieme dei comportamenti da gestire che rendono l'attività complessa sono:

- strategia di navigazione della stanza che permetta al robot di pulire interamente la stanza
- strategia di trovare un percorso alternativo a fronte di un ostacolo che impedisce la marcia
- pulizia di sporco con vari livello di densità
- gestione del livello di carica della batteria

Funzionalità future del robot

Leggendo l'elenco precedente, vengono in mente altre funzionalità che sarebbe interessante realizzare, ad esempio:

- programmazione dell'orario di inizio della pulizia della stanza, facendo riferimento al tempo logico
- pulizia diversa a seconda del materiale del pavimento della stanza, ovvero se il pavimento è di legno, piuttosto che di marmo o di ceramica
- implementazione di una funzione che crea polvere di tanto in tanto, in questo caso il robot non smetterebbe mai di pulire
- inserimento di più basi di carica, così che il robot nel caso che abbia l'esigenza di ricaricarsi sceglie quella più vicina
- emissione di un segnale acustico nel caso in cui il robot rimanga scarico in mezzo alla stanza
- sperimentare strategie di navigazione alternative

- gestire la carica in maniera più intelligente possibile, nel caso in cui pulisca dei punti vicino alla base allora può iniziare a tornare alla base da un livello di carica più basso rispetto a quando si trova in punti più lontani

Nel caso di voler indagare la chiave multi-agente, si propone di:

- creare due tipi di robot, una tipologia che pulisce e l'altra che sporca
- creare tipologie di robot con abilità diverse, ad esempio un robot è in grado di lavare e un altro di asciugare, di conseguenza per pulire lo sporco serve l'azione di due robot e non di uno solo
- essendo situati più robot nella stanza, ogni robot si occupa di pulirne un'area

Capitolo 6

Conclusioni

6.1 Osservazioni su Jason

La nozione di *agente intelligente* viene sminuita dal fatto che la pianificazione avviene totalmente off-line. L'architettura BDI, che è il modello specifico su cui si basa Jason, non prevede nessuna forma una pianificazione on-line. Il motivo è che tale approccio richiederebbe all'agente una quantità di risorse temporali molto elevata, inoltre, mentre esso sta pianificando, il mondo potrebbe cambiare portando ad invalidare il ragionamento in corso. Lo scenario è che l'agente inizia a pianificare, dopo un po' di tempo produce il piano, ma mentre sta ragionando il mondo è variato, di conseguenza il piano prodotto *in questo momento* risulta inutile. L'approccio BDI si promette di evitare la ripianificazione, pensando al fatto che l'agente deve essere continuamente in fase di percezione dell'ambiente e la fase di pianificazione deve essere molto agile. Da qui emerge la motivazione della pianificazione off-line, il fatto di non poter pianificare dinamicamente si paga dal punto di vista della flessibilità, ma si riesce ad ottenere velocità, reattività ed efficienza nella selezione del piano da eseguire, e la scelta dell'azione del piano è un processo poco costoso. Da sottolineare è che se l'agente possiede molti piani nella libreria dei piani la reazione rimane comunque "abbastanza veloce", in quanto Jason utilizza per la selezione tecniche di tipo *hash*. Dal momento che i piani reagiscono a fronte di un evento, per Jason può essere relativo ad una percezione oppure ad un nuovo goal da realizzare, questi vengono definiti da una specifica struttura. Si utilizza la tecnica di ricerca di tipo hash (si pensi alle hash table di Java). Ogni evento ha una testa che

identifica la tipologia a cui appartiene, ovvero una chiave che mi permette di ottenere in un solo passo l'informazione ricercata. Si ottiene così l'insieme dei piani applicabili. La fase successiva è la valutazione del contesto, che oltre a richiamare la base delle credenze può anche chiamare funzioni complesse che restituiscono valori booleani, a partire dalla conoscenza che possiede l'agente in quel momento dell'ambiente. Se effettivamente tali funzioni risultano molto complessi, potrebbe essere necessario impiegare molto tempo. In generale, Jason ha cercato di realizzare un compromesso fra la reattività e la pianificazione.

La tesi non prevede di modificare Jason, ma una idea generata dal suo studio è quella di personalizzare la funzione di selezione dei piani applicabili dato che Jason è implementato in Java. La modifica riguarderebbe la possibilità di modificare la funzione in modo che crei un *thread* per ogni piano, il cui compito di ognuno è controllare il triggering event e il contesto di un determinato piano in modo da realizzare un controllo in parallelo. A questo punto il recupero di tutti i piani rilevanti (passo 6 del ciclo di ragionamento) viene svolta servendosi delle tecniche hash di ricerca (come già avviene), mentre la determinazione dei piani applicabili (passo 7) attraverso questo diverso approccio che verte sull'idea di parallelizzare il controllo (da realizzare in Java).

Un altro aspetto interessante è pensare come sarebbe possibile gestire più eventi in parallelo. In Jason, l'architettura interna presuppone che l'agente può avere più intenzioni in un determinato momento, però poi in realtà ha un unico flusso di controllo, nel senso che sceglie di volta in volta qual è l'attività. I piani in esecuzione possono essere molteplici in parallelo, però, di volta in volta viene eseguita solo una azione di un determinato piano. L'idea è di poter estendere l'architettura con l'idea di portare avanti in parallelo più piani. Si mette in luce però che potrebbero emergere dei problemi riguardo all'aggiornamento delle conoscenze, in particolare bisognerebbe gestire l'accesso alla base delle credenze per evitare interferenze fra le varie attività in esecuzione. Naturalmente, le riflessioni su questo aspetto di parallelizzazione hanno senso solo se si ha la possibilità di lavorare su una architettura multi-core.

6.2 Esplorazioni future

La tesi vuole essere un punto di partenza a chi interessa avventurarsi nello studio del linguaggio di programmazione ad agenti Jason. La tesi, pertanto, si rivolge prioritariamente agli studenti di Ingegneria dell'Informazione, ma anche a chi solo vuole farsi un'idea di questo nuovo mondo degli agenti.

Un intento di questa tesi è stato realizzare un modello software attraverso il quale è possibile realizzare sperimentazioni all'interno della facoltà perciò lo scenario che è stato considerato è *Lego-Mindstorms-like*. A fronte di questo, sarebbe molto interessante realizzare un modello fisico a partire da quello software.

Il messaggio è dunque di dire che Jason è interessante come approccio di alto livello di programmare robot ma ci sono diversi aspetti che devono essere approfonditi e che saranno oggetto di ricerca.

La sfida rimane, quindi, ancora aperta e, anche se la natura può fornirci un valido aiuto con i suoi modelli, sta nell'intuizione e nell'intelligenza degli esseri umani, in particolare degli ingegneri, saper trovare il modo giusto per concretizzarli nella realtà quotidiana.

Bibliografia

[1] Rafael H. Bordini, Jomi Fred Huber, Michael Wooldridge. Programming multi-agent systems in AgentSpeak using Jason. 2007. Wiley.

[2] Maja J Mataric. The Robotics Primer. 2007 The MIT Press.

[3] Stuart Russel, Peter Norvig. Intelligenza Artificiale: un approccio moderno. 2005. Pearson Education.

[4] Jason v.1.3.1 http://sourceforge.net/projects/jason/.