

Programmazione di robot con linguaggi ad agenti basati su
architettura BDI

Corso di Sistemi Multi Agente LM

Contoli Chiara
Santandrea Giuliano

11 gennaio 2013

Indice

1	Introduzione	3
1.1	Cos'è un robot?	3
1.2	Architetture di controllo e linguaggi di programmazione per Robot	4
1.2.1	Controllo deliberativo	5
1.2.2	Controllo reattivo	6
1.2.3	Controllo ibrido	7
1.2.4	Controllo behavior-based	7
2	Perché ha senso utilizzare un'architettura BDI?	9
2.1	Premessa	9
2.2	Principali caratteristiche dell'architettura BDI	10
2.2.1	Concetti dell'Agent-Oriented Programming	10
2.2.2	Jason e il ciclo di ragionamento	11
3	Framework di simulazione	15
4	Come programmare un robot ad agenti	19
4.0.3	Il framework di Simulazione: Simbad	19
4.0.4	CARTAgO: un ponte tra Simbad e Jason	20
4.0.5	Implementazione dell'integrazione Simbad + Jason + CARTAgO	22
5	Sperimentazione	26
5.1	Scenario 1: Walls	26
5.2	Scenario 2: Walls and cooling function	27
5.3	Scenario 3: Walls and noise	28
5.4	Scenario 4: Walls, cooling function and noise	28
5.5	Scenario 5: Following the light	28
5.6	Scenario 6: Pickup cherry Robot	29
5.7	Scenario 7: Maze Robot	30
5.8	Scenario 8: Maze Robot with Blind Robot	32
6	Simbad e i requisiti per la programmazione di robot cognitivi	35
6.0.1	Integrazione/interoperabilità	35

6.0.2	RealTime	35
6.0.3	Gestione della sensoristica	36
7	Conclusioni e sviluppi futuri	38

Capitolo 1

Introduzione

1.1 Cos'è un robot?

Al giorno d'oggi si parla sempre più spesso di Robot e dei loro domini di applicazione. Se in passato il robot era visto semplicemente come una macchina automatica capace di aiutare, e in certi casi sostituire, l'uomo nei compiti più ardui, oggi il concetto di robot è molto più sofisticato. L'evoluzione della tecnologia infatti, ha permesso di ampliare gli orizzonti introducendo in queste “macchine” concetti quali: il pensiero, il ragionamento, la risoluzione di problemi e altre caratteristiche vicine alle creature biologiche, come insetti ed esseri umani. Così, facendo riferimento a [6] si ha che:

Un *robot* è un sistema autonomo che esiste in un mondo fisico, capace di percepire il proprio ambiente, e può agire su di esso al fine di raggiungere i suoi obiettivi.

In questa definizione vi sono caratteristiche molto importanti da considerare:

- *autonomo*: il robot agisce sulla base delle proprie decisioni e non controllato dagli umani;
- *mondo fisico*: il robot si localizza nello stesso spazio fisico dove vi sono anche umani, animali e qualunque altra cosa esista nel mondo fisico;
- *percepire l'ambiente*: il robot ha *sensori* che permettono di reperire informazioni dall'ambiente;
- *agire su di esso*: il robot esegue azioni in risposta a quanto percepito dall'ambiente al fine di raggiungere il suo obiettivo;
- *raggiungere obiettivi*: il robot ha uno o più obiettivi e agisce per poterli raggiungere. Gli obiettivi possono essere molto semplici o molto complessi

1.2 Architetture di controllo e linguaggi di programmazione per Robot

Il lavoro di un controllore è quello di fornire un cervello al robot, in modo che possa essere autonomo e raggiungere i suoi obiettivi. Esistono tantissimi modi per programmare il controller di un robot, ma mettere insieme linee di codice non significa aver definito il comportamento di un robot. Per trovare il modo migliore di realizzare il controllore di un robot, dato un certo task, è necessario avere dei principi guida e dei vincoli che definiscono in maniera formale come organizzare il sistema di controllo del robot (il suo cervello). Questi principi e vincoli sono forniti dalle *architetture di controllo*, ovvero l'insieme degli strumenti necessari a progettare un controllore di robot in maniera più semplice.

Sensori, attuatori e decisioni hanno necessità di interagire in maniera efficace al fine di permettere al robot di compiere il suo lavoro. Un altro aspetto da considerare è il fatto che il controllore può essere implementato in Hardware (HW) o in Software (SW), ma più un controllore è complesso, più è probabile che sia implementato in SW. Questo perché l'implementazione in SW è flessibile e per programmi più generali; mentre l'implementazione HW è ideale per usi specializzati che necessitano alta velocità di esecuzione. Il controllore usa il programma per risolvere i problemi e raggiungere i suoi obiettivi. Il processo di risoluzione di un problema, utilizzando una procedura (finita) step-by-step è chiamato *algoritmo*. Anche la robotica riguarda essa stessa lo sviluppo e analisi di algoritmi per usi che sono rilevanti nei robot, come la navigazione, la manipolazione e l'apprendimento.

Dal momento che il controllore del robot è un programma, è lecito chiedersi quale sia il linguaggio più adatto per scrivere questi controllori. La verità è che non c'è un linguaggio che è migliore rispetto agli altri, poiché la scelta del linguaggio può dipendere da cosa si vuol cercare di far fare al robot, da qual è il loro utilizzo, dal HW del robot e così via. Quel che è certo è che un controllore può essere scritto in vari linguaggi. In teoria qualunque programma può essere scritto con qualunque linguaggio di programmazione. Il punto cruciale è la scelta del linguaggio di programmazione che meglio si adatta al task da compiere.

Si ritiene importante ricordare che le architetture esistono per fornire linee guida di progettazione per i controllori di robot, e linguaggi di programmazione che consentono di seguire facilmente queste linee guida sono particolarmente convenienti.

Trascurando i linguaggi usati per programmare i robot, nei paragrafi successivi viene fatta una breve panoramica sulle architetture usate per implementare i controllori. Le architetture differiscono per le regole e i vincoli che impongono alla struttura dei programmi dei robot e al relativo SW di controllo. Esistono i seguenti tipi di controllo:

- Controllo deliberativo;
- Controllo reattivo;
- Controllo ibrido;
- Controllo basato sul comportamento (o behavior-based (BB));

Ognuno di questi controlli definisce un'architettura ed è più o meno adatta a realizzare un controllore di robot per un dato task. Le architetture differiscono fondamentalmente nel modo in cui affrontano i seguenti aspetti:

- Tempo: quanto in fretta accadono le cose? i componenti del controller lavorano tutti alla stessa velocità?
Questo ha a che vedere con quanto velocemente un robot deve rispondere al suo ambiente, comparato con quanto velocemente il robot è in grado di percepire e pensare
- Modularità; quali sono i componenti del sistema di controllo? chi può parlare con chi?
Questo ha a che vedere con il modo in cui il sistema di controllo è diviso in pezzi o componenti, e al modo con cui questi interagiscono, al fine di realizzare il comportamento nel suo complesso.
- Rappresentazione: che cosa sa il robot e che cosa tiene nella sua mente?

1.2.1 Controllo deliberativo

Il controllo deliberativo guarda al futuro, quindi lavora su una lunga scala temporale (quanto lunga dipende da quanto lontano si guarda nel futuro). Il sistema di controllo è diviso in più moduli, i quali includono le percezioni, la pianificazione e le azioni; i moduli lavorano in maniera sequenziale, ovvero l'output di un modulo diventa l'input del successivo. Le cose avvengono una per volta, non tutte allo stesso momento.

Gli approcci puramente deliberativi trovano la maggiore applicazione nell'ambito del problem-solving come il gioco degli scacchi, in cui la pianificazione e la strategia sono la cosa più importante da fare. In certe situazioni, prendere del tempo per valutare diverse possibilità è possibile (cioè c'è il tempo per farlo) e necessario (senza un minimo di strategia, le cose non funzionano). La *pianificazione* è il processo tramite il quale si valutano gli esiti delle possibili azioni, e si ricerca la sequenza di azioni che permettono di raggiungere il goal. La *ricerca* deriva dalla pianificazione. Consiste nel ricercare la soluzione all'interno della rappresentazione dei possibili stati. Se gli stati sono tanti, questo può richiedere anche tantissimo tempo. Più è lunga la pianificazione, meno il robot sarà reattivo ai "pericoli". Esistono dei metodi per ottimizzare la pianificazione. C'è un limite a quello che può essere fatto in tempo reale. In genere, un robot fisico che si muove in un ambiente dinamico è soggetto a vincoli temporali. Un robot non può semplicemente sedersi e decidere. L'approccio deliberativo consiste di tre passi che devono essere fatti in sequenza Sense, Planning, Acting (eseguire il piano) (SPA). Questo tipo di architetture ha dei seri svantaggi per i robot:

1. la ricerca nello spazio degli stati può portare via molto tempo. Combinare tutti i possibili dati dai sensori insieme all'uso di mappe può ulteriormente ingrandire lo spazio degli stati. Se il tempo di pianificazione è molto più lento del tempo impiegato dal robot per muoversi, egli dovrà fermarsi e aspettare che abbia finito di pianificare, al fine di preservarsi. In tal caso, è meglio che pianifichi il meno possibile e si muova il più possibile. Se la pianificazione è veloce, questa può essere fatta ad ogni step. Nei problemi reali di robot, questo è impossibile.

2. la rappresentazione può occupare molta memoria, anche se il problema più grande può essere dato dagli algoritmi, che possono esaurirla durante l'esecuzione.
3. si assume che le informazioni siano sempre aggiornate, il che è ragionevole, perché se no la pianificazione sarebbe inutile.
4. l'uso dei piani è utile solo se: l'ambiente non cambia durante l'esecuzione del piano, in modo da "danneggiare" il piano; il robot sa sempre in che stato è; ci si porta nello stato successivo.

Architetture puramente deliberative non sono più in uso per la maggior parte dei robot fisici a causa delle combinazioni tra sensori, attuatori e le sfide temporali. Ci sono però eccezioni, applicazioni rare in cui è necessaria una forte pianificazione e non vi sono pressioni temporali; casi in cui l'ambiente è statico. La chirurgia robotica è uno di questi. Approcci puramente deliberativi si trovano fuori dalla robotica, ad esempio nel gaming. L'approccio SPA non è stato comunque abbandonato in robotica.

1.2.2 Controllo reattivo

Il controllo reattivo guarda all'immediato, alle richieste real-time da parte dell'ambiente, senza guardare né al passato, né al futuro, quindi lavora su piccole scale temporali. Il sistema di controllo è diviso in più moduli tutti attivi contemporaneamente e in grado di scambiarsi messaggi in vari modi. In questo caso le cose accadono allo stesso tempo e non una alla volta.

I sistemi reattivi consistono in un insieme di situazioni (stimoli, condizioni) e in un insieme di azioni elementari (risposte, azioni o comportamenti). Esiste un mapping stretto tra i sensori e gli attuatori dei robot. L'approccio reattivo è uno dei più comunemente usati per i controller dei robot. Approcci puramente reattivi non fanno uso di alcuna rappresentazione interna dell'ambiente, poiché si limitano a ricevere input dai sensori e ad agire di conseguenza. Per avere un sistema reattivo semplice, l'ideale è avere degli stati che possono essere univocamente individuati dal robot, quindi le condizioni devono essere mutuamente esclusive. Individuare tutte le possibili situazioni del sistema è comunque difficile. L'architettura più conosciuta per il controllo reattivo è l'*architettura a sussunzione* ideata dal Prof. Rodney Brooks presso il MIT nel 1985. L'idea di questo tipo di architettura è costruire il sistema in maniera incrementale, dalle parti semplici a quelle più complesse. Consiste in una collezione di moduli ognuno dei quali adempie un task. Tutti i layer lavorano contemporaneamente e non in sequenza. L'idea è che i livelli superiori possono assumere l'esistenza dei livelli inferiori e dei loro obiettivi, perciò i livelli superiori possono usare quelli inferiori per raggiungere i propri obiettivi. Questi layer sono progettati, implementati e testati in maniera incrementale. Ogni layer aggiunto viene aggiunto sopra quello precedente. Il layer superiori possono però temporaneamente disabilitare le funzionalità dei layer inferiori.

1.2.3 Controllo ibrido

Il controllo ibrido combina le lunghe scale temporali del controllo deliberativo con quelle brevi del controllo reattivo. Il sistema di controllo è costituito da tre moduli principali: la parte deliberativa, la parte reattiva e una parte che si frappona fra queste due. Queste parti lavorano in parallelo e comunicano anche tra di loro. Come descritto in [7], la forse più conosciuta e usata architettura ibrida è la *Three Layered Architecture* (TLA). Questa architettura è composta da tre componenti:

1. componente funzionale: si interfaccia con l'HW e fornisce percezioni e azioni a basso livello;
2. componente deliberativo: produce piani per far raggiungere al robot i propri obiettivi, supervisionando anche l'esecuzione temporale dei piani;
3. componente esecutivo: interposto tra i due componenti precedenti e la cui funzionalità è dipendente dal contesto di esecuzione del piano generato dal componente deliberativo. Trasforma i piani in azioni di basso livello, che possono essere eseguite per controllare i moduli del livello funzionale. Reagisce anche agli eventi e fornisce meccanismi limitati per il monitoraggio e il recovery dai fallimenti dei piani.

Il principio di progetto dietro la TLA è incapsulare l'attività time-consuming del processo deliberativo all'interno di un componente deliberativo e incrementare la reattività fornendo separatamente un componente esecutivo con un tempo di reazione più veloce rispetto a quello del componente deliberativo.

Siccome l'attività di pianificazione può essere molto costosa in termini temporali, è utile memorizzare i piani una volta che sono stati generati. Una buona idea è quella di pre-pianificare per tutte le possibili situazioni che potrebbero accadere, e memorizzarli per poi averli disponibili nel tempo. Questo approccio è noto con il nome di *pianificazione off-line*, che avviene durante la fase di sviluppo del robot (progettazione/implementazione del controller del robot). L'approccio duale consiste nella *pianificazione on-line*, ovvero quando il robot, a tempo di esecuzione, deve preoccuparsi di costruire il piano lui stesso. La pianificazione off-line è vista di buon occhio perché consente di avere un buon compromesso tra l'architettura deliberativa e quella reattiva.

1.2.4 Controllo behavior-based

Il controllo BB lavora per mettere le scale temporali dei controlli deliberativi e reattivi insieme. Questo controllo è provvisto di solito di più di tre moduli principali, ognuno dei quali si occupa di portare a termine un certo compito. I moduli lavorano in parallelo e sono in grado di parlare tra di loro, ma in maniera diversa rispetto a quella in cui comunicano i moduli del controllo ibrido. Questo tipo di controlli hanno un componente reattivo (come nelle architetture ibride), ma non hanno alcun componente deliberativo. Hanno dei moduli che rappresentano dei "behavior". Quindi il BB è implementato come una collezione di behaviors. I behavior incapsulano e/o mantengono particolari goals; si prolungano nel tempo,

nel senso che non sono istantanei ma richiedono una certa quantità di tempo per essere portati a termine; sono in grado di ricevere input dai sensori e anche da altri behavior; sono più complessi delle azioni. I behavior tipicamente sono eseguiti parallelamente/concorrentemente al fine di rispondere in maniera più immediata quando necessario. Inoltre, sono progettati per impiegare lo stesso tempo di esecuzione, ovvero, non è l'ideale avere behavior molto più veloci o più molto lenti di altri.

Capitolo 2

Perché ha senso utilizzare un'architettura BDI?

Il capitolo precedente si è concluso con una piccola panoramica sulle architetture principalmente utilizzate nel corso degli anni per la programmazione di robot simulati a computer (robot SW), mettendo in luce vantaggi e svantaggi di ciascun approccio. L'obiettivo di questo elaborato è l'esplorazione e la sperimentazione della programmazione di robot utilizzando un linguaggio ad agenti basato su architettura BDI. Esistono molteplici linguaggi ad agenti basati su tale architettura; il linguaggio di riferimento utilizzato per la sperimentazione è *Jason*, una implementazione di AgentSpeak [9].

2.1 Premessa

Fino ad ora si è parlato di Robot facendo riferimento a quelli fisici, ovvero Robot che si muovono in un ambiente fisico reale. L'esplorazione e la sperimentazione condotta fa invece riferimento a Robot che esistono all'interno di un computer, e quindi costituiscono una simulazione. Una simulazione è una rappresentazione semplificata del mondo reale, e quindi il Robot non si scontra per davvero con le proprietà reali del mondo fisico poiché le simulazioni non sono mai così complesse come il mondo reale. Un robot cibernetico (o SW), non è perciò un vero Robot. Questa distinzione è d'obbligo poiché nel corso del lavoro non sono state prese in considerazione problematiche importanti tipiche dei robot reali. Le principali problematiche riguardano:

- sensori: nel mondo reale questi sono spesso affetti da errori di misura, dovuti all'HW stesso o all'ambiente in cui il Robot si muove;
- attuatori: anch'essi possono essere soggetti ad errori dovuti al rumore;
- ambiente: nel mondo reale l'ambiente è principalmente dinamico e soggetto a rumore;
- velocità di percezione e azione: in un ambiente reale, dinamico, al fine di preservare la propria incolumità, quella di altri Robot, o persone che si muovono nell'ambiente,

e anche preservare l'ambiente stesso, il Robot deve essere particolarmente veloce a reagire agli stimoli esterni. Questi aspetti toccano inevitabilmente le tematiche del real-time.

Sono state perciò fatte diverse assunzioni per semplificare le condizioni del contesto in cui si muoverà il robot; ciò nonostante, si è comunque voluto approfondire la ricerca leggendo quanto presente in letteratura, senza però condurre sperimentazioni, riguardo alle tematiche del real-time nel contesto BDI, che sarà trattato in seguito.

2.2 Principali caratteristiche dell'architettura BDI

Come viene descritto in [9], l'architettura è basata sul modello del comportamento umano che fu sviluppato dai filosofi. Questo modello è chiamato *Belief-Desire-Intention* (BDI). L'architettura ha origine dal lavoro sul progetto Rational Agency condotto presso lo Stanford Research Institute a metà degli anni '80. Il modello si basa principalmente sulla teoria del *human practical reasoning* sviluppata dal filosofo Michael Bratman, il quale si focalizzò in particolare modo sul ruolo delle intenzioni nel ragionamento pratico. Il practical reasoning è il *ragionamento orientato alle azioni*, ovvero il processo di decidere cosa fare.

I domini applicativi dei Robot richiedono sempre più spesso un crescente livello di *intelligenza* e *autonomia*, che per i Robot cognitivi costituiscono una vera sfida. Vi sono rami della robotica il cui obiettivo è studiare e sviluppare Robot che possiedono capacità di raggiungere obiettivi complessi in ambienti dinamici. Questa è una delle ragioni per la quale l'architettura BDI ci sembra una buona idea, in quanto fornisce principi molto interessanti con i quali si possono progettare controllori di Robot.

2.2.1 Concetti dell'Agent-Oriented Programming

Vi sono tre concetti:

- *Belief*: sono informazioni che l'agente ha sul mondo; le informazioni potrebbero anche non essere aggiornate o non accurate;
- *Desires*: costituiscono le volontà dell'agente, ovvero obiettivi che l'agente vuole raggiungere; il fatto che l'agente abbia un obiettivo non implica che egli agirà per raggiungerlo; inoltre, è possibile che l'agente abbia obiettivi che non sono compatibili tra loro;
- *Intentions*: costituiscono l'insieme di azioni che portano l'agente a realizzare il proprio obiettivo.

Parlando di questi concetti è stata più volte ripetuta la parola *agente*. Di seguito vengono riportate le caratteristiche degli agenti:

- costituiscono un sistema *situato* in un *ambiente*;

- sono in grado di *percepire* tramite *sensori* l'ambiente che li circonda, e posseggono un insieme di *azioni* che posso eseguire attraverso degli *attuatori* al fine di modificare l'ambiente.

Gli agenti non hanno il completo controllo dell'ambiente. Un agente possiede inoltre le seguenti proprietà:

- *autonomia*: dare una definizione di autonomia non è semplice; l'agente può far parte di una società (vivere in un ambiente con più individui). L'autonomia esiste solo se vi sono più individui. In una definizione semplificata, l'autonomia consiste nella capacità di scegliere cosa fare in completa indipendenza.
- *proattività*: capacità di far capitare qualcosa invece di aspettare.
- *reattività*: capacità di rispondere ai cambiamenti che avvengono nell'ambiente.
- *abilità sociali*: capacità di cooperare e coordinare attività con altri agenti. Sono interattivi, scambiano informazioni

Enunciando le caratteristiche degli agenti è possibile ritrovare diverse analogie con le caratteristiche dei robot enunciate in precedenza. Questa è un'altra ragione per la quale l'architettura BDI ci sembra una buona idea per sperimentare la programmazione di robot.

2.2.2 Jason e il ciclo di ragionamento

Il linguaggio di riferimento utilizzato è *Jason*. In questo paragrafo viene introdotto in breve il linguaggio e come funziona il ciclo di ragionamento di Jason.

Jason permette di esprimere i concetti dell'architettura BDI offrendo opportuni costrutti al programmatore. Per i dettagli del linguaggio, si rimanda a [9].

L'agente è provvisto di una *base di conoscenza* (*belief base*) che nella sua forma più semplice è costituita da una collezione di letterali. In linguaggi ispirati dalla logica le informazioni sono rappresentate sotto forma di predicati che esprimono una particolare proprietà. I belief rappresentano quindi proprietà del mondo che l'agente crede siano vere.

L'altra nozione molto importante è il *goal*, il quale rappresenta una proprietà del sistema che l'agente vuole raggiungere. Jason distingue tra due tipi di goal:

- *achievement goals*: rappresentano gli "stati" che l'agente vuole raggiungere;
- *test goal*: utilizzati normalmente per reperire informazioni dalla belief base; è considerato un goal in quanto potrebbe accadere di non riuscire a portare a termine questo obiettivo, perché non è presente l'informazione cercata nella belief base.

L'ultima importante nozione è quella del *piano*. I piani costituiscono il know-how dell'agente, cioè come egli intende raggiungere i suoi obiettivi. Il piano è costituito da tre parti:

- *triggering event*: rappresenta l'evento che determina un cambiamento della conoscenza dell'agente;

-
- The diagram illustrates the architecture of an agent, showing the flow of information and decision-making processes. The agent's components and their interactions are as follows:
- Percepts** enter the **perceptive** block (1).
 - The **perceptive** block (1) connects to the **BUF** (2).
 - The **BUF** (2) interacts with the **BRF** (3) and the **SocAcc** (4).
 - The **SocAcc** (4) connects to the **checkMail** block (3).
 - The **checkMail** block (3) sends **Messages** to the **checkMail** block (3).
 - The **BRF** (3) sends **Beliefs** to the **Belief Base** and **Events** to the **Check Context** block (7).
 - The **Check Context** block (7) sends **Relevant Plans** to the **Unify Event** block (6).
 - The **Unify Event** block (6) sends **Plans** to the **Plan Library**.
 - The **Check Context** block (7) sends **Applicable Plans** to the **Selected Intention** block (9).
 - The **Selected Intention** block (9) sends **Intended Means** to the **New Intention** block (10).
 - The **New Intention** block (10) is part of the **Intensions** structure.
 - The **Intensions** structure includes **Push New Plan** and **New** blocks.
 - The **Intensions** structure sends **Updated Intention** to the **sendMsg** block.
 - The **sendMsg** block sends **Messages** to the **act** block.
 - The **act** block sends **Actions** to the **Messages** block.
 - The **Intensions** structure also sends **Suspended Intensions** (Actions and Msgs) to the **SocAcc** block (4).

La figura 5.1 è tratta da [9].
Le percezioni giocano un ruolo fondamentale. Quello che è rilevante sono le percezioni che determinano un cambiamento nello stato della conoscenza dell'agente. Se il programmatore

non apporta modifiche alle funzioni del ciclo di ragionamento di Jason, la funzione adibita alla gestione delle percezioni sceglie *un* solo evento rilevante per ciclo, tra tutti i possibili eventi presenti attualmente nel buffer. Siccome un'altra sorgente di informazione molto importante è la conoscenza appresa da altri agenti che fanno parte del sistema, ad ogni ciclo l'interprete controlla la presenza di eventuali messaggi nella mailbox. Anche qui, se non viene personalizzata la funzione di selezione, i messaggi saranno processati così come sono stati ricevuti. Ad ogni ciclo, viene quindi scelto un solo evento rilevante che dovrà essere gestito. Gli eventi sono gestiti sotto forma di lista e, se non viene personalizzata per definire un priorità nella selezione, viene scelto sempre il primo evento della lista, mentre nuovi eventi saranno aggiunti in coda. Gli eventi sono quindi gestiti da una coda FIFO. Una volta che un evento viene selezionato, questo viene rimosso dalla coda. Una volta scelto l'evento, è necessario scegliere un piano che permette all'agente di gestire l'evento. La prima cosa che viene fatta è cercare tutti i piani *rilevanti* per quell'evento. Questo significa cercare tutti quei piani che possiedono un *triggering event* che unifica con l'evento selezionato. Qui viene quindi determinato l'insieme di tutti i possibili piani, dal quale verrà selezionato quello applicabile. Da questo insieme vengono selezionati quelli *applicabili*, ovvero viene selezionato il piano che, in base alla conoscenza attuale dell'agente, ha più possibilità di avere successo. Per fare ciò, viene controllata la veridicità del contesto del piano. Se vi sono più piani attualmente applicabili, sarà necessario scegliere uno di questi. Avere più piani applicabili significa che potenzialmente più piani possono gestire con successo l'evento scelto. L'agente dovrà però scegliere *uno* solo di questi piani e impegnarsi nella sua esecuzione. Impegnarsi a eseguire il piano significa che l'agente ha l'*intenzione* di perseguire le azioni di quel piano. Se l'agente deciderà di perseguirlo, il piano finirà nell'insieme delle Intenzioni. Un evento è inteso sia come un cambiamento dell'ambiente a cui l'agente deve reagire, sia come un goal che l'agente vuole raggiungere. Inoltre, si fa notare che i goal attualmente presenti nell'insieme degli eventi sono tutti possibili *desideri* nei quali l'agente può impegnarsi, mentre tutti i possibili piani applicabili per quell'evento sono possibili sequenze di azioni che l'agente può compiere al fine di raggiungere l'obiettivo. La funzione che sceglie il piano applicabile lo fa in base all'ordine in cui questi sono stati scritti dal programmatore nel codice sorgente, o nell'ordine in cui questi sono stati scambiati tra gli agenti. Si distingue tra due tipi di eventi:

- *interni*: si ha un cambiamento negli obiettivi dell'agente;
- *esterni*: si ha un cambiamento percepito dall'ambiente;

Gli eventi interni sono creati quando l'agente ha un nuovo obiettivo da raggiungere; questo significa che, prima che il corso di azioni che hanno generato l'evento venga ripreso, è necessario trovare ed eseguire fino al completamento un piano per raggiungere l'obiettivo. Questo significa che, un evento interno, invece che essere visto come nuova intenzione, viene inserita come intenzione in cima alla pila delle intenzioni esistenti. Viene quindi a crearsi uno *stack* di piani (per una certa intenzione). Tipicamente l'agente possiede più intenzioni che competono tra di loro, ognuna rappresentante un focus di attenzione. In ogni caso, ad ogni ciclo di ragionamento viene eseguita solamente un'azione dell'intenzione scelta. Le intenzioni da perseguire seguono di default uno scheduling di tipo RoundRobin, ovvero a turno,

ogni intenzione è selezionata e di queste viene eseguita una sola azione del piano. L'agente divide quindi equamente la sua attenzione sulle proprie intenzioni. L'ultimo passo del ciclo di ragionamento consiste nell'esecuzione di una azione. Una intenzione è un parziale corso di azioni che l'agente ha scelto di perseguire. Tipicamente i piani sono costituiti da una o più azioni (scritte nel body). Le azioni qui contenute saranno eseguite in ordine. Una volta che una azione è stata eseguita, questa viene cancellata dal body del piano. Si distingue invece tra due tipi di azioni:

- *internal action*: metodi Java che ritornano un booleano che comunica se l'esecuzione è avvenuta oppure no;
- *basic action* (o *environment action*): azioni che rappresentano gli attuatori all'interno della architettura dell'agente;

Si fa notare che se un'azione fallisce, fallisce anche il piano in cui si trova l'azione.

Il ciclo di ragionamento di Jason è personalizzabile dal programmatore, che può decidere di re-implementare una o più parti di esso per adattare il ragionamento dell'agente al proprio caso di studio.

Date le peculiarità messe a disposizione dal linguaggio, nel prossimo capitolo ci occuperemo dei framework di simulazione di Robot attualmente presenti, illustrandone le principali caratteristiche.

Capitolo 3

Framework di simulazione

In letteratura esistono diversi simulatori di Robot. Prima di effettuare la scelta sono stati consultati alcuni framework, tra cui:

- ARGoS;
- VRep;
- Webots;
- SimPark;
- Gazebo;
- MRDS;
- UsarSIM;
- Simbad;

Di seguito vengono illustrate le caratteristiche principali.

ARGoS è un simulatore multi-robot capace di simulare contemporaneamente un gran numero di robot. E' stato sviluppato nell'ambito dello Swarmanoid Project con lo scopo di fornire uno strumento flessibile per la simulazione, scalabile ed computazionalmente efficiente. I tipi di robot che si possono simulare sono gli **eye-bot**, specializzati nel fare sensing e analisi dell'ambiente da posizioni privilegiate, i **foot-bot**, specializzati nel muoversi sul terreno e nel trasporto di oggetti (o altri robot) e **hand-bot**, che si muovono a una altezza intermedia rispetto ai primi due e sono in grado di scalare superfici e oggetti. E' caratterizzato da un'architettura modulare, alto livello di tuning di simulazione, supporto per il multi-threading, possibilità di usare molteplici motori fisici. ARGoS è inoltre un emulatore, ovvero il codice sviluppato per il controller dei robot simulati può poi essere "migrato" sui robot reali. È rilasciato con licenza open source e supporta Linux e Mac.

VRep è un simulatore 3D multi-robot basato su un'architettura di controllo distribuita. I

programmi di controllo possono venire attaccati agli oggetti della scena 3D. Questi programmi possono essere degli script, dei nodi ROS [12] o un controller remoto che VRep è un simulatore 3D multi-robot basato su un'architettura di controllo distribuita. I programmi di controllo possono venire attaccati agli oggetti della scena 3D. Questi programmi possono essere degli script, dei nodi ROS [12] o un controller remoto che interagisce, via socket, tramite delle opportune API. I linguaggi supportati sono C/C++, Python, Java, Lua, Matlab or Urbi. Supporta i motori fisici ODE [13] e Bullet [1], meccanismi di risoluzione cinematica diretta/inversa (utili se i robot posseggono giunzioni meccaniche complesse), controlli di collisione, dispone di algoritmi per il path planning da utilizzare nei controller, videocamera simulata con possibilità di image processing, sensori di prossimità, possibilità di integrare modelli CAD. Il codice è parzialmente open-source, in particolare sono open source i codici delle remote API e dell'interfaccia a ROS. E' a pagamento ma è disponibile una versione gratuita per studenti per utilizzo personale. Supporta Mac, Windows e Linux.

Webots è un ambiente di sviluppo per modellare, programmare e simulare robot mobili. Si possono decidere tutte le proprietà fisiche di ciascun oggetto simulato (forma, massa, colore, attrito, ..). I controller si scrivono sfruttando le API per i linguaggi C, C++, Java, Python, Matlab, URBI. Si può collegare Webots a ROS, per sfruttare lo stack di tecnologie che questo mette a disposizione. La simulazione sfrutta il motore fisico ODE, sono disponibili algoritmi di collision detection personalizzabili. Permette di simulare una vasta gamma di robot commerciali fra cui il cane Aibos, e-puck, HOAP-2, iRobot Create, Katana, Kheperas, Koala, Kondo KHRs, Nao, Pioneer, Shrimp III, Surveyor SVR-1, etc. In particolare per alcuni robot (es. l'E-Puck) è possibile portare il codice nel robot fisico. Webots è disponibile per Windows, Mac e Linux. E' largamente impiegato nel mondo della ricerca, dove è utilizzato nello studio di genetic algorithms, particle swarm optimization, artificial neural networks, swarm intelligence, stereo-vision, etc. La licenza è commerciale ma sono disponibili degli sconti per studenti.

SimPark è un simulatore fisico multi robot general-purpose in ambiente 3D. È stato usato come ambiente di simulazione per la RoboCup 3D Soccer Simulation League al RoboCup 2010 di Singapore. Robocup è una iniziativa con lo scopo di incoraggiare la ricerca sull'intelligenza artificiale e la robotica fornendo un problema standard, che consiste in una competizione di calcio fra robot Nao [11]. I motori fisici utilizzati sono ODE, Bullet, Physix. Le simulazioni vengono definite tramite un opportuno "scene description language", mentre i robot simulati vengono controllati da remoto tramite socket. Disponibile per Windows, Mac e Linux ed è con licenza open source.

Gazebo è un simulatore 3D e rientra nel progetto Player/Stage. Questo progetto è sviluppato da una comunità internazionale di ricercatori e utilizzato in molti laboratori di ricerca. È composto dai componenti Player, Stage e Gazebo. Player è un robot server che fornisce l'accesso e il controllo a una piattaforma robotica e si occupa di simulare il robot e i suoi sensori/attuatori; Stage permette la simulazione di gruppi numerosi di robot (ma in ambiente 2D) e infine Gazebo è un'estensione di Stage per permettere la simulazione di un piccolo numero di robot, ma in un ambiente 3D complesso e realistico. Gazebo sfrutta il motore fisico ODE. I sensori simulabili includono i sonar, scanning laser range-finders, GPS, monocular e

stereo camera. I robot simulabili includono i PR2, Pioneer2DX, Pioneer2AT e SegwayRMP. I robot sono programmabili in C,C++, Python e Ruby. Gazebo è inoltre compatibile con ROS. E' rilasciato come software libero, licenza GPL.

MRDS Microsoft Robotics Developer Studio è un framework di controllo e simulazione per robot sviluppato dalla Microsoft e liberamente scaricabile, anche se closed-source. È basato su un modello service oriented scalabile e dispone di un'interfaccia di programmazione testuale o visuale. Ogni entità della simulazione dispone di un set di callback che vengono invocate quando avviene un evento (es. collisioni con altre entità, frame update). È integrato con il motore fisico Physics della Ageia. Disponibile per windows.

UsarSIM Urban Search and Rescue Simulation è simulatore sviluppato originariamente per la ricerca nell'ambito della Search and Rescue (SAR) e per il RoboCup contest [12]. Sfrutta l'Unreal Engine per la grafica e la fisica, che garantisce un buon livello di accuratezza nella simulazione fisica, efficienza computazionale e flessibilità. I robot disponibili sono di vari tipi: con ruote, con gambe, volanti e marini. Gli attuatori e i sensori sono personalizzabili, con possibilità aggiungere dei modelli di rumore per renderli più realistici. sono controllabili tramite connessione TCP. Sono state fatte delle valutazioni quantitative che dimostrano l'alta fedeltà del comportamento dei robot simulati rispetto a quelli reali. È opensource e funziona su Linux e Windows [3] .

Simbad: è il simulatore utilizzato per il progetto. I motivi che hanno portato a questa scelta sono:

- possibilità di comprendere il funzionamento interno: essendo scritto in Java è stato possibile capire nel dettaglio il funzionamento interno del simulatore
- maggiore facilità di integrazione: anche Jason e CArtaGO sono scritti in Java. Non si è dovuti ricorrere a delle interfacce native come JNI nel caso si fosse utilizzato un simulatore scritto in un altro linguaggio
- bassa complessità del simulatore: Simbad, a differenza di altri simulatori, non si prefigge come scopo il fornire una rappresentazione della fisica il più accurata possibile [8], ma si concentra di più sulla semplicità di utilizzo

Il simulatore non ha una netta distinzione fra il tempo logico e il tempo fisico: questo difetto rende ogni simulazione dipendente dalla macchina su cui questa viene eseguita, rendendola inoltre non ripetibile. Inoltre, anche se il simulatore ha una corretta modellazione del tempo logico e quindi le simulazioni sono ripetibili, non appena gli viene affiancato un controllo che ha un suo flusso di controllo differente (in questo caso l'ambiente Jason) le simulazioni ritornano ad essere non ripetibili a seconda della sincronizzazione adottata. Il sistema è composto da:

- il simulatore di robot Simbad
- i controller dei robot scritti in Jason
- artefatti CArtaGO (che fanno da ponte fra i primi due)

In questo paragrafo viene anche riportata una breve comparazione con il framework LeJOS [4]. E' stato collegato Jason al robot Lego MindStorms NXT, in modo da poter scrivere in linguaggio ad agenti un controller per quest'ultimo. Jason viene fatto girare su una macchina esterna al robot e, via bluetooth, comunica le azioni da eseguire al robot mentre questo invia le percezioni a Jason. La differenza principale rispetto al nostro framework è che il collegamento fra Jason e il robot è di tipo sincrono, ovvero il ciclo di ragionamento procede sincronizzato con l'esecuzione delle azioni da parte del robot. Si forma quindi un unico flusso di controllo in cui il ciclo di ragionamento di Jason deve necessariamente attendere che il robot invii via bluetooth le percezioni dei sensori, poi Jason invia al robot le azioni da compiere e infine deve attendere che il robot abbia completato queste azioni prima di procedere con un nuovo ciclo di ragionamento. La conseguenza ovvia è che il tutto è molto più lento, ma a differenza del nostro framework non si devono usare delle code e sia Jason che il robot sono sempre aggiornati. In Jason + Cartago + Simbad si fa utilizzo di code per le azioni e per le percezioni, di conseguenza si possono verificare situazioni in cui Jason non possiede i belief più aggiornati. L'altra differenza è che viene controllato un robot reale che si muove in un ambiente fisico reale: entrano automaticamente in gioco tutti quei fattori come il rumore dei sensori, l'incertezza degli attuatori che in una simulazione vanno introdotti esplicitamente.

Capitolo 4

Come programmare un robot ad agenti

Il simulatore scelto per la nostra sperimentazione, come già anticipato nel capitolo precedente, è interamente scritto in Java; questo ha semplificato la sua integrazione con gli strumenti necessari alla nostra sperimentazione. In questo capitolo viene spiegato quali sono gli strumenti che si intendono adottare per la sperimentazione, le considerazioni fatte per poter realizzare l'integrazione e come questa integrazione è stata realizzata. L'integrazione è stata realizzata con l'aiuto e la collaborazione del prof. Alessandro Ricci.

4.0.3 Il framework di Simulazione: Simbad

Il primo passo consiste con il prendere confidenza con il framework di simulazione scelto. Le prime informazioni sono state tratte da [5] e [8]. A questo link, oltre ad un breve guida, è possibile reperire tutto il materiale necessario per lavorare con il simulatore: il codice sorgente di tutte le classi che compongono il simulatore, e la javadoc.

Con Simbad è possibile costruire il proprio controller del robot fornendo tre elementi:

- un main program: lancia Simbad e l'ambiente descritto;
- una descrizione dell'ambiente: descrive gli elementi presenti nell'ambiente;
- la classe del robot: contiene il controller del robot;

La classe che funge da controller del robot deriva da una classe *Agent* propria del simulatore. Per definire il controller è sufficiente sovrascrivere due metodi:

- *initBehavior()*: chiamato dal simulatore appena l'agente-robot viene istanziato, serve, eventualmente, per inizializzare alcune cose del robot;
- *performBehavior()*: descrive il comportamento del robot; chiamato dal simulatore ad ogni step di simulazione (di default, 20 volte al secondo; come si vedrà in seguito questo aspetto è indice della mancata distinzione fra tempo logico e fisico, un difetto di Simbad).

Il control loop del simulatore è il seguente per ogni agente (robot) presente nell'ambiente:

1. controlla eventuali collisioni geometriche contro altri oggetti;
2. aggiorna i sensori (se presenti) in base alla posizione corrente e alla frequenza di aggiornamento di questi;
3. aggiorna gli attuatori (se presenti) secondo la frequenza di questi;
4. aggiorna la posizione in base ai parametri della cinematica (velocità di traslazione e di rotazione);

Come i dispositivi reali, sensori e attuatori potrebbero non essere aggiornati ad ogni step di simulazione, poiché questo dipende dalla loro frequenza di aggiornamento.

Il nostro obiettivo è scrivere l'analogo del `performBehavior()` non in linguaggio Java ma in Jason, sfruttando comunque l'ambiente e le primitive messe a disposizione del simulatore. Per fare ciò, è necessario unire a Simbad un altro framework: Jason e CArtAgO.

4.0.4 CArtAgO: un ponte tra Simbad e Jason

Il simulatore ha un proprio flusso di controllo che chiama periodicamente *performBehavior()*. Tale flusso di controllo deve essere totalmente separato da quello di Jason-CArtAgO. Se l'agente vuole poter fare delle azioni, per come è fatto il simulatore, le deve poter fare nel `performBehavior()`. In questo paragrafo viene descritta l'integrazione tra Simbad e Jason. CArtAgO (Common ARTifact infrastrutture for Agents Open environment) [2] è un framework che consente la programmazione e l'esecuzione di ambienti virtuali per sistemi multi-agente; fornisce un'astrazione. Si basa sul meta-modello Agents&Artifacts per la modellazione e progettazione di sistemi multi-agente. Gli agenti costituiscono parti del sistema che incapsulano l'attività autonoma all'interno di un certo ambiente, mentre gli artefatti sono strumenti tramite cui si articolano le attività degli agenti; sono entità computazionali che hanno come scopo l'essere usati dagli agenti. L'artefatto non è progettato per avere un suo corso delle azioni. Sono dei mediatori di azioni, tra gli agenti e l'ambiente, tra agenti e altri agenti. Gli agenti non interagiscono direttamente con il sistema; interagiscono con gli artefatti.

L'artefatto viene implementato estendendo una opportuna classe (*Artifact*) e il metodo *init* rappresenta il costruttore della classe. A questa classe possono essere passati, eventualmente, dei parametri d'ingresso all'atto della creazione dell'artefatto, eseguendo l'azione *makeArtifact* in Jason. Gli artefatti permettono di eseguire:

- operazioni: azioni invocabili dall'agente Jason (l'azione è come se venisse invocata su quella porzione di ambiente rappresentata dall'artefatto);
- operazioni interne: operazioni non invocabili dagli agenti ma che un'operazione dell'artefatto può mandare in esecuzione per questioni legate alle funzionalità dell'artefatto stesso.

Inoltre, consentono di definire le così dette *proprietà osservabili*, ovvero proprietà che l'agente vuole tenere monitorate e dalle quali vuole ricevere informazioni. Le proprietà osservabili generalmente sono rappresentate da delle tuple con un funtore e uno o più argomenti di qualunque tipo.

Il robot possiede sensori e attuatori; l'idea è quella di mappare i sensori, e quindi le percezioni, attraverso le proprietà osservabili definibili con CArtAgO, e le azioni, e quindi gli attuatori, tramite le operazioni interne definibili con CArtAgO e invocabili da Jason. L'artefatto farà quindi da corpo, da mediazione tra il simulatore e l'agente vero, cioè l'agente Jason. Dovrà avere delle proprietà osservabili che riflettono le informazioni sensoriali che noi vogliamo che l'agente abbia. Di questi artefatti ce ne deve essere uno per ogni robot che si vuole mettere nell'ambiente. Inoltre, per l'interazione con la GUI, la quale permette la gestione della simulazione, verrà usato un altro artefatto, il quale incapsulerà le funzionalità dell'I/O. In particolare, le azioni che avvengono sulla GUI saranno mappate su operazioni dell'artefatto per la gestione degli eventi riguardanti la GUI stessa. Come si modellano azioni e percezioni è importantissimo. Devo quindi poter definire percezioni e azioni che il robot è in grado di fare. In Simbad manca totalmente il layer che individua le azioni che dialogano con gli attuatori, in quanto non è un vero simulatore di robot, cioè non possiede il modello del robot reale. Ricapitolando:

- le percezioni vengono mappate tramite le proprietà osservabili definite nell'artefatto;
- le azioni vengono modellate come operazioni offerte dall'artefatto;

Un altro aspetto importante sempre legato alla simulazione è il modello di esecuzione delle azioni, ovvero come il simulatore gestisce l'effetto sull'ambiente da parte di uno o più robot ad ogni step temporale. Il problema è come gestire la contemporaneità di queste azioni.

Esistono due modelli fondamentalmente:

1. Al tempo t della simulazione, si chiede a tutti i robot di eseguire le azioni, ma quando uno esegue l'azione, modifica il mondo, perciò, quando si dà la parola ad un altro robot, questi vede il mondo non dell'istante t , ma dell'istante $t+n$, cioè il mondo nuovo, modificato. Il mondo cambia all'esecuzione delle azioni di ogni robot.
2. Al tempo t dà la parola a tutti gli agenti che propongono l'azione che vogliono fare, ma l'azione sul mondo non viene fatta al momento della richiesta. Queste vengono collezionate e, per fairness, vengono eseguite solo dopo che è stato chiesto a tutti di fare le azioni. Ad esempio: vi sono due robot, R1 e R2, i quali chiedono entrambi di spostarsi nel medesimo punto P; alla fine, solo uno dei due vi riesce. In Simbad, considerate le primitive messe a disposizione dal simulatore, che possono essere trovate consultando la documentazione, viene adottato il primo modello.

Quindi, quando viene invocato `performBehavior`, deve esserci solo un agente a eseguirlo.

Si deve considerare il fatto che l'agente Jason potrebbe eseguire azioni e percezioni in un qualunque momento, mentre qui, è necessario che questo venga fatto solamente quando viene invocata la `performBehaviour`. L'idea è che in fase di `performBehavior` si leggono le

informazioni dai sensori, si aggiornano le proprietà osservabili, si raccolgono anche le azioni che l'agente vorrebbe fare e le si fa eseguire. I casi sono due:

- si mette in campo una coda di azioni: ad ogni `performBehavior()` viene presa la prima azione dalla coda; un'idea alternativa eventualmente è fare seguire tutte le azioni presenti nella coda fino a quel momento;
- si sincronizzano le due parti: quando si esegue un'azione lato agente, ci si blocca in attesa della `performBehavior` per poterla eseguire;

Potrebbe anche capitare che lo stato dei sensori non cambi.

4.0.5 Implementazione dell'integrazione Simbad + Jason + CAr-tAgO

L'integrazione fra l'ambiente Jason e un generico simulatore può avvenire in diversi modi:

- tramite delle API che mette a disposizione il simulatore si ha la possibilità di controllare gli attuatori del robot e ricevere i valori dei sensori. L'interazione avviene tramite codice, ed è necessario programmare nello stesso linguaggio con cui sono disponibili le API del simulatore (o in alternativa usare la JNI, l'interfaccia nativa di java per comunicare con altri linguaggi).
- tramite connessione di rete TCP/IP (se il simulatore lo permette). Si apre una socket (incapsulata in un opportuno artefatto) che si connette al simulatore e in particolare al robot. Il collo di bottiglia è che ogni interazione deve passare dalla rete e quindi questo approccio può portare ad avere un controller meno reattivo.

A fronte delle considerazioni fatte nei paragrafi precedenti e considerato il fatto che il simulatore Simbad è scritto in Java, l'integrazione è stata implementata come segue. Lato CArAgO vengono creati due artefatti:

- *SimbadSimulator*: ha il controllo della simulazione (tramite GUI) e permette il setup delle istanze necessarie alla simulazione, come la creazione dell'ambiente e dell'agente Simbad. Qui sono definite sia le operazioni invocabili dall'agente Jason (definite tramite l'annotazione `@OPERATION`), che le operazioni interne (definite tramite l'annotazione `@INTERNALOPERATION`). Si ricorda che le operazioni interne non sono invece invocabili dall'agente Jason.
- *RobotBodyArtifact*: rappresenta il corpo del robot; qui vengono definite quali sono le percezioni che il robot deve avere, vengono inizializzate (una sola volta, all'atto della configurazione/creazione del robot); vengono lette e, se cambiate, aggiornate le informazioni sensoriali; vengono elencate l'insieme di azioni che può fare il robot. *RobotBodyArtifact* è un nome simbolico; può essere scelto qualunque nome e, per ogni robot che si vuole avere nell'ambiente, è necessario avere uno di questi artefatti.

Unitamente agli artefatti di cui sopra, sono state implementate altre classi Java molto importanti, in particolare:

- *SensorData*: struttura dati importantissima, modificabile a piacere, contiene la mappatura di tutti i dati sensoriali, le informazioni che si vogliono mappare a livello di sensori. Tutto ciò che si vuole far percepire al robot, andrà definito in questa classe.
- *SensorDataBuffer*: rappresenta il bounded buffer all'interno del quale saranno inserite tutte le percezioni. L'idea è che quando il buffer è pieno, vengono sovrascritte le percezioni più vecchie. Questa classe è comunque personalizzabile con una politica diversa, ad esempio: se il buffer è pieno, le percezioni nuove vengono scartate.

Tra le classi da fornire per il corretto funzionamento ci deve essere ovviamente anche quella che rappresenta l'agente Simbad (per agente Simbad si intende agente nell'accezione robotica, ovvero di robot fisico che agisce sull'ambiente). Questa classe consente di:

- creare una istanza del robot Simbad;
- definire e istanziare i sensori che equipaggiano il robot;
- fornire i metodi per la lettura dei dati sensoriali
- implementare i metodi `initBehavior()` e `performBehavior()`;

Per quanto riguarda la lista di azioni che può eseguire il robot, è stato adottato il seguente approccio:

- viene creata una lista di azioni che sarà parametrizzata rispetto ad una classe che identifica le azioni che può fare il robot. L'idea è quella di utilizzare il pattern *Command*, mettendo a disposizione un metodo *exec*, che eseguirà tutte le azioni che il robot è in grado di fare.
- per ogni azione va creata una nuova classe che lo rappresenta, la quale estenderà la classe di cui al punto precedente utilizzata per la parametrizzazione. Al costruttore saranno passati tutti i parametri utili necessari ad eseguire l'azione. Questa classe implementerà poi il *exec()*, che al suo interno conterrà l'azione fatta a livello di agente Simbad;
- all'interno dell'artefatto che rappresenta il corpo del robot vengono aggiunte tante operazioni (notazione *@OPERATION*) quante sono le operazioni che il robot è in grado di fare;
- all'interno della classe che definisce l'agente Simbad, vengono definiti, in maniera analoga al punto precedente, tanti metodi quante sono le azioni che il robot è in grado di fare. Infatti, il metodo che sarà richiamato all'interno dell'operazione definita nell'artefatto, è proprio il corrispondente metodo definito nella classe rappresentante l'agente Simbad. All'interno di tale metodo, verrà inserita nella lista di azioni che può fare l'agente una istanza dell'azione che l'agente vuole eseguire.

L'architettura risulta quindi essere la seguente: c'è il simulatore che ha un suo flusso di controllo autonomo, che periodicamente chiama per ogni robot *PerformBehaviour()* (PB). Tale flusso di controllo deve essere totalmente separato da quello di CArtaGO. Il PB diventa il fulcro per fare tutto: esecuzione di tutte le azioni che l'agente vuole poter fare in quel momento (con relativo svuotamento della lista), percezione (o raccolta dei dati sensoriali). In sostanza, ad ogni PB vengono scaricate le azioni possibili e si aggiunge una nuova snapshot sullo stato dei sensori. Si ritiene importante fare presente che lato artefatto, questi si blocca in attesa che vi sia almeno un dato sensoriale percepito. In particolare, viene utilizzata una opportuna primitiva di CArtaGO, *await*, la quale permette la sospensione in attesa di ricevere nuove informazioni sullo stato dei sensori (si sospende in attesa che nel buffer ci sia almeno un dato). La sospensione è tale per cui è comunque possibile interagire con l'artefatto, ovvero invocare le operazioni offerte, altrimenti sarebbe impossibile interagirvi. Non appena sono disponibili informazioni, l'*await* viene sbloccata e si va a leggere lo stato dei sensori. Si è scelto di tener traccia sia dello stato attuale dei sensori, che di quello precedente, in modo che i valori vengano aggiornati solo quando cambiano. Lato artefatto quindi, preleva continuamente le informazioni dal buffer (quando disponibili); se non ci sono, l'artefatto si sospende. Il simulatore invece, è un produttore che non si ferma mai.

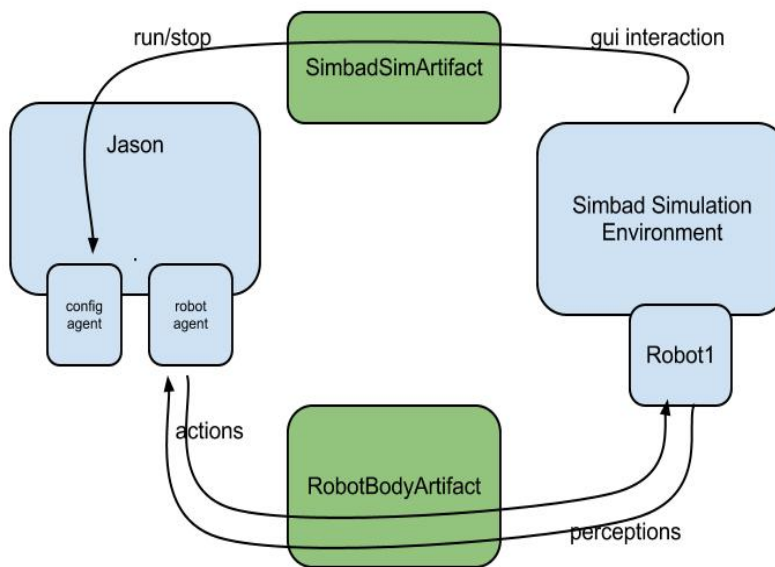


Figura 4.1: Architettura del sistema

C'è un aspetto delicato nell'integrazione Jason+Cartago+Simbad che va tenuto in considerazione: la possibile discrepanza fra i valori degli eventi e i belief nella Belief Base. Per capirne l'origine bisogna analizzare il ciclo BDI Jason. Le percezioni dell'agente vengono aggiornate dall'artefatto Cartago; quando c'è un cambiamento delle observable properties dell'artefatto

viene generato un “evento esterno” che viene inserito nella coda degli eventi esterni. Ad ogni ciclo di ragionamento, nella fase di sense, viene considerato un solo evento alla volta (che contiene l’aggiornamento di una o più prop. osservabili) e da questo vengono generati gli eventi interni. Questi eventi vengono inseriti nell’Event Set per essere prima o poi processati. Possono quindi crearsi situazioni in cui la belief base contiene valori di observable properties più aggiornati rispetto ai valori che recupero nel triggering event, quindi se si vuole lavorare sui belief più aggiornati conviene sempre recuperare, tramite test goal (?bel(Val)), il belief dalla BB invece di usare quello del triggering event (che potrebbe riferirsi a un valore passato e già vecchio).

Capitolo 5

Sperimentazione

Sono stati implementati diversi scenari; in un primo momento sono stati ricreati gli scenari proposti da Simbad, e poi sono stati proposti alcuni scenari di difficoltà incrementale. L'idea della sperimentazione è quella di vedere cosa vuol dire affrontare problemi progettando soluzioni con un approccio ad agenti, nello specifico adottando una architettura BDI e il linguaggio Jason.

5.1 Scenario 1: Walls

L'ambiente è una stanza rettangolare limitata da dei muri. Il robot inizialmente si avvicina al muro e poi lo costeggia in modo da mantenere sempre una distanza costante dal muro dall'inizio alla fine. Lo scopo del robot è percorrere tutto il corridoio:

- senza sbattere
- senza allontanarsi troppo dal muro

Il robot possiede una cintura di sensori a ultrasuoni, equamente distribuiti su a 360° tutto intorno al suo corpo. Con questi sensori il robot è in grado di percepire ostacoli (in questo caso i muri). I sensori restituiscono un valore numerico che indica la distanza del sensore rispetto a un ostacolo; se l'ostacolo non è presente o è più distante della soglia di percezione del sensore, questo restituisce distanza infinita. Tra gli attuatori il robot possiede dei comandi per controllare la velocità traslazionale e la velocità rotazionale.

In questo scenario si vuole applicare una particolare strategia di moto per il robot. Si è preso un scenario descritto in [6] e l'algoritmo seguente è stato riproposto in linguaggio Jason:

```
if distance-to-wall < range
    turn away from wall
else
    turn to the wall
```

ecco come l'algoritmo appare nell'agente (nel momento in cui esegue questi piani il robot si sta già muovendo di una certa velocità traslazionale):

```

@turn_away
+sonar(8, Value) : Value < 1.5
  <-
    setRotationalVelocity(1)
  .

```

```

@turn_towards
+sonar(8, Value)
  <-
    setRotationalVelocity(-1)
  .

```

I piani Jason ottenuti sono piani semplici, che ricalcano l'algoritmo precedente. Considerazioni: i piani scattano al variare dei belief, che vengono percepiti dall'agente tramite l'artefatto CArtAgO. Gli eventi di belief addition vengono creati non appena l'artefatto percepisce un cambiamento dei valori dei sensori del robot. Si tratta di un comportamento puramente reattivo e non un comportamento basato su goal dichiarativi. Inoltre i piani sono direttamente legati ai belief “grezzi” percepiti dai sensori. Nel caso del nostro framework Jason+CArtAgO+Simbad i belief dell'agente relativi ai valori dei sensori vengono aggiornati continuamente dall'artefatto CArtAgO e sono sempre presenti. In generale però i belief percepiti dall'ambiente potrebbero essere da elaborare o potrebbero a certo punto non essere più percepiti: in [6] viene fatto l'esempio del robot che percepisce il numero di birre solo quando apre il frigo. Quindi potrebbe convenire trasformare le informazioni ottenute dall'ambiente in rappresentazione simbolica consumabile dall'agente (mental notes), tramite opportuni piani. In questo modo si hanno due tipologie di piani: i piani che elaborano le informazioni dei sensori e i piani che si basano su queste informazioni elaborate per conseguire dei goal.

5.2 Scenario 2: Walls and cooling function

Nello scenario precedente il robot oscillava continuamente per mantenere la corretta distanza dal muro. Stavolta invece l'agente utilizza un *cooling factor* per effettuare il damping, ovvero una progressiva diminuzione delle oscillazioni. In questo modo il robot, dopo aver percorso una certa distanza, smette di oscillare e procede parallelo al muro. Di seguito i piani Jason:

```

@approaching_wall
+sonar(8, Value) : not(wall_near) & range(R) & Value < R
  <-
    .print("wall is near!");
  +wall_near
  .

```

```

@turn_away [atomic]
+sonar(8, Value) : wall_near & range(R) & Value < R

```

```

    <-
    ?vel(C);
    V = C * 0.9; //cooling factor
    -+vel(V);
    setRotationalVelocity(V) //rotate in the opposite direction
    .

    @turn_toward [atomic]
+sonar(8, Value) : wall_near
    <-
    ?vel(C);
    V = C * 0.9; //cooling factor
    -+vel(V);
    setRotationalVelocity(-V) //rotate in the opposite direction
    .

```

5.3 Scenario 3: Walls and noise

E' lo stesso scenario di quello precedente, ma nei sensori del robot è stato introdotto del rumore. Il rumore consiste nell'aggiungere un delta casuale (con distribuzione normale di probabilità) al valore percepito da un sensore. L'agente implementa l'algoritmo più semplice (quello senza damping) e il robot resiste al rumore introdotto nei sensori, nel senso che riesce a mantenersi più o meno a una certa distanza dal muro.

5.4 Scenario 4: Walls, cooling function and noise

In questo scenario il robot si muove con l'algoritmo di damping e si vede che, a causa del rumore, il movimento non riesce a convergere in una traiettoria parallela al muro.

5.5 Scenario 5: Following the light

In una stanza con degli ostacoli limitata da dei muri è presente una luce. Nella stanza è inoltre presente un robot dotato di sensori di luminosità che si muove seguendo questa luce. Con il passare del tempo questa prima luce si spegne e ne viene accesa un'altra, e così via per tre volte. Di volta in volta il robot si volta per seguire la nuova sorgente luminosa. Gli ostacoli influenzano la luminosità percepita dai sensori del robot. Il task del robot è quindi quello di seguire la sorgente luminosa.

Il robot possiede due sensori di luminosità posti sulla parte frontale del robot. Il robot possiede dei comandi per controllare la velocità traslazionale e la velocità rotazionale. Anche qui si vuole applicare una particolare strategia di moto per il robot.

5.6 Scenario 6: Pickup cherry Robot

In questo scenario è presente un solo robot nell'ambiente. L'ambiente è costituito da un box che circoscrive un'area sulla quale vengono poste degli oggetti. Questi oggetti rappresentano le ciliegie che il robot dovrà cercare e mangiare. Il task viene considerato completato quando il robot avrà raccolto e mangiato 5 ciliegie. Il robot si muoverà nell'ambiente in maniera casuale, ovvero non vengono implementati algoritmi di Coverage per l'ottimizzazione dell'esplorazione. L'idea iniziale era la seguente behavior nell'avere il robot così equipaggiato:

- sonar: per individuare le ciliegie;
- bumper: per rendersi conto di essere sufficientemente vicino alla ciliegia in modo da poterla mangiare.

Questo scenario è stato ripreso da uno dei degli scenari di esempio forniti con il simulatore SImbad. A causa di limitazioni tecniche di SImbad, però, il robot non è in grado di percepire le ciliegie tramite i suoi sensori (le ciliegie vengono implementate in modo tale che non vengono prese in considerazione nel calcolo delle collisioni/distanze percepite); questo rende impossibile la percezione della ciliegia come oggetto fisico. In questo scenario sono state quindi fatte alcune ipotesi aggiuntive (non realistiche) per emulare la percezione del rilevamento e del tocco della ciliegia.

Il robot possiede le seguenti percezioni:

- odometry: mappa l'odometro del robot; il robot è capace di rendersi conto di quante distanza ha percorso (questa misura è affetta da errori, che si propagano e aumenta mano a mano che la distanza viene coperta);
- trans_vel: velocità di traslazione: il robot è capace di percepire la velocità (in m/s) alla quale si sposta in linea retta;
- rot_vel: velocità di rotazione: il robot è capace di percepire la velocità (in rad/s) alla quale riesce a ruotare;
- counter: rappresenta lo scorrere del tempo dal punto di vista del robot (si ricorda che SImbad non ha il concetto di tempo logico);
- proximity: emula la percezione che si avrebbe da un sonar o da un bumper; questa percezione viene mappata su una funzione messa a disposizione da SImbad, che permette di capire se si è vicini ad una ciliegia oppure no.

Azioni che può compiere l'agente:

- setTranslationalVelocity: imposta la velocità alla quale il robot si sposta in linea retta;
- setRotationalVelocity: imposta la velocità rotazionale del robot;
- rotateY: implementa una rotazione intorno all'asse (rotazione geometrica);

- detach: permette al robot di mangiare la ciliegia (consiste nella rimozione dell'oggetto dalla scena);

L'idea è che: il robot comincia a muoversi nell'ambiente ad una velocità di traslazione costante. Ogni qualvolta riceve una percezione (con esito positivo) di prossimità ad una ciliegia, allora il robot incrementa un contatore personale delle ciliegie mangiate. Ogni qualvolta il counter raggiunge un multiplo di 80 invece, viene impostata una nuova velocità rotazionale, in modo da creare un andamento causale del robot.

Questo esempio è molto semplice e mette già in luce diversi limiti imposti dal simulatore, che hanno però contribuito a spingerci a tentare di implementare azioni più simile alla realtà, o comunque più vicine alle azioni compiute da un robot reale.

5.7 Scenario 7: Maze Robot

In questo scenario il Robot viene collocato all'interno di un labirinto. Il labirinto costituisce l'ambiente (*environment*) del robot e il suo task consiste nel trovare l'uscita dal labirinto. Per prima cosa è stato costruito il labirinto, traendo ispirazione da un'immagine reperita in rete per documentarsi su di essi e sulle strategie risolutive [10]. In questo tipo di scenario il robot possiede il seguente equipaggiamento:

- sonar: sono quattro: frontale, posteriore, laterale destro e laterale sinistro. Tramite i sonar laterali è possibile percepire la distanza dalle pareti del labirinto;
- bumper: un bumper posizionato frontalmente;

Sia i sonar che il bumper vengono mappati come proprietà osservabili, cioè come percezioni del robot; oltre a queste, tra le proprietà osservabili vi sono:

- counter: rappresenta lo scorrere del tempo dal punto di vista del robot
- trans_vel: velocità di traslazione: il robot è capace di percepire la velocità (in m/s) alla quale si sposta in linea retta;
- rot_vel: velocità di rotazione: il robot è capace di percepire la velocità (in rad/s) alla quale riesce a ruotare;
- odometer: è l'odometro: tiene conto della distanza percorsa dal robot;

Si fa l'ipotesi che il robot non ha alcuna conoscenza dell'ambiente, ovvero che non abbia alcuna rappresentazione del mondo che lo circonda. Le ipotesi che vengono fatte influenzano in parte la strategia risolutiva, perché la conoscenza del robot incide sul tipo di approccio risolutivo che egli può adottare. Infatti, per implementare alcune strategie risolutive il robot deve aver alcune conoscenze o, se non le ha, deve poter essere in grado di acquisirle. Esistono diverse strategie risolutive per uscire da un labirinto, alcune delle quali sono descritte in [14]. Sempre utilizzando l'approccio a difficoltà incrementale, si è scelto di adottare come

strategia risolutiva quella nota con il nome di *Wall follower*. Questa tecnica risolutiva è nota anche come *Left-hand rule* (LHR) e *Right-hand rule* (RHR) a seconda di quale sia il verso di percorrenza del labirinto. Se il labirinto è semplicemente connesso, ovvero tutte le pareti sono collegate insieme o collegate alla pareti che delimitano il labirinto, prima o poi il robot riuscirà ad uscirne (se è presente almeno una uscita; in caso contrario raggiungerà nuovamente l'ingresso). Il labirinto scelto in realtà, non gode di questa proprietà, poiché non è semplicemente connesso. Vi sono infatti alcune pareti che, se raggiunte, non sono connesse né ad altre pareti, né alle pareti perimetrali; quindi quello che può succedere è che, se il robot raggiunge una di queste pareti, può entrare in un loop infinito in cui percorre a vita le pareti raggiunte, senza mai arrivare all'uscita (o tornare all'ingresso). L'altra ipotesi aggiuntiva fatta (che contribuisce a semplificare lo scenario, anche in vista della successiva estensione) è la seguente: il robot conosce indicativamente quanto sono lunghi i due percorsi che può fare per arrivare all'uscita. Per conoscere indicativamente quanto sono lunghi i due percorsi, sono state effettuate diverse prove sia con la LHR che con la RHR, e grazie all'interfaccia grafica fornita da Simbad è stato possibile trovare i valori indicativi sfruttando l'odometro. Un'altra semplificazione (anch'essa oggetto di estensione nello scenario successivo), è che il robot sa a priori che se deve applicare la RHR, deve cominciare a muoversi senza ruotare; mentre se deve applicare la LHR, sa che dovrà ruotare di 180 gradi.

L'idea è la seguente: il robot sceglie in maniera casuale quale dei due approcci adottare (LHR o RHR); una volta scelto l'approccio, si impegna a raggiungere il proprio obiettivo. Per fare ciò, valuta continuamente i dati sensoriali. Per applicare una delle due regole, il robot deve andare alla ricerca del muro da seguire, sapendo che inizialmente egli si trova in uno "stato" di *start* (ovvero all'inizio del labirinto). A partire da tale stato, il muro viene considerato trovato se la distanza da esso è inferiore/uguale 0.20 . Una volta trovato il muro, il robot memorizza il fatto che l'ultimo muro che ha "visto" è stato a sinistra (destra) in base al sonar che ha rilevato la distanza inferiore alla soglia; inoltre, aggiorna lo "stato" ad *along-wall* per indicare che si trova lungo il labirinto. Sapendo da quale parte ha visto l'ultimo muro, saprà di quanto deve ruotare per poter proseguire il cammino verso l'uscita:

- ultimo muro visto a sinistra: rotazione a destra (ruota di -90°);
- ultimo muro visto a destra: rotazione a sinistra (ruota di $+90^\circ$);

Le azioni che può svolgere il robot sono le seguenti:

- `setTranslationalVelocity`: imposta la velocità alla quale il robot si sposta in linea retta;
- `setRotationalVelocity`: imposta la velocità rotazionale del robot;
- `rotateY`: implementa una rotazione intorno all'asse (rotazione geometrica);

Vi sono due questioni delicate nell'adempito del task assegnato al robot: una riguarda proprio la rotazione di cui sopra, che in questa prima implementazione è una pura rotazione geometrica intorno all'asse; l'altra riguarda i casi in cui il robot non deve eseguire solamente una rotazione per continuare a seguire il muro, ma deve, parlando in parole povere, "svoltare

l'angolo. Nel caso in cui il robot è costretto a svoltare l'angolo, vi è un momento in cui non percepirà più il muro al suo fianco (pur non essendo arrivato ancora in fondo al percorso). Per discriminare il caso in cui deve svoltare dal caso in cui ha terminato il suo task, si sfrutta la conoscenza iniziale del robot, e cioè la lunghezza indicativa del percorso. Il robot è dotato di odometro, perciò indicativamente sa quanta distanza dall'inizio del labirinto ha già percorso. Si distinguono tre casi:

1. il robot sta applicando la LHR: per svoltare l'angolo dovrà effettuare delle svolte a sinistra; se muovendosi il robot non rileva più il muro al suo fianco e se la distanza percorsa attualmente è inferiore alla distanza totale del percorso (ovvero il robot sa di essere ancora in mezzo al labirinto), allora il robot arresta il movimento traslatorio e si prefiggerà l'obiettivo di percorrere una breve distanza, percorsa la quale potrà ruotare di $+90^\circ$, per poi riprendere a muoversi. Utilizzando questo approccio, il robot adotterà la medesima strategia per girare l'angolo ogni volta che sarà necessario.
2. il robot sta applicando la RHR: per svoltare l'angolo dovrà effettuare delle svolte a destra; se muovendosi il robot non rileva più il muro al suo fianco e se la distanza percorsa attualmente è inferiore alla distanza totale del percorso, allora il robot arresta il movimento traslatorio e si prefiggerà l'obiettivo di percorrere una breve distanza, percorsa la quale potrà ruotare di -90° , per poi riprendere a muoversi. Utilizzando questo approccio, il robot adotterà la medesima strategia per girare l'angolo ogni volta che sarà necessario.
3. il robot, prescindendo dalla regola che sta applicando, non rileva più il muro al suo fianco e ha percorso una distanza maggiore/uguale a quella del percorso totale: in tal caso il robot ha completato il suo task e può arrestare il movimento, aggiornando anche lo "stato" corrente a *end* (fine del labirinto).

Per adempiere al task, sono stati creati opportuni piani, ciascuno dei quali identifica un possibile scenario in cui il robot può trovarsi.

Per i valori utilizzati nei vari scenari descritti sono state eseguite diverse prove in modo da cercare di adottare il valore che veniva reputato più appropriato. L'uso dell'odometro non esenta il robot dal commettere errori, anzi, come si vedrà nello scenario successivo esso diventerà fonte primaria di errori. Anche i valori ritenuti più opportuni sono comunque qualitativi, dati dall'esperienza accumulata nelle diverse prove effettuate.

5.8 Scenario 8: Maze Robot with Blind Robot

Questo scenario è una evoluzione dello scenario precedente. Sono stati introdotti alcuni elementi:

1. rotazione e spostamento visti come piani, quindi come sequenza di azioni che il robot deve adempiere per poter ruotare dell'angolo desiderato;

2. un secondo robot, in modo da poter sperimentare la comunicazione tra agenti;

In questo scenario l'idea è la seguente: vi sono due robot; il robot dello scenario precedente, ha le medesime percezioni e azioni e percorre il labirinto seguendo la LHR. Oltre a percorrere il labirinto, memorizza tutti i passi che fa per arrivare alla fine, ovvero costruisce quella che viene chiamata *mappa (o percorso) odometrico*. Questo lo fa memorizzando il numero del passo, il tipo di passo (se un movimento o una rotazione) e, nel caso di movimento, quanto ha percorso, mentre nel caso di rotazione, dove deve ruotare (sinistra o destra). Giunto alla fine del labirinto, il robot invia un messaggio al secondo robot. Il secondo robot è completamente "cieco", nel senso che non possiede alcuno sensore di tipo sonar o bumper, possiede solamente l'odometro ed è in grado di eseguire le medesime azioni dell'altro robot. Il robot che ha memorizzato il percorso fatto per arrivare alla fine del labirinto, invia al robot cieco un messaggio con il quale gli richiede se è pronto; a questo messaggio si replica con una risposta affermativa, a fronte della quale il robot invia il primo passo da effettuare. Il robot cieco, una volta ricevuto il messaggio, esegue quanto comunicato, spostandosi o ruotanti delle quantità indicate. Ogni volta che il robot cieco ha eseguito un passo, invia al robot un messaggio di acknowledge (è stato implementando un protocollo di comunicazione ispirato al TCP/IP, ovvero il robot cieco, compiuto un passo, invia all'altro robot un ack in cui specifica l'indice del passo che si aspetta di eseguire). Se non vi sono errori, o meglio, se l'errore commesso è molto piccolo, anche il blind robot giunge all'uscita del labirinto con successo.

Per quanto riguarda la rotazione e lo spostamento, sono stati usati le percezioni derivanti dall'odometro e dal tempo del robot. In particolare:

- rotazione: per effettuare questa azione, il robot si ferma, imposta una velocità di rotazione, attende per un tempo uguale al tempo che impiegherebbe a fare la rotazione e, una volta ruotato, imposta a zero la velocità di rotazione. L'idea era quella di usare le leggi della fisica, cioè: sapendo che il robot si muove ad una certa velocità costante di un certo numero di rad/s, e vuole coprire un certo angolo in radianti, calcoliamo quanto tempo egli impiega a coprirlo. Quindi, fornita la velocità e l'angolo di rotazione, il robot deve proseguire nella rotazione fino a che non è trascorso il tempo computato. Così facendo, non si ha la mera rotazione geometrica intorno all'asse, ma una rotazione molto più simile alla realtà.
- movimento: spostarsi in avanti (o indietro) di una certa quantità, è stato progettato con il seguente principio: il robot tramite l'odometro, prima di fermarsi, memorizza quanta distanza ha percorso sino a quel momento; nota la distanza che deve coprire, comincia a muoversi, fino a che la differenza tra il valore corrente dell'odometro e il valore precedentemente memorizzato non è maggiore/uguale alla distanza da coprire.

Utilizzando un approccio molto più simile alla realtà, si hanno anche degli effetti molto più simile alla realtà. Infatti, confrontando quanto avviene nello scenario precedente con quanto accade qui si può vedere che: con la rotazione reale, la rotazione non è perfetta, in quanto il robot può ruotare un po' di più (o un po' di meno) dell'angolo specificato. Sempre

con l'approccio reale, il robot è più portato a rimanere bloccato vicino a muro, durante la fase di rotazione, a causa del corpo che funge da ingombro durante i movimenti. Questo, è molto più simile alla realtà come comportamento, rispetto al caso in cui si ha una rotazione geometrica, in cui non si fa uso né del tempo né dei dati derivanti dai sensori (l'odometro nel caso specifico).

Nella figura

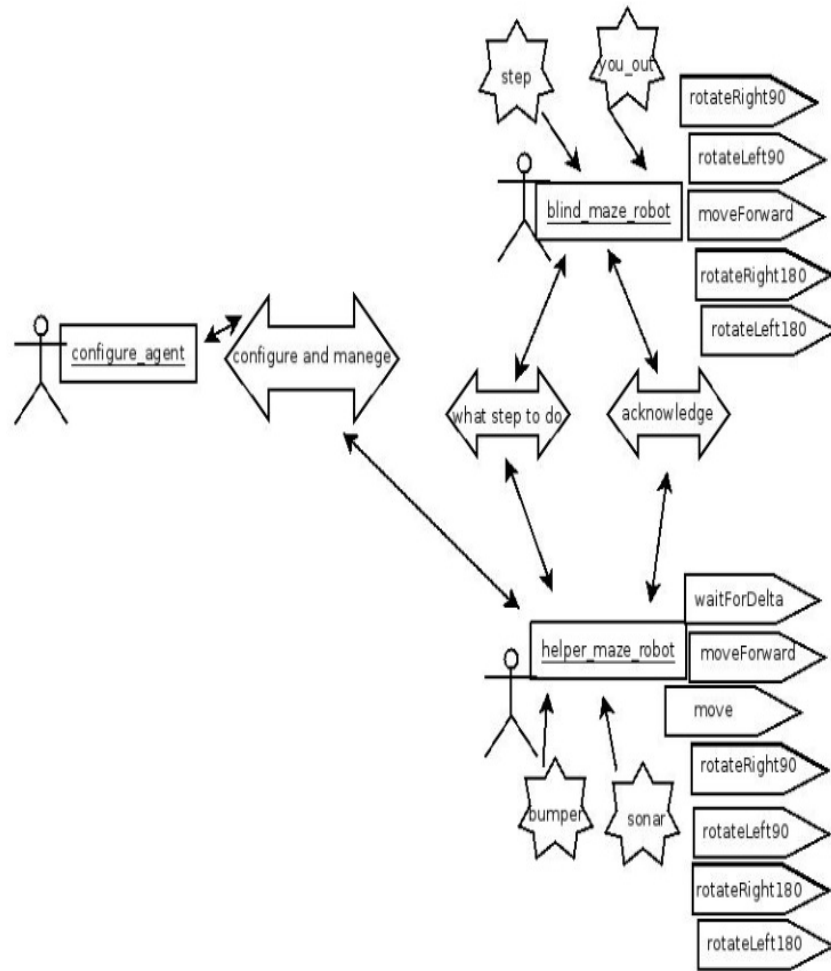


Figura 5.1: Sistema scenario labirinto

Capitolo 6

Simbad e i requisiti per la programmazione di robot cognitivi

Nel paper [7] vengono elencati quali sono i requisiti che devono avere i linguaggi di programmazione ad agenti per programmare i controller di robot cognitivi. Di seguito vengono elencati questi requisiti e si cerca di capire se il framework Simbad li soddisfa oppure no. In generale, come si vedrà, molti dei requisiti non sono soddisfatti.

6.0.1 Integrazione/interoperabilità

Per facilitare il riuso e lo sviluppo di sistemi complessi sono stati sviluppati dei framework per garantire l'interoperabilità dei vari moduli. Uno di questi è ROS (Robot Operating System), che è ormai diventato uno standard. Jason non ha ancora un supporto a questo framework, ma dovrebbe essere possibile aggiungerlo, possibilmente sfruttando CArtaGO, allo stesso modo come lo si è fatto per integrarlo con il simulatore Simbad. Lato ROS, è da poco disponibile una prima implementazione di ROS in java, seppure in versione alfa [15].

6.0.2 RealTime

Un robot deve poter dare delle garanzie di real time agli eventi/azioni per reagire a un ambiente dinamico come è quello reale. Due vincoli di reattività realtime sono il reaction time e il response time. Il reaction time è il ritardo fra l'istante in cui un evento viene ricevuto dal sistema e l'istante in cui questo evento viene preso in considerazione, mentre il response time è il ritardo fra l'istante di ricezione dell'evento e l'istante in cui il piano generato per gestire l'evento è stato completamente eseguito. Fissare un upper bound per questi valori significa fissare dei vincoli realtime. In [15] viene ipotizzato che il modo migliore per gestire vincoli RT sia di avere un sistema di controllo BDI distribuito costituito da più componenti BDI, e solo alcuni di questi in grado di fornire garanzie RT. Simbad invece non fornisce alcuna garanzia RT: le percezioni del robot vengono recuperate dall'artefatto e messe in una coda su cui non si hanno garanzie temporali, poiché i due flussi di controllo (quello di Jason e quello di Simbad) sono indipendenti. Allo stesso modo le azioni che Jason invia

a Simbad finiscono in una coda gestita dall'artefatto ma non si hanno garanzie temporali sull'esecuzione di queste.

6.0.3 Gestione della sensoristica

Un linguaggio per il controllo robotico dovrebbe permettere una gestione avanzata degli input da sensori. Le funzionalità richieste comprendono:

- garbage collection degli eventi obsoleti;
- rappresentazione della conoscenza sufficientemente espressiva in modo da avere una rappresentazione unificata dei vari input;
- capacità di reasoning in modo da riconoscere e gestire situazioni di alto livello;
- gestione di priorità degli eventi;
- rappresentazione simbolica ad alto livello;

Consideriamo ora Simbad: le percezioni del robot, tramite l'artefatto, vanno a modificare la Belief Base (BB) dell'agente e aggiungono degli eventi. Questi eventi finiscono nell'event set e prima o poi vengono gestiti (ricordiamo che si è lasciata gestione di default, ovvero coda FIFO). Di conseguenza non si perdono mai eventi e quindi non avviene neanche garbage collection. Jason consente di modificare il suo ciclo di ragionamento in modo da gestire la priorità degli eventi, anche se in questo framework non è stata sfruttata questa possibilità. Queste modifiche vanno fatte in modo oculato per evitare situazioni di event starvation. Le percezioni del robot diventano belief nell'agente e quindi sono in forma simbolica, ma non ad alto livello. Sta al programmatore realizzare dei piani che trasformano i valori "grezzi" dei sensori in informazioni interessanti. Rappresentazione/esecuzione di piani complessi Sono richieste funzionalità come

- la composizione di piani complessi a partire da sottopiani (decomposizione gerarchica);
- supporto di vincoli funzionali e temporali che si presentano fra i vari task del robot;
- monitoring/controllo di esecuzione ai diversi livelli della gerarchia dei piani;
- supporto di subgoal in grado di sospendere oppure no l'intenzione dai quali sono stati generati;
- priorità nell'esecuzione delle intenzioni;
- supporto di azioni/intenzioni atomiche;

L'unico supporto che ha Jason per programmazione modulare dei piani è tramite l'invocazione di subgoal all'interno del piano. Questi subgoal possono essere bloccanti (!g1) o non bloccanti (!!g1) e generano degli eventi che finiscono nell'event set. Se il subgoal è bloccante l'intenzione che l'ha generato potrà proseguire solo dopo che il subgoal (e quindi l'intenzione che è stata scelta per gestirlo) è terminato.

Jason fornisce degli strumenti per il controllo e manipolazione dei piani/intenzioni a diversi livelli, ad esempio fornisce le internal action per far fallire o compiere delle intenzioni non completamente portate a termine, e il cui goal matcha con un certo literal (internal action `.succeed_goal` o `.fail_goal`). Modificando il ciclo di ragionamento (in particolare il metodo `selectIntention`) si può modificare la priorità delle intenzioni. Jason consente l'utilizzo dell'annotazione `[atomic]` per eseguire atomicamente un piano. Riguardo al supporto per esprimere vincoli funzionali fra diversi piani viene fornita ad esempio l'internal action `.await` che permette di bloccare un piano in attesa che un belief compaia nella BB. In questo modo posso esprimere in maniera indiretta (ovvero tramite dei belief) la necessità che un piano dipenda da un altro piano per proseguire.

Capitolo 7

Conclusioni e sviluppi futuri

Simbad si è rivelato essere molto semplice da usare e anche facilmente integrabile con i tool che sono stati usati per le sperimentazioni (Jason e CArtAgO). E' utile come punto di partenza per provare a sperimentare cosa vuol dire programmare il controller di un robot, poiché quando lo si progetta, pur nelle sua semplicità d'uso, ci si scontra ugualmente con le problematiche reali dei robot fisici. Di contro, questa sua semplicità non permette di avere alcun controllo/accesso di ciò che si trova più a basso livello in un robot (come ad esempio i motori). Un altro punto a sfavore del simulatore è l'assenza di una separazione tra il tempo logico e quello fisico, che genera comportamenti errati nel robot stesso. Per quanto riguarda la scelta di utilizzare un linguaggio ad agenti basati su architettura BDI, Jason nello specifico, questi offre un modello che si presta bene alla modellazione e progettazione di controller di robot. Il fatto di avere un modello basato sullo human reasoning permette di progettare soluzioni relative ad un certo task in maniera intuitiva, da un certo punto di vista. Avere in mano strumenti che permettono di modellare la base di conoscenza, il poterla interrogare, modellare le percezioni e le azioni, è molto importante; questo perché quello che si tende a fare, dato un certo task da risolvere, è pensare ad una soluzione costituita da un controller che affronta il problema come lo affronteremmo noi se ci trovassimo a dover compiere lo stesso task. Quello che però manca attualmente, è la gestione della componente real time (che in alcuni scenari è una componente importante) e della possibilità di avere una priorità nella gestione degli eventi. Un possibile sviluppo futuro potrebbe essere quello di modificare/estendere il ciclo di ragionamento di Jason per definire una priorità nella schedulazione degli eventi (derivanti dalle percezioni avute dall'ambiente) per fare in modo che il robot possa tentare di correggere alcuni comportamenti anomali (o erratici) durante lo svolgimento del suo task. Questo tipo di estensione sarebbe però da sperimentare su un altro framework, più avanzato rispetto a Simbad, in modo da vedere come si comporta il medesimo controller (cioè lo stesso codice) messo in esecuzione su framework diversi. Un altro test interessante sarebbe vedere come si comportano controller che svolgono il medesimo task, ma scritti in linguaggi ad agenti diversi tra loro, e messi in esecuzione su framework diversi tra loro.

Bibliografia

- [1] Bullet. Game physics simulation.
<http://bulletphysics.org/>.
- [2] CArTAgO. Common artifact infrastructure for agents open environments.
<http://cartago.sourceforge.net>.
- [3] S. C. et al. Usarsim: a robot simulator for research and education. *In Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*, IEEE Press, Pis-cataway, NJ:1400–1405, 2007.
- [4] A. S. Jensen. *Implementing Lego Agents using Jason*. 2010.
- [5] N. B. Louis Huguens. Simbad: an autonomous robot simulation package for education and research. *From Animals To Animats - Lecture Note in Computer Science*, 4095/2006:831–842, 2006.
- [6] M. J. Mataric. *The Robotics primer*. Intelligent Robotics and Autonomous Agents series, Massachusetts Institute of Technology, 2007.
- [7] J.-J. M. Pouyan Ziafati, Mehdi Dastani and L. van der Torre. Agent programming languages requirements for programming cognitive robots. *Workshop on Programming Multiagent Systems*, International Conference on Autonomous Agents and Multiagent Systems:39– 54, June 2012, Valencia.
- [8] S. Project. Simbad project home.
<http://simbad.sourceforge.net/index.php>.
- [9] M. W. Rafael H. Bordini, Jomi Fred Hubner. *Programming Multi Agent Systems In AgentSpeak Using Jason*. Wiley Series in Agent Technology, Wiley, December 2007.
- [10] P. D. Reiners. Robots, mazes, and subsumption architecture.
<http://www.ibm.com/developerworks/java/library/j-robots/>.
- [11] A. Robotics. Nao robot.
[http://it.wikipedia.org/wiki/NAO_\(robot\)](http://it.wikipedia.org/wiki/NAO_(robot)).

- [12] ROS. Ros documentation.
<http://www.ros.org/wiki/>.
- [13] R. Smith. Ode open dynamics engine.
<http://www.ode.org>.
- [14] Wikipedia. Maze solving algorithm.
http://en.wikipedia.org/wiki/Maze_solving_algorithm.
- [15] J. Wright. Rosjava - java for robotics? semesa2.