UNIVERSITÀ
DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Ingegneria Informatica, delle Comunicazioni ed Elettronica

Elaborato finale

Prolog planner

Task planner logico per la manipolazioni di blocchi tramite un UR5

Supervisore
Prof. Luigi Palopoli

Laureando
Davide De Martini

Anno accademico 2022/2023

Ringraziamenti

A tutti

Indice

Sommario **2**

1 Introduzione **2**
1.1 Definizione del problema . . . 3
1.2 Obiettivi desiderati . . . 3
1.3 Struttura della tesi . . . 3

2 Stato dell'arte **4**
2.1 Prolog . . . 4
2.1.1 Struttura e sintassi . . . 5
2.1.2 Esecuzione . . . 6
2.1.3 Debugging . . . 7
2.2 Evoluzione di prolog . . . 7
2.2.1 Tappe fondamentali dell'evoluzione di Prolog . . . 8
2.3 Lavori notabili . . . 9

3 Descrizione del caso di studio **10**
3.1 Prima milestone: Studio di Prolog . . . 11
3.2 Seconda milestone: Sviluppo del planner . . . 12
3.2.1 Rappresentazione dei blocchi . . . 12
3.2.2 Azioni del planner . . . 13
3.2.3 Generatore del piano . . . 15

4 Descrizione dell'architettura ROS **16**
4.1 Introduzione a ROS . . . 16
4.1.1 Nodi . . . 17
4.1.2 Topics . . . 17
4.1.3 Messaggi . . . 17
4.1.4 ROS services e parameter server . . . 17
4.1.5 Esempio rete ROS . . . 18
4.2 Architettura utilizzata . . . 18
4.2.1 Terza milestone: Wrapping codice Prolog . . . 18
4.2.2 Quarta milestone: Simulazione . . . 19
4.3 Difficoltà incontrate . . . 21

5 Conclusione **22**

Bibliografia **22**

Sommario

Sommario è un breve riassunto del lavoro svolto dove si descrive l'obiettivo, l'oggetto della tesi, le metodologie e le tecniche usate, i dati elaborati e la spiegazione delle conclusioni alle quali siete arrivati.

Il sommario dell'elaborato consiste al massimo di 3 pagine e deve contenere le seguenti informazioni:

- contesto e motivazioni
- breve riassunto del problema affrontato
- tecniche utilizzate e/o sviluppate
- risultati raggiunti, sottolineando il contributo personale del laureando/a

1 Introduzione

In questo primo capitolo viene introdotto il caso di studio, partendo dalla definizione del problema, definendo poi gli obiettivi desiderati e concludendo con una spiegazione della struttura della tesi.

Per la mia tesi triennale mi sono voluto concentrare sul campo dell'intelligenza artificiale e la robotica. Il caso di studio identificato dal prof. Luigi Palopoli, che è stato poi oggetto del mio tirocinio e della mia tesi, è l'utilizzo di Prolog, un linguaggio logico e dichiarativo, per l'implementazione di un task planner ad alto livello. Questo deve essere in grado di coordinare un manipolatore robotico, nel nostro caso un UR5 della Universal Robots, per l'operazione di manipolazione di blocchi. Il planner tramite delle primitive come *ruota* e *muovi* deve essere in grado di pilotare il braccio robotico per la creazione di un pilastro. Esso è stato concepito ad alto livello per permettere la compatibilità con più tipologie di robot e la flessibilità delle applicazioni. Nel caso studiato verrà utilizzato per la creazione, appunto, di un pilastro, ma questa astrazione ad "alto livello" permette di essere usato per più compiti: ad esempio per la creazione di una costruzione come una casa o qualsiasi struttura che si può creare con dei blocchi di Lego. Queste primitive verrando inviate al nodo controllore del movimento che, tramite algoritmi di cinematica e interpolazione dei punti, troverà le configurazioni dei giunti adatte al movimento richiesto.

Successivamente si è voluto implementare il tutto in un ambiente simulato, così da avere un riscontro anche visivo del lavoro svolto e capire le possibili applicazioni in un contesto reale. La simulazione è stata svolta con ROS e Gazebo, due strumenti open source ampiamente utilizzati nel mondo della robotica. Il primo ci permette di comunicare al robot tramite una serie di topic e servizi, mentre il secondo ci permette di simulare l'ambiente in cui il robot si trova.

La scelta di questo caso di studio è nata principalmente dal mio interesse verso la robotica e le applicazioni dell'intelligenza artificiale in vari contesti. L'utilizzo di un linguaggio logico per la creazione di un task planner è stato un approccio nuovo che non avevo mai considerato e fin da subito mi è interessato. La rappresentazione della conoscienza in un robot tramite Prolog è un approccio molto interessante che può essere applicato in molti altri contesti. Un altro motivo che mi ha portato a scegliere questo caso di studio è la sua possibilità di espansione. Due possibili miglioramenti potrebbero essere la temporizzazione delle azioni e la successiva ottimizzazione del tempo di esecuzione finale (*makespan*) e l'utilizzo del machine learning per istanziare i fatti iniziali. Sapere che questo lavoro possa essere portato avanti e migliorato è sicuramente un altro motivo principale della sua scelta. Mi sono quindi rivolto al prof. Palopoli e lui mi ha proposto di sviluppare questo progetto. La

scelta di utilizzare Prolog è nata dal fatto che è un ottimo strumento per modellare la conoscenza in un agente. Esso permette di creare delle "basi di conoscenza", contenenti sia fatti che regole, su cui successivamente fare inferenza. L'esecuzione di un programma prolog è comparabile alla dimostrazione di un teorema mediante la regola di inferenza della soluzione. Tutte queste premesse, e l'apmpio utilizzo di prolog nel mondo dell'intelligenza artificiale, ci hanno incuriosito e portato allo sviluppo del progetto.

1.1 Definizione del problema

Il caso di studio presentato non è nuovo nel mondo dell'intelligenza artificiale. L'utilizzo di linguaggi logici per applicazioni di planning è, ed era, uno degli approcci preferiti dalla comunità per risolvere problemi in questo dominio. È difficile trovare delle applicazioni nella vita reale che utilizzano solamente Prolog, questo il più delle volte viene usato insieme ad altri strumenti o linguaggi di programmazione tipo Java.

Inizialmente la sfida è stata di prendere domestichezza con il linguaggio di programmazione. Essendo un paradigma che non avevo mai utilizzato e, abituato ai linguaggi imperativi come Python, Java, ho provato a usare un approcio simile in Prolog, fallendo. Questo perchè, al contrario di come siamo abituati, Prolog fa un uso massiccio della ricorsione e il concetto di variabile è differente da come siamo abituati. Presa domestichezza la sfida diventò quella di definire un modo efficace per rappresentare l'ambiente studiato. Dovevo riuscire a trovare un astrazione della realtà tale che, utilizzando una stringa, riuscisse a rappresentare tutte le informazioni necessarie sulle proprietà attuali del blocco di lego. Successivamente questa si è trasformata nel identificare le azioni base che il nostro planner doveva inviare al robot. Tutto ciò doveva essere svolto mantenendo consistenza tra il mondo modellato e la base di conoscenza, quindi evitare che un blocco che si è spostato nel mondo reale rimanga nella sua posizione precedente nella base di conoscenza.

1.2 Obiettivi desiderati

In sede di sviluppo del progetto abbiamo voluto specializzare il planner nella costruzione di pilastri di altezza scelta dall'utente. Questi sono composti dai *megablocks*, dei blocchi di lego di dimensione maggiorata, per facilitare la presa al nostro manipolatore. Nella base di conoscenza questi saranno rapressentati come dei fatti *ground*, ciò significa che il nostro interprete Prolog li assumerà per veri a priori. Il predicato che svolgerà la creazione del pilastro dovrà restituire tutte le possibili soluzioni che soddisfano il nostro problema. Successivamente una serie di azioni saranno calcolate per la costruzione del pilastro e queste verranno inviate al robot. Esso è un UR5 del produttore Universal Robots, un manipolatore a 6 gradi di libertà. Alla sua estremità è presente un gripper a due dita parallele della Soft Robotics che permette la presa dei blocchi. Tramite il piano generato, il robot dovrà costruire il pilastro richiesto con i blocchi messi a disposizione. Riceverà quindi una serie di azioni ad alto livello e convertirà queste in movimenti veri e propri del braccio tramite il nodo di motion planning.

1.3 Struttura della tesi

Questa tesi inizierà presentando lo stato dell'arte delle applicazioni in Prolog, dando inizialmente una visone generale del linguaggio e la sua sintassi. Vedremo l'evoluzione del linguaggio negli anni e la sua applicazione in ambiti sia di ricerca che di industria. In seguito ci sarà un capitolo dedicato più dettagliatamente al caso di studio in cui il programma Prolog sarà spiegato più dettagliatamente. Successivamente andrò a spiegare cosa è ROS e gli strumenti che ho utilizzato per simulare il robot. In questa sezione descriverò l'architettura e i nodi creati per il progetto, soffermandomi anche su concetti di cinematica e di motion planning. Infine un capitolo sarà dedicato alle conclusioni e ai lavori futuri.

Iniziamo ora introducendo appunto lo stato dell'arte di prolog e le pubblicazioni notabili, andando

a dare un approfondimento anche al mondo dell'industria.

2 Stato dell'arte

In questo capitolo verranno presentati i principali lavori che compongono lo stato dell'arte di Prolog. In particolare verrà inizialmente introdotto il linguaggio, poi sarà presentata la storia di Prolog, partendo dalla creazione fino a come lo conosciamo ora. Infine verrà introdotta la sua applicazione nel mondo portando i lavori notabili, sviluppando il focus nel ambito dell'automazione e della robotica. Per la scrittura di questo capitolo ho utilizzato varie fonti. Esse saranno riportate a inizio di ogni sezione e in ogni parte in cui sono state utilizzate.

2.1 Prolog

In questa sezione verrà introdotto il linguaggio, dandogli una definizione e una descrizione della sua attuale implementazione. Per la scrittura di questa sezione ho utilizzato principalmente il libro *The art of Prolog* [7] e il lavoro *50 years of Prolog and beyond* [5].

Ad oggi Prolog è considerato il linguaggio di programmazione logica più utilizzato e più importante. La sua evoluzione, però, non è stata lineare. Negli anni ha subito diverse modifiche, la cosa curiosa è che, a differenza di altri linguaggi, queste venivano integrate nel progetto principale invece che creare dei linguaggi paralleli. Qui di seguito elenco i principi fondamentali che ogni implementazione di Prolog deve rispettare secondo lo studio di Körner et al. [5]:

1. Il linguaggio deve essere basato sulla logica dei predicati di primo ordine utilizzando le clausole di Horn. Sia per la creazione delle basi di conoscenza che per le interrogazioni delle basi di conoscenza (query).

2. Avere l'abilità di manipolare predicati e clausole come termini, così che i meta-predicati possano essere scritti come normali predicati.

3. Risoluzione SLD [6] basata al principio di risoluzione di Robinson.

4. Unificazione di termini arbitrari che includono variabili.

5. Esplorazione depth-first dell'albero di ricerca.

I punti fondamentali per un implementazione di Prolog sono quindi: l'uso delle clausole di Horn, la risoluzione SLD e l'unificazione di termini arbitrari. Ci sono altre importati funzioni che un implementazione dovrebbe supportare:

6. Negazione come fallimento e altri aspetti logici come la disgiunzione e l'implicazione.

7. Possibilità di alterare il contesto dell'esecuzione durante la risoluzione.

8. Possibilità di utilizzare la programmazione con *constraints*.

Queste sono quindi le caratteristiche che ogni implementazione di Prolog dovrebbe avere, se anche una dovesse mancare allora non si potrebbe considerare un implementazione di Prolog.

La *Regola del pollice* per capire se un risolutore logico può essere considerato un implementazione di Prolog e quindi rispettare i pilastri elencati sopra è la seguente. Il test richiede che la ben nota regola *append/3* possa essere scritta esattamente come segue

```
append([], X, X).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

e questa può essere interrogata in differenti modi: fare l'append tra due liste `append([1,2],[c,d], R)`, oppure decostruire una lista come `append(A,B,[1,2])` o infine istanziarla con argomenti arbitrari come `append([X|T],[c],[Z,Z,Z])`. Se la regola funziona senza anomalie, l'implementazione utilizzata può considerarsi un implementazione di Prolog.

Descriviamo ora più in dettaglio la struttura di Prolog e la sua sintassi.

2.1.1 Struttura e sintassi

Iniziamo definendo la tipologia di dati supportata da Prolog. Il dato in Prolog viene chiamato termine, esso può essere un atomo, un numero, una variabile oppure un termine composto.

- Atomo: è un termine che non è né un numero né una variabile. Gli atomi sono usati per rappresentare nomi di oggetti, relazioni o costanti.
- Numero: i numeri in Prolog possono essere sia interi che reali.
- Variabili: sono usate per rappresentare oggetti o valori sconosciuti. Le variabili in Prolog iniziano con una lettera maiuscola o con il carattere di sottolineatura.
- Termine composto: esso è composto da un atomo chiamato *funtore* e un numero di argomenti. Il numero di argomenti è chiamato *arità* del termine composto. Casi speciali di termini composti sono le liste e le stringhe.

L'utilizzo di questi dati ci permette di definire le *clausole di Horn* [4] che sono alla base del linguaggio. La sintassi del prolog è basata appunto sulla logica dei predicati di primo ordine, limitata però alle clausole di Horn. Queste sono delle disguinzioni di letterali in cui al massimo uno dei letterali è positivo. Un esempio è il seguente:

$$\neg umano(X) \vee mortale(X) \tag{2.1}$$

Questo significa che:

$$\forall X(\neg umano(X) \vee mortale(X)) \tag{2.2}$$

Utilizzando l'equivalenza logica:

$$\neg X \vee Y \equiv X \Rightarrow Y \tag{2.3}$$

Quindi:

$$\forall X(umano(X) \Rightarrow mortale(X)) \tag{2.4}$$

In questo esempio possiamo vedere come è costituita una clausola di Horn. Se la premessa (*umano*) è vera allora anche la conseguenza (*mortale*) è vera.

Le clausole possono essere senza testa:

```
umano(luca).
padre(livio, lorenzo).
```

Oppure con testa:

```
mortale(lorenzo) :- umano(lorenzo).
```

Come detto prima, la logica in Prolog è espressa in termini di relazioni tra fatti e regole, la verifica di queste relazioni prende il nome di *query* o *interrogazione*. Un fatto in Prolog può essere visto come una clausola con il corpo vuoto, ad esempio:

```
umano(luca).
```

In questo esempio vediamo come si può esprimere logicamente che luca sia un umano. Un fatto può essere visto anche come una regola che a priori è sempre vera:

```
umano(luca) :- true.
```

Una regola, invece, è una clausola completa di testa e corpo, la testa è vera solamente se anche il corpo è vero. Un esempio di regola è quello visto prima:

```
mortale(X) :- umano(X).
```

Il corpo di una regola è un insieme di predicati, questi possono essere congiunti o disgiunti. L'operatore di congiunzione è la virgola (`,`) mentre quello di disgiunzione è il punto e virgola (`;`). Prolog viene fornito di predicati predefiniti, spesso utilizzati per la manipolazione di liste, aritmetica, input/output. Esempi di questi sono *append*, *is*, *write*, *read*, ecc.

L'insieme di questi 'costrutti' compone l'interità della sintassi base in prolog. Ci sono anche altri operatori, uno dei più importanti è il seguente: Il predicato \+ /1 definisce la *negazione come fallimento*. Ciò permette a Prolog di essere un sistema di ragionamento non monolitico.

```
legale(X) :- \+ illegale(X).
```

Per definire degli algoritmi iterativi come quelli che siamo abituati a vedere in linguaggi imperativi, Prolog utilizza la ricorsione. Io stesso ne ho fatto largo uso per la mia base di conoscenza, un esempio è il seguente:

```
list_length([], 0).

list_length([_|T], N) :-
    list_length(T, N1),
    N is N1 + 1.
```

In questo esempio il predicato è utilizzato per calcolare la lunghezza di una lista, vediamo come ci sia il caso base (lista vuota) e il caso ricorsivo. In ogni caso al momento l'ho usato soltato come esempio, nel capitolo 3 spiegherò meglio la struttura del programma e come ho risolto i problemi incontrati in Prolog.

2.1.2 Esecuzione

L'esecuzione di un programma Prolog è basata sulla ricerca di un insieme di predicati che soddisfano una data query. Il metodo di risoluzione utilizzato è la risoluzione SLD [6] (Selection, Linearization, and Driving). Questo processo di inferenza consiste in più fasi:

1. Selezionare una clausola goal, essa rappresenta quello che si desidera dimostrare.
2. Unificare la clausola goal con quella del programma, quindi trovare un modo di rendere uguali i termini nelle due clausole.
3. Risolvere la clausola unificata, il programma userà quindi le sostituzioni fatte nella fase precedente per risolvere la clausola. Questo genererà nuove clausole che verranno aggiunte all'insieme di clausole da risolvere.
4. Ripetere i passaggi 2 e 3 fino a quando non si raggiunge un punto di terminazione. La strategia di scelta delle clausole è la selezione lineare più a sinistra.
5. Il processo termina quando viene raggiunto uno di questi casi:
    - Una clausola vuota viene generata dimostrando quindi che la query è vera.
    - Non è possibile selezionare ulteriori clausole per unificare e risolvere.
    - Viene raggiunto il limite di profondità o di tempo (prestabilito dall'utente).

Per esplorare tutte le possibili soluzioni di un problema, Prolog utilizza il *backtraking*. Se durante l'esecuzione di una regola o l'unificazione di un fatto si raggiunge un punto in cui non è più possibile trovare delle soluzioni, Prolog torna indietro (backtrack) per cercare altre possibilità. Questo processo quindi permette di riprendere l'esplorazione delle alternative non ancora considerate. Durante il processo di backtracking Prolog annulla tutte le assegnazioni fatte precedentemente e cerca altre alternative. Le variabili unificate vengono quindi scollegate per permettere la ricerca di altre soluzioni. Un esempio di questo strumento:

```
padre(giovanni, maria).
padre(giovanni, giuseppe).
madre(maria, francesca).

genitore(X, Y) :- padre(X, Y).
genitore(X, Y) :- madre(X, Y).
```

Se noi dovessimo eseguire la query *genitore(giovanni, X)* Prolog troverebbe due soluzioni: $X = maria$ e $X = giuseppe$. Facendo così Prolog ha esplorato tutte le alternative possibili dell'albero di ricerca.

2.1.3 Debugging

Prolog fornisce un insieme di strumenti per il debugging, uno di questi è il *trace*. Questo strumento permette di vedere l'esecuzione del programma passo passo. In questo modo è possibile vedere come Prolog risolve le query e quali clausole vengono selezionate. Nella console di SWI-Prolog è possibile attivare il trace con il comando *trace.* e disattivarlo con *notrace.*. Avviandolo ad ogni chiamata di un predicato verrà mostrato il suo nome e i suoi argomenti. Inoltre verrà mostrato il risultato della sua risoluzione. Questo renderà il debugging di un programma più semplice, e permetterà di capire meglio il funzionamento di Prolog.

2.2 Evoluzione di prolog

La sezione seguente si concentra sull'evoluzione del linguaggio Prolog nel corso degli anni. Sin dalla sua ideazione negli anni 70', Prolog ha subito notevoli sviluppi, adattamenti e miglioramenti per adeguarsi alle esigenze deglla comunità. Per la scrittura di questa sezione mi sono affidato all'articolo di Körner et al. [5], il lavoro *Logic programming for deliberative robotic task planning* [9] e altre fonti e articoli.

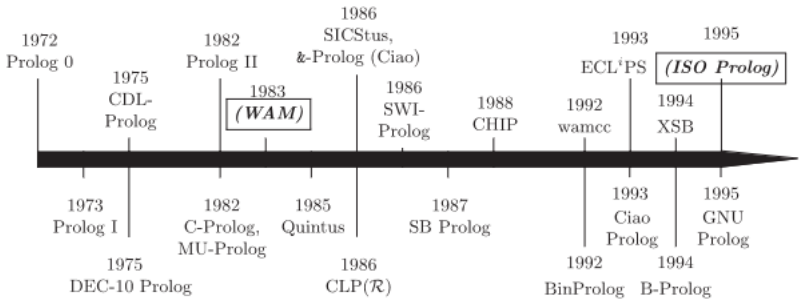

Figura 2.1: Evoluzione di Prolog [5]

Di seguito vedremo una panoramica delle principali tappe che hanno contrassegnato l'evoluzione di Prolog (vedi figura 2.1). Esploreremo le innovazioni chiave e i contributi accademici che hanno, negli anni, arricchito il linguaggio. Vedremo poi le due principali varianti di Prolog che sono state sviluppate nel corso degli anni, e come esse abbiano influenzato il linguaggio principale.

Prolog fa parte principalmente di tre rami della ricerca: intelligenza artificiale, risoluzione automatica dei teoremi e processamento del linguaggio naturale.

Il campo dell'intelligenza artificiale naque all'incirca nel 1956 e diede velocemente vita al linguaggio di programmazione funzionale LISP. Da li in poi l'interesse aumentò e diede vita ad altri linguaggi più o meno ad alto livello.

La soluzione automatica di teoremi è un campo della matematica che si occupa di trovare dimostrazioni automatiche di teoremi matematici. Questa materia fece un grande passo avanti quando Alan Robinson introdusse la regola di risoluzione per inferenza nel 1965. La risoluzione può essere utilizzata per ottenere una procedura semi-decisionale per la logica predicativa ed è al centro della maggior parte delle procedure di inferenza nella programmazione logica.

A seguito di questi progressi, un visionario precoce nello sviluppo del campo della programmazione logica è stato Cordall Green, che già alla fine degli anni 60 immaginava come estendere la risoluzione per costruire automaticamente la soluzione di un problema. In particolare voleva applicare questo

strumento per la risoluzione di domande basate sulla logica di primordine. Questo rappresentò il primo zenit della programmazione logica impiegata in applicazioni di AI.

Un altro progetto notabile è quello di Ted Elcock che sviluppò Absys, un linguaggio di programmazione dichiarativo che anticipò alcune delle funzioni di Prolog come la negazione come fallimento, operatori di aggregazione e il backtracking.

Nel frattempo Alain Colmerauer stava provando a sviluppare un sistema di conversazione uomo-macchina, ciò lo portò a sviluppare Q-systems, un sistema impegato per le traduzioni da Inglese a Francese dei rapporti meteorologici Canadesi. Il suo obiettivo di modificare Q-systems, per ottenere un sistema domanda e risposta, lo portò a sviluppare un linguaggio di programmazione basato sulla logica predicativa. Questo linguaggio fu chiamato Prolog, e fu sviluppato nel 1972 dal francese Alain Colmerauer. Il risultato fu quindi lo sviluppo di Prolog: un sistema di risoluzione lineare ristretto alle clausole di Horn che poteva risolvere problemi in maniera non deterministica.

L'articolo di Körner [5] fornisce una panoramica completa dell'evoluzione di Prolog, e di come questo linguaggio sia stato influenzato nel corso degli anni. Io mi concentrerò soltato su alcuni aspetti chiave che hanno portato Prolog a diventare il linguaggio che conosciamo oggi. Se siete interessati a una panoramica più completa seguendo una linea temporale come quella in figura 2.1 vi consiglio caldamente di leggere l'articolo.

2.2.1 Tappe fondamentali dell'evoluzione di Prolog

Nella sua prima decade di vita Prolog subì molte variazioni, la sua prima versione fu Prolog 0, seguita da Prolog 1. In queste versioni il linguaggio prevedeva già qualche predicato built in che poi fu incluso nello standard ISO. Queste versioni includevano già il predicato `dif/2`, con questo iniziò a prendere piede il concetto di unificazione. Successivamente seguì l'aggiunta della negazione come fallimento, questa fu una delle aggiunte più importanti, in quanto permise di risolvere problemi che non erano risolvibili con la sola unificazione.

Nel 1974 David H.D. Warren sviluppò Warplan, un sistema di pianificazione automatica basato su Prolog. Lo sviluppo di questo progetto, che poi divento la sua tesi di dottorato, lo portò a voler ottimizzare l'interprete Prolog, siccome ritenuto lento rispetto ad altri linguaggi di più alto livello come LISP. Sviluppò quindi un nuovo interprete, chiamato WAM (Warren Abstract Machine).

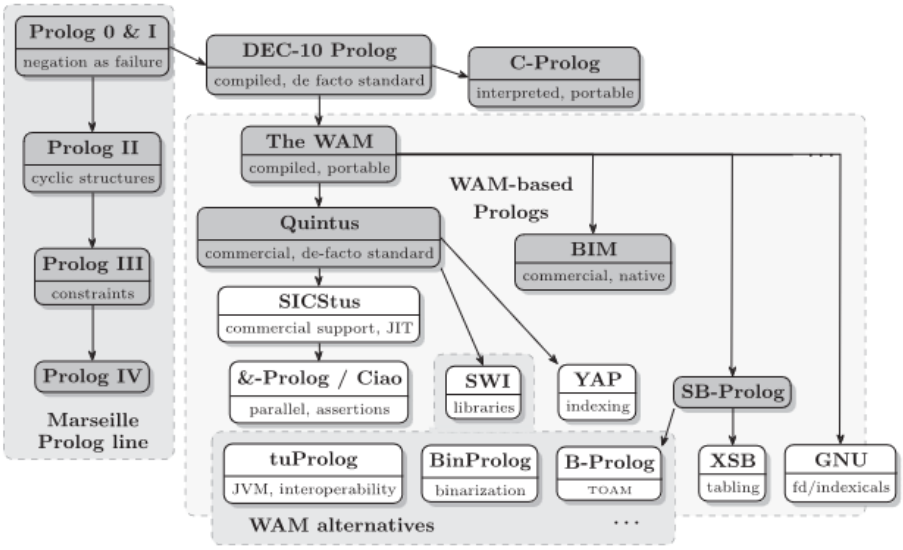

Figura 2.2: Quandro generale delle versioni di Prolog [5]. In grigio scuro sono riportate le versioni che ad oggi (2022 [5]) non sono più supportate.

WAM era, ed è tutt'ora, lo standard per i compilatori Prolog. Due delle implemenatazioni più di successo sono *GNU Prolog* e quella utilizzata per il mio progetto *SWI-Prolog*. Entrambe queste implementazioni sono open source e disponibili online.

GNU Prolog è un compilatore Prolog gratuito e open source, è stato sviluppato da Daniel Diaz. È stato sviluppato per essere un compilatore Prolog standard, e quindi non include estensioni pro-

prietarie. La sua implementazione utilizza WAM come base per produrre codice in C che verrà poi compilato da GCC. C fu poi rimpiazzato da un linguaggio miniassembly specializzato perchè ritenuto più efficiente.

SWI-Prolog naque dalla mancanza di Quintus (un implementazione di Prolog) di chiamate ricorsive tra Prolog e C. La potenza di questa implementazione è la community che la supporta, infatti è una delle implementazioni più utilizzate al mondo sia per la ricerca che non. Oltre che la forte community SWI-Prolog, al contrario di GNU Prolog, è interpretato, e non compilato. Questo permette di avere un ambiente di sviluppo molto più flessibile e adatto alla ricerca.

Questo, come detto in introduzione, è solo un breve riassunto delle tappe fondamentali dell'evoluzione di Prolog. Se siete interessati a una panoramica più completa vi consiglio caldamente di leggere l'articolo di Körner [5]. Nell'articolo viene descritto in modo più dettagliato ISO Prolog e le differenze tra le varie versioni utilizzate al giorno d'oggi.

2.3 Lavori notabili

Lo scopo di questa sezione è di presentare i lavori più notabili che hanno utilizzato Prolog o comunque un linguaggio logico per risolvere problemi di task planning e intelligenza artificiale. Durante lo sviluppo del progetto è stato necessario avere dei lavori di riferimento per capire come la comunità ha risolto questo tipo di problemi. Premesso questo non voglio quindi entrare troppo in dettaglio, ma soltanto illustrare i lavori lasciando al lettore la possibilità di approfondirli.

Un lavoro che mostra come i linguaggi di programmazione logica, e in particolare Prolog, possano essere usati per risolvere problemi di task planning è sicuramente quello di Meli [9]. In questo lavoro viene presentato lo stato dell'arte del task planning, e come questo possa essere creato con un linguaggio di programmazione logica. L'articolo inizia introducendo la necessità del mondo di oggi di progettare sistemi robotici con capacità di decisione deliberativa. Continua poi spiegando i requisiti per un sistema robotico deliberativo, quelli identificati sono questi moduli (vedi figura 2.3): *planning*, *acting*, *monitoring*, *observing*, *goal reasoning*, *learning*. Un sistema robotico è quindi un gruppo di robot con

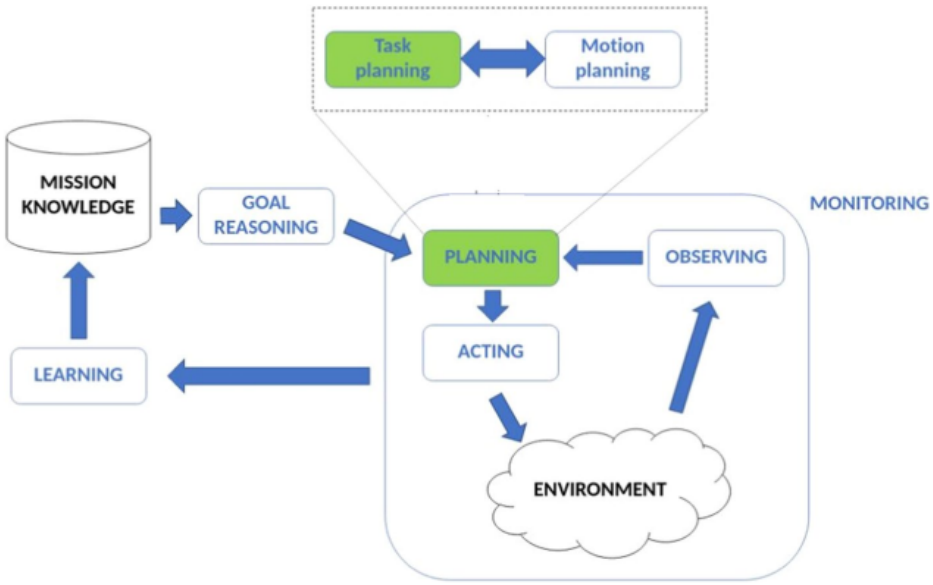

Figura 2.3: Funzioni di un robot deliberativo [9]

un obiettivo comune, questo può essere portato a termine attraverso un insieme di azioni che devono essere compiute dal robot nel mondo. Il paper poi continua introducedo la tassonomia dei task planner identificando tre macrotipologie di logica: *logica standard*, *logica temporale* e *logica probabilistica*. Da qui il paper introduce la task usata come esempio: *Peg transfer task*, in italiano compito di trasferimento perni. Il compito consiste nel posizionare gli elastici di un determinato colore nel perno del colore corrispondente. Viene poi formalizzata in *PDDL*, un linguaggio di descrizione di domini di pianificazione. Questo è stato fatto con l'obiettivo, poi, di tradurlo in un linguaggio logico. Segue poi

una descrizione della soluzione del problema nei tre tipi di logica precedentemente descritti. Conclude poi con una discussione dei risultati ottenuti spiegando come la logica standard sia la più adatta per risolvere questo tipo di problemi.

Un altro lavoro che utilizzò Prolog fu *Fifth Generation Computer Systems*. Questa fu una iniziativa di 10 anni, promossa dal Giappone, che aveva come obiettivo quello di creare i computer utilizzando tanta programmazione logica e programmazione parallela. Questo perchè come obiettivo finale si volevano avere dei sistemi per la ricerca in ambito di intelligenza artificiale. Più informazioni a riguardo di questo progetto possono essere trovate nel lavoro di Körner [5].

Anche la nota azienda IBM utilizzò Prolog per lo sviluppo di un computer capace di rispondere alle domande. Questo fu sviluppato nel 2011 e prese il nome di *Watson*. Nel 2013 IBM annuciò che Watson sarebbe stato utilizzato per l'aiuto a prendere decisioni riguardo al trattamento del cancro ai polmoni. Prolog è stato usato per creare la base di conoscenza e per l'analisi del linguaggio. Una curiosità è che anche la nostra università ha partecipato allo sviluppo iniziale di Watson insieme ad altre università statunitensi [14].

Un ultimo lavoro che utilizza Prolog per il design di un braccio robotico è *Design an Arm Robot through Prolog Programming Language* [10]. Lo scopo di questo articolo era quello di progettare un braccio robotico comandato da un programma Prolog. Nel paper viene introdotto il caso di studio fornendo al lettore le nozioni base di robot e sistema intelligente. Nella seconda parte poi viene spiegato il design utilizzato per il sistema Prolog e come è stato implementato, includendo anche il codice Prolog utilizzato per risolvere il problema.

Questi sono alcuni dei lavori più interessanti che ho trovato sia per il campo della robotica che per quello dell'intelligenza artificiale. Prolog ha una storia che conta più di 50 anni e quindi è stato utilizzato in molti altri lavori. Questa panoramica serviva per capire le potenzialità di Prolog e come questo sia utilizzato al giorno d'oggi. La ricerca nell'ambito di Prolog e dell'intelligenza artificiale continua a progredire, portando ad importanti sviluppi. Oltre al campo della robotica e dell'intelligenza artificiale, è rilevante menzionare anche l'intelligenza artificiale neurosimbolica. Questo approccio combina elementi di intelligenza artificiale simbolica, come Prolog, con tecniche di intelligenza artificiale ispirate al funzionamento del cervello, come le reti neurali artificiali. L'integrazione di questi due paradigmi promette di aprire nuove strade nel campo dell'intelligenza artificiale e potenzialmente superare alcune delle limitazioni dei singoli approcci. La ricerca in questo ambito è in continua evoluzione, offrendo nuove prospettive e sfide interessanti.

3 Descrizione del caso di studio

In questo capitolo verrà presentato più in dettaglio il caso di studio. Verrà definito inizialmente il workflow adottato durante il periodo di tirocinio, spiegando le milestone principali raggiunte. Successivamente verranno illustrate le parti principali del programma Prolog, approfondendo il funzionamento e le sfide incontrate durante lo sviluppo.

Lo sviluppo del task planner è stato oggetto del mio periodo di tirocinio, il lavoro è durato complessivamente circa 4 mesi. Durante questo periodo ho dovuto inizialmente studiare il linguaggio partendo dalle sue basi, sviluppare un idea e successivamente implementarla. In robotica un task planner è un componente software che si occupa di pianificare e coordinare le attività del robot al fine di raggiungere determinati obiettivi. Il task planner prende in input il goal, analizza il constesto in cui il robot opera e restituisce in output un piano che indica l'ordine delle azioni da eseguire per completare con successo il compito assegnato. Una volta generato, questo viene eseguito dal sistema di controllo del robot, nel mio caso il *motion planner*. In sostanza, il task planner è responsabile della pianificazione delle attività del robot in modo efficiente e intelligente. È uno dei componenti fondamentali per il controllo e automazione delle applicazioni robotiche complesse.

Descriviamo ora le milestone individiate per lo sviluppo del progetto:

- Prendere domestichezza con Prolog: fase preliminare di studio del linguaggio

- Sviluppare il codice in Prolog: fase di sviluppo del codice
- Wrappare il codice Prolog con un linguaggio ad alto livello: fase di ricerca e studio della libreria adatta per il wrapping
- Creare la simulazione in Gazebo e ROS: fase di sviluppo della simulazione

In questo capitolo voglio entrare nel merito delle prime due milestones, la terza e la quarta verranno approfondite nel capitolo 4. Queste milestone verranno contestualizzate con una data indicativa sia per l'inizio che per la fine. Per ogniuna verrà indicato il lavoro svolto e il suo contributo per la realizzazione del caso di studio.

3.1 Prima milestone: Studio di Prolog

In questa sezione andreamo a vedere in dettaglio la prima milestone, ovvero lo studio del linguaggio Prolog. Questa è stata fondamentale per lo sviluppo del task planner, in quanto mi ha permesso di capire le potenzialità del linguaggio e come sviluppare il progetto. Prolog ha una curva di apprendimento molto ripida, infatti il tempo che ho dedicato a questa fase è stato all'incirca di un mese e mezzo, da marzo fino a metà aprile. Durante questo periodo mi sono studiato lo stato dell'arte del linguaggio, sia in ambito robotica e task planner che non (vedere capitolo 2) e mi sono esercitato con esso.

Inizialmente sono stato introdotto al linguaggio tramite video su YouTube e articoli online. Una cosa che è stata fondamentale per lo studio del linguaggio è stato il libro *Prolog Programming for Artificial Intelligence* [1] e il materiale fornito con esso. Questo conteneva delle esercitazioni che modellavano il noto problema del *Blockworld*. In queste lezioni infatti veniva spiegato come costruire una base di conoscenza che contenesse al suo interno le informazioni necessarie per descrivere lo stato dei blocchi nello spazio tridimensionale. L'esercizio supponeva che ci fosse una telecamera posta sopra il nostro 'mondo' e che fosse ortogonale ad esso, quindi la coordinata z non poteva essere calcolata con la sola osservazione. I blocchi erano rappresentati con il fatto `see/3`, questo identificava un blocco alla coordinata X e Y specificata. C'era anche un predicato aggiuntivo che specificava il contatto del blocco, questo era il predicato `on/2`. Questo serviva a specificare se il blocco si trovasse sopra un altro blocco oppure sopra il tavolo. Un esempio di base di conoscenza utilizzante questi fatti è la seguente:

```
see(a,2,5).
see(d,5,5).
see(e,5,2).

on(a,b).
on(b,c).
on(c,table).
on(d,table).
on(e,table).
```

Come si può notare da questo esempio, l'ipotetica telecamera individua solamente i blocchi *a*, *d*, *e* siccome *b* e *c* sono sotto *a*. Essendo uno spazio tridimensionale doveva essere identificata anche la coordinata z. Questa veniva calcolata tramite un approccio ricorsivo. La ricorsione in questi paradigmi di programmazione è ampiamente usata e questo primo esempio mi è servito per capire come funzionasse. L'algoritmo per calcolare la coordinata z di un blocco era il seguente:

```
z(Block,0) :- on(Block, table).
z(Block,Z) :- on(Block, OtherBlock), z(OtherBlock, Z1), Z is Z1 + 1.
```

Questo è un classico esempio di algoritmo ricorsivo in cui si ha un caso base e un caso ricorsivo. Il caso base indica che se il blocco si trova sul tavolo la sua coordinata sarà 0, quello ricorsivo invece itera finchè non trova il blocco che poggia sul tavolo, poi ricorsivamente incrementa di 1 la coordinata z.

In seguito, verso fine marzo inizio aprile, ho provato a ampliare questo esempio cercando un modo efficiente di rappresentare la realtà. È quindi iniziato il periodo di sviluppo del codice in cui il programma ha iniziato a prendere forma. Tutte le varie versioni del programma sono documentate e reperibili nella repository GitHub del progetto [8]. Vediamo ora come è stato sviluppato il codice del planner e le sfide incontrate.

3.2 Seconda milestone: Sviluppo del planner

In questa sezione approfondiremo le fasi di creazione del planner, il codice del planner e le sfide incontrate durante lo sviluppo. Il periodo effettivo di creazione del codice del planner è iniziato verso metà aprile ed è finito all'incirca a metà giugno. Ritengo che questo sia stato il compito più impegnativo e allo stesso tempo più interessante del progetto. Impegnativo perchè come detto prima Prolog non è un linguaggio semplice da imparare e creare qualcosa da 0 può risultare ostico, interessante perchè mi ha permesso di capire come funzionano i task planner e delle potenzialità di Prolog. Andiamo quindi a dividere questa sezione in tre parti, cioè le tre parti che compongono il programma: la rappresentazione dei blocchi, le azioni e il generatore del piano.

3.2.1 Rappresentazione dei blocchi

Una parte fondamentale del mio progetto è stata la scelta della convenzione per rappresentare i blocchi nella base di conoscenza. Questa, come detto nel capitolo 2, è composta da fatti e regole. I fatti sono le informazioni che il programma ha a disposizione per ragionare, mentre le regole sono le azioni che il programma può eseguire per modificare lo stato del mondo. In questa versione, che non utilizza la logica probabilistica, i fatti sono istanziati dall'utente. In questo caso si dice che sono *grounded*, quindi veri al 100%. Premesso ciò andiamo a vedere come sono stati rappresentati i blocchi, presentando anche l'evoluzione nelle varie versioni.

Inizialmente la rappresentazione era stata ideata con il seguente fatto:

```
block(ID, X, Y, Z, W, H, D, O, S).
```

L'ID rappresenta un identificativo univoco del blocco, questo può essere ad esempio b1. La tripla X, Y, Z rappresenta le coordinate nello spazio tridimensionale della posizione del blocco, queste vengono usate per localizzare il blocco nello spazio. La tripla W, H, D rappresenta la dimensione del blocco, rispettivamente la lunghezza, l'altezza e la profondità. Infine il parametro O rappresenta l'orientamento del blocco rispetto al mondo. Per ragioni di semplicità non abbiamo voluto rappresentarlo in una maniera canonica come gli angoli di eulero oppure i quaternioni ma, invece, con un intero che indica una posizione fissata del blocco. Questo intero ha un range da 1 a 6 e questi numeri indicano: 1 = il blocco è posizionato col la testa[1] rivolta verso l'alto (posizione usuale del blocco per essere impilato), 2 = il blocco è sottosopra quindi la testa poggierà sul tavolo, 3 = il blocco poggia sul lato e la testa è verso l'interno del tavolo, 4 = il blocco poggia sul lato e la testa è verso l'esterno del tavolo, 5 = il blocco poggia sul lato e la testa è verso la parte sinistra del tavolo, 6 = il blocco poggia sul lato e la testa è verso la parte destra del tavolo. S rappresenta la forma del blocco che nel nostro caso di studio sarà sempre un cubo. Questo parametro quindi non verrà mai utilizzato, è stato solamente aggiunto per la completezza.

Questa rappresentazione, però non era ottimale per il mio scopo. Non teneva conto del fatto che i blocchi possono essere impilati e quindi della relazione di contatto tra i blocchi. Dopo varie riflessioni ho quindi deciso di aggiungere due nuovi attributi al mio fatto `block`:

```
block(ID, X, Y, Z, W, H, D, O, CH, CL, S).
```

Sono stati aggiunti gli attributi CH e CL. Questi servono a specificare il "contatto alto" e il "contatto basso". Con contatto alto mi riferisco al contatto che il blocco ha con la sua testa, mentre con contatto basso mi riferisco al contatto che il blocco ha con la sua base. Ad esempio se il blocco b1 dovesse essere sopra al blocco b2 il suo attributo CL sarebbe b2 mentre l'attributo CH di b2 sarebbe b1. Questo fornisce già un astrazione abbastanza adeguata per la nostra realtà, ma con l'evolvere del progetto non si è rivelata sufficiente. Era necessario infatti che venisse in qualche modo rappresentato il concetto di

[1] Con testa mi riferisco alla parte superiore del blocco di lego

blocco composto da due blocchi. Bisognava quindi trovare un modo di creare e modificare i fatti *on the fly* e di rappresentare la nozione di blocco composto. Il primo problema è stato risolto con l'aggiunta della proprietà *dynamic* al predicato block. Questo permette di modificare i fatti in runtime, quindi di aggiungere e rimuovere fatti. La rimozione e l'aggiunta di nuovi fatti è possibile tramite i predicati built-in `retract/1` e `assertz/1`. Il primo serve a rimuovere un fatto dalla base di conoscenza, il secondo invece serve ad aggiungerlo. Per rendere un predicato dinamico serve specificarlo a inizio file con la seguente direttiva: `:- dynamic block/13` nel caso del blocco. Per rappresentare invece il blocco composto mi sono servito di due ulteriori attributi, il predicato `block/13` è quindi così composto:

```
block(ID, X, Y, Z, W, H, D, O, CH, CL, S, MB, L).
```

Questi ultimi due attributi sono MB e L, servono rispettivamente a identificare i blocchi di cui è composto il blocco e a indicare se il blocco fa parte di un blocco composto o no. Porto un esempio così da essere più chiaro:

```
block(b1, 1, 0, 0, 1, 2, 1, 1, b2, table, block, [b1], 1).
block(b2, 1, 0, 2, 1, 2, 1, 1, air, b1, block, [b2], 1).
block(s1, 1, 0, 0, 1, 4, 1, 1, air, table, block, [b1, b2], 0).
```

Il blocco b1 si trova in coordinate 1, 0, 0, ha dimensioni 1, 2, 1 e sopra di lui si trova b2 e sotto il tavolo, è composto da se stesso e fa parte di un blocco composto. Il blocco b2 si trova in coordinate 1, 0, 2, ha dimensioni 1, 2, 1 e sopra di lui si trova l'aria e sotto b1, è composto da se stesso e fa parte di un blocco composto. Infine, il blocco composto s1 si trova in coordinate 1, 0, 0, ha dimensioni 1, 4, 1 e sopra di lui si trova l'aria e sotto il tavolo, è composto da b1 e b2 e non fa parte di un blocco composto.

Questa è la rappresentazione che, ad oggi, ritengo più efficace per rappresentare questa astrazione. Avendo il progetto dei possibili scenari futuri che potrebbero richiedere una rappresentazione diversa, questa potrebbe cambiare in futuro.

3.2.2 Azioni del planner

Ora andremo a vedere il set di azioni che il planner può eseguire e come queste sono state implementate in Prolog. Ho individuato 3 tipi diversi di azioni: *move_block*, *rotate_block* e *link*. Verra riportata la loro implementazione e di seguito una spiegazione del loro funzionamento. Queste azioni vanno intese come astrazioni ad alto livello di azioni che il robot può compiere. Il compito del task planner, quindi, non è quello di descrivere nei minimi dettagli il movimento che dovrà compiere il braccio, quello è compito del motion planner. Il suo scopo è di fornire un piano ad alto livello che il robot dovrà seguire per raggiungere lo stato finale desiderato.

Iniziamo con la `rotate_block/5`, questa azione serve a portare un blocco dal suo orientamento attuale a quello desiderato, ovviamente per i nostri scopi l'orientamento desiderato sarà sempre 1. L'implementazione che ho scelto per questa azione è la seguente:

```
1 rotate_block(Block, X, Y, Z, NO) :-
2     block(Block, X, Y, Z, W, H, D, O, TL, TH, S, MB, L),
3     (list_length(MB, N), N > 1 -> rotate_list(MB, NO); NO = NO),
4     L is 0,
5     add_action(rotate(Block, X, Y, Z, NO)),
6     retract(block(Block, X, Y, Z, W, H, D, O, TL, TH, S, MB, L)),
7     assertz(block(Block, X, Y, Z, W, H, D, NO, TL, TH, S, MB, L)).
```

Come si può vedere, la testa della nostra clausola di Horn prende in input 5 parametri: il blocco da ruotare, le sue coordinate e il nuovo orientamento. I primi 4 parametri vengono utilizzati nella riga successiva per cercare di unificare con il predicato `block/13` e ottenere così le informazioni del blocco. Avendo il blocco ora ne conosciamo tutte le sue proprietà tra cui l'orientamento attuale e se è un blocco composto. Quest'ultimo viene usato nell'istruzione successiva, se stiamo operando un blocco composto da più blocchi allora dobbiamo ruotare anche questi ultimi. Per farlo utilizzo il predicato `rotate_list/2` che prende in input la lista dei blocchi che compongono il blocco composto e il nuovo orientamento. Questo predicato si occupa di ruotare tutti i blocchi della lista.

```
rotate_list([], NO).

rotate_list([H|T], NO) :-
    rotate_compose(H, _, _, _, NO),
    rotate_list(T, NO).

rotate_compose(Block, X, Y, Z, NO) :-
    block(Block, X, Y, Z, W, H, D, O, TL, TH, S, MB, L),
    (list_length(MB, N), N > 1 -> rotate_list(MB, NO); NO = NO),
    retract(block(Block, X, Y, Z, W, H, D, O, TL, TH, S, MB, L)),
    assertz(block(Block, X, Y, Z, W, H, D, NO, TL, TH, S, MB, L)).
```

Con questa serie di predicati riusciamo appunto a ruotare tutti i blocchi che lo compongono. Iteriamo inizialmente tutti gli elementi nella lista e poi li passiamo uno ad uno al predicato `rotate_compose/5`. Questo è identico al predicato principale, l'unica differenza è che non controlla se un blocco fa parte di un blocco composto oppure no (L=1).

Tornando al predicato principale, successivamente verrà svolto il controllo per verificare che il blocco non faccia parte di un blocco composto. Svolto quest'utlimo controllo possiamo aggiungere l'azione eseguita al piano (verrà spiegata meglio nella sezione 3.2.3) e infine aggiornare il predicato `block/13` con il nuovo orientamento. Per svolgere quest'ultima azione utilizzeremo i predicati built-in `retract/1` e `assertz/1` che ci permettono di rimuovere un predicato e aggiungerne uno nuovo.

La seconda azione presa in considerazione è la `move_block/7`, questa azione serve a spostare un blocco da una posizione ad un'altra. L'implementazione che ho scelto per questa azione è la seguente:

```
move_block(Block, X, Y, Z, NX, NY, NZ) :-
    block(Block, X, Y, Z, W, H, D, O, TL, TH, S, MB, L),
    (list_length(MB, N), N > 1 -> move_list(MB, NX, NY, NZ); NX = NX, NY = NY, NZ = NZ),
    L is 0,
    add_action(move(Block, X, Y, Z, NX, NY, NZ)),
    retract(block(Block, X, Y, Z, W, H, D, O, TL, TH, S, MB, L)),
    assertz(block(Block, NX, NY, NZ, W, H, D, O, TL, TH, S, MB, L)).
```

Questo predicato è molto simile se non uguale al precedente, cambia solo il fatto che non si deve ruotare il blocco e che si devono aggiornare le coordinate del blocco. La logica quindi è uguale come la sua implementazione, le uniche differenze sono che nel piano verrà aggiunta l'azione `move/7` e che il predicato `block/13` verrà aggiornato con le nuove coordinate. Anche in questo caso c'è il controllo sul blocco composto, quindi ci sarà anche l'implementazione di `move_list/4` che è molto simile a `rotate_list/2`.

L'ultima azione presa in considerazione per il nostro caso di studio è la `link/3`. Questa, essenzialmente, serve a verificare se i blocchi sono impilati, se si gli "incolla" creando il blocco composto. La sua implementazione è la seguente:

```
link(B1, B2, R) :-
    block(B1, X1, Y1, Z1, W1, H1, D1, O1, TL1, TH1, S1, MB1, L1),
    block(B2, X2, Y2, Z2, W2, H2, D2, O2, TL2, TH2, S2, MB2, L2),
    count(s, N),
    all_diff([B1, B2]),
    L1 = 0,
    L2 = 0,
    X1 = X2,
    Y1 = Y2,
    Z1 is Z2 - H2,
    W1 = W2,
    D1 = D2,
    HighP is H1 + H2,
```

```
14      add_action(link(B1, B2)),
15      retract(block(B1, X1, Y1, Z1, W1, H1, D1, O1, TL1, TH1, S1, MB1, L1)),
16      retract(block(B2, X2, Y2, Z2, W2, H2, D2, O2, TL2, TH2, S2, MB2, L2)),
17      assertz(block(B1, X1, Y1, Z1, W1, H1, D1, O1, B2, TH1, S1, MB1, 1)),
18      assertz(block(B2, X2, Y2, Z2, W2, H2, D2, O2, TL2, B1, S2, MB2, 1)),
19      string_concat('s', N, PIL),
20      retract(count(s, N)),
21      N1 is N+1,
22      assertz(count(s, N1)),
23      atom_string(STACK,PIL),
24      R = STACK,
25      assertz(block(STACK, X2, Y2, Z2, W1, HighP, D1, 1, TL2, TH1, block, [B1,B2],0)).
```

Iniziamo definendo in input i due blocchi che vogliamo "incollare", R è un parametro in cui verrà salvato l'ID del blocco composto creato. Le due istruzioni seguenti servono a recuperare le informazioni dei due blocchi che vogliamo utilizzare cercando di unificarli con i predicati `block/13`. L'istruzione a linea 5 serve a controllare che i due blocchi non siano uguali, questo perchè non ha senso creare un blocco composto con un solo blocco. Verifichiamo poi un insieme di precondizioni tra cui la verifica che i due blocchi non siano già parte di un blocco composto, che siano allineati e che siano della stessa dimensione. Successivamente ci calcoliamo l'altezza del blocco composto come somma delle altezze dei due blocchi. Aggiorniamo la base di conoscenza e creiamo aggiungiamo anche il blocco composto che avrà come ID la stringa "s" seguita da un numero che viene incrementato ad ogni blocco composto creato.

Queste sono le tre azioni base che il planner deve eseguire per poter risolvere il problema, ora vediamo come vengono utilizzate per generare il piano che poi verrà eseguito dal nodo di motion planning.

3.2.3 Generatore del piano

La generazione del piano è il cuore del planner, è qui che vengono utilizzate le azioni base per generare il piano che verrà eseguito dal nodo di motion planning. Il task planner è composto, per il nostro scopo, principalmente da un predicato: `pillar/7`. Questo è strutturato nel seguente modo:

```
1  pillar(X, Y, Z, High, Width, Depth, Actions) :-
2      find_blocks(Blocks),
3      valid_blocks(Blocks, High, ValidBlocks, Width, Depth),
4      nth0(0, ValidBlocks, B1),
5      nth0(1, ValidBlocks, B2),
6      stack(B1, B2, X, Y, Z, R),
7      select(BL1, ValidBlocks, ValidBlocks1),
8      select(BL2, ValidBlocks1, ValidBlocks2),
9      stackRec(ValidBlocks2, R, X, Y, Z),
10     plan(Actions).
```

I primi 3 argomenti sono le coordinate dove verrà costruito il pilastro, i tre seguenti sono le dimensioni del pilastro. L'ultimo argomento è l'output del planner cioè una lista di azioni che verranno eseguite dal nodo di motion planning. Il predicato inizia quindi salvando nella lista *Blocks* tutti i blocchi utilizzabili, quindi con L=0, presenti nella base di conoscenza. Successivamente, tramite un algoritmo di esplorazione ricorsiva, trova la combinazione di blocchi adatta per soddisfare l'altezza richiesta dall'utente. Il codice prolog è il seguente:

```
1  valid_blocks(Blocks, DesiredHeight, ResultBlocks, DesiredWidth, DesiredDepth) :-
2      length(Blocks, N),
3      between(1, N, NumBlocks),
4      length(ResultBlocks, NumBlocks),
5      select_blocks(Blocks, ResultBlocks, DesiredHeight, DesiredWidth, DesiredDepth).
6
```

```
7
8  select_blocks(_, [], 0.0, DesiredWidth, DesiredDepth).
9  select_blocks(Blocks, [Block|Remaining], DesiredHeight, DesiredWidth, DesiredDepth) :-
10     seleziona(Block, Blocks, RemaningBlocks),
11     block(Block, _, _, _, DesiredWidth, Height, DesiredDepth, _, _, _, _, _, _),
12     DesiredHeight >= Height,
13     RemainingHeight is DesiredHeight - Height,
14     select_blocks(RemaningBlocks, Remaining, RemainingHeight, DesiredWidth, DesiredDepth).
15
16
17 seleziona(X, [X|Resto], Resto).
18 seleziona(X, [Y|Resto], [Y|Resto1]) :-
19     seleziona(X, Resto, Resto1).
```

A prima vista può sembrare complesso ma cercherò di spiegarlo in maniera semplice ed efficace. L'entrypoint di questa "procedura" è `valid_blocks/5`, il compito di questo predicato è di istanziare una lista di lunghezza compresa tra 1 e n, dove n è il numero di blocchi disponibili. Inizialmente la lista verrà istanziata a 1, nel caso l'iterazione fallisca questa verrà incrementata di 1 fino ad arrivare alla capienza massima cioè n. Successivamente viene chiamato il predicato `select_blocks/5`, questo restituirà la lista contenente i blocchi che soddisfano i requisiti di altezza, larghezza e profondità richiesti dall'utente. Inizialmente, quindi, il programma prolog cercherà di trovare una soluzione con un solo blocco, se non la trova passerà a due blocchi e così via fino ad arrivare ad una soluzione oppure al fallimento del predicato.

Trovata la lista di blocchi che soddisfa i requisiti, il programma prosegue con la creazione del pilastro. Questo viene creato tramite i predicati `stack/6` e `stackRec/5`, l'ultimo è la versione ricorsiva in cui si passa la lista dei blocchi da utilizzare e l'ID del blocco composto di "base". Questi due predicati utilizzano le azioni spiegate prima per creare il pilastro. L'ultima istruzione, `plan/1`, copierà le azioni create durante la generazione del pilastro nella lista Actions che verrà restituita in output dal predicato `pillar/7`.

Questa è quindi la soluzione al caso di studio che ho pensato e implementato. Ovviamente è migliorabile e ci sono sicuramente altri modi per risolvere questo problema, ma questo è il risultato che ho ottenuto e che ho deciso di presentare in questa tesi.

4 Descrizione dell'architettura ROS

Nel capitolo precedente ho approfondito il linguaggio Prolog e come ho sviluppato il caso di studio. In questo capitolo verrà descritta l'architettura ROS utilizzata per la simulazione e la possibile applicazione reale. In particolare verrà data una breve descrizione di cosa è ROS e come funziona, per poi passare alla descrizione dell'architettura utilizzata per la simulazione concludendo poi con una sezione dedicata alle difficoltà incontrate. Iniziamo quindi con una breve introduzione a ROS.

4.1 Introduzione a ROS

ROS (Robot Operating Systems) è un framework opensource per lo sviluppo e la programmazione di robot. ROS implementa drivers e algoritmi che sono lo stato dell'arte in robotica grazie anche alla community che negli anni si è formata e ingrandita. Essendo infatti un progetto opensource, ROS è sviluppato e mantenuto dalla community stessa. Questo è un grosso vantaggio infatti così facendo il supporto verso nuovi hardware sarà sempre al passo con i tempi. Oltre a questo la community è molto attiva per consigli e aiuti sia per i meno esperti che non e questo sicuramente è un suo punto a favore. Per realizzare questo capitolo mi sono documentato dalla wiki del progetto [12] e dall'articolo *ROS: an open-source Robot Operating System* [11].

Iniziamo quindi introducendo i concetti principali di un architettura ROS.

4.1.1 Nodi

Con il termine nodo si intende un processo che esegue un compito specifico, questo utilizza ROS per comunicare con gli altri nodi. I nodi possono pubblicare oppure iscriversi ad un topic, inoltre possono offrire un servizio o richiederlo.

In ROS1, quello che è stato utilizzato per la simulazione, è necessario che ci sia un nodo master. Per avviarlo è necessario digitare il comando `roscore`. Questo avrà il compito di coordinare le attività del nostro sistema distribuito. Più in dettaglio dovrà: tenere traccia dei nomi dei nodi ecc. tramite il registro dei nomi, fornire un servizio di ricerca consultabile dai nodi stessi, tenere traccia dei topic e dei messaggi che vengono pubblicati e infine fornire il servizio del parameter server. I messaggi e le chiamate di servizio non passano dal nodo master, infatti la comunicazione in ROS è peer-to-peer. Questo permette di avere un sistema distribuito e scalabile. Inoltre, essendo peer-to-peer, non è necessario che tutti i nodi siano attivi contemporaneamente, infatti è possibile che un nodo si sottoscriva ad un topic anche se il nodo che lo pubblica non è ancora attivo. Questo permette di avere un sistema molto flessibile e scalabile.

In conclusione un nodo è un processo che esegue un compito specifico e che comunica con gli altri nodi attraverso i topic, i servizi e il parameter server.

4.1.2 Topics

In ROS i topic sono un canale di comunicazione asincrona che permette ai nodi di scambiarsi messaggi. I Topic in ROS seguono l'architettura *publisher-subscriber*, un nodo quindi può pubblicare messaggi su un topic specifico mentre uno o più nodi si sottoscrivono al topic per ricevere i messaggi. I topic in ROS sono organizzati gerarchicamente e sono identificati da un nome univoco. La gerarchia permette di organizzarli in base diversi aspetti dell'applicazione o dei dati che vengono scambiati. Come detto prima la comunicazione è asincrona, quindi il nodo pubblicante non aspetta una risposta immediata dal nodo abbonato. Con il comando `rostopic list` possiamo avere una panoramica dei topic che sono attivi nella nostra rete. Con il comando `rostopic echo <nometopic>`, invece, possiamo vedere i messaggi che vengono pubblicati su un topic specifico. In questo modo possiamo capire se i topic sono attivi e quindi se i nodi stanno comunicando tra di loro.

In conclusione i topic sono quindi un canale di comunicazione asincrona che permette ai nodi di scambiarsi messaggi. Questi sono organizzati gerarchicamente e sono identificati da un nome univoco.

4.1.3 Messaggi

I messaggi sono i dati che vengono scambiati dai nodi attraverso i topic. Questi possono essere di vari tipi e sono definiti in un file `.msg`. Questo file contiene la definizione del messaggio, ovvero il nome del messaggio e i campi che lo compongono. I campi possono essere di vari tipi, come ad esempio: interi, stringhe, array, ecc. La personalizzazione dei messaggi permette all'utente un controllo totale della propria applicazione riuscendo a definire i messaggi che meglio si adattano alle proprie esigenze. Con il comando `rostopic info <nometopic>` possiamo avere una panoramica dei messaggi che vengono scambiati su un topic specifico. Inoltre con il comando `rosmsg show <nomemsg>` possiamo avere i dettagli di come è composto il messaggio specifico.

In conclusione i messaggi sono i dati che vengono scambiati dai nodi attraverso i topic. Questi sono definiti in un file `.msg` e possono essere di vari tipi per adattarsi alle esigenze dell'utente.

4.1.4 ROS services e parameter server

L'ultima nozione fondamentale da sapere sull'architettura ROS è il concetto di *ROS service* e *parameter server*. I ROS service sono un modo alternativo per comunicare in ROS, questi a differenza dei topic sono sincroni. Questo significa che il nodo che richiede un servizio deve aspettare che il nodo che lo fornisce risponda. Anche i ROS services sono identificati da un nome univoco e vengono chiamati/richiesti appunto da esso. Una volta che un nodo richiede un servizio questo bloccherà la sua esecuzione, riprendendola solamente quando il servizio sarà stato fornito. I servizi in ROS vengono usati principalmente se è necessario ottenere una risposta immediata, ad esempio la richiesta di dati di configurazione, l'esecuzione di operazioni complesse oppure la richiesta di operazioni specifiche.

Il *parameter server* in ROS è un componente centrale che consente ai nodi di memorizzare e recuperare dei parametri di sistema. Può essere inteso come un database centralizzato in cui i nodi

possono leggere e scrivere i valori dei parametri. Questi parametri possono essere usati per configurare il comportamento dei nodi, impostare valori costanti o condividere dati tra i componenti del sistema. I parametri sono organizzati tramite dei namespace così da semplificare la ricerca e la gerarchia.

4.1.5 Esempio rete ROS

Per capire meglio come sono strutturati questi componenti utilizziamo il famoso simulatore *turtlesim* e generiamo il grafo della rete tramite il comando `rosrun rqt_graph rqt_graph`. Il grafo generato è il seguente I nodi sono identificati dagli ovali mentre i topic sono identificati dai rettangoli. Vediamo

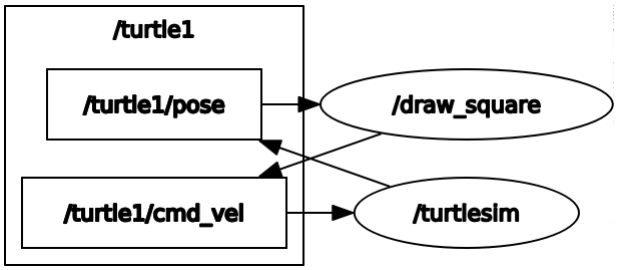

Figura 4.1: Grafo della rete turtlesim

come i due nodi: turtlesim e draw_square comunicano tra loro tramite i topic `/turtle1/pose` e `/turtle1/cmd_vel`. Il nodo /draw_square pubblica sul topic /cmd_vel la traiettoria per la "tartaruga", questa sarà poi inviata al nodo del simulatore per poi essere eseguita. Quest'ultimo a sua volta pubblicherà la posa del robot al topic /pose che sarà poi letto dal nodo /draw_square per recuperare la posizione del robot e disegnare la traiettoria. Il risultato è visibile in figura 4.2.

Figura 4.2: Esempio di simulazione turtlesim

Passiamo ora a vedere come è strutturata la rete per il mio caso di studio

4.2 Architettura utilizzata

In questa sezione mi concentrerò a illustrare l'architettura utilizzata per il mio caso di studio andando anche a definire le milestone indicate nel capitolo 3. Utilizzerò appunto queste per dare la struttura a questo capitolo, erano rimaste da spiegare quindi il wrapping del codice Prolog e la creazione della simulazione.

4.2.1 Terza milestone: Wrapping codice Prolog

In informatica, il termine wrapping si riferisce a una tecnica o un processo in cui un elemento o un oggetto viene incapsulato da un altro strato o struttura. Questo viene fatto per rendere l'interfaccia

del programma più semplice oppure per astrarre maggiormente la stessa.

Nel mio caso il codice prolog è stato incapsulato grazie alla libreria PySwip [13], questa fornisce un interfaccia per comunicare con il motore di inferenza SWI-Prolog tramite il linguaggio Python. In questo modo ho potuto utilizzare il codice prolog come se fosse una libreria Python, rendendo l'interfaccia più semplice e più comprensibile. Oltre al fattore di semplicità, questo è stato fatto per poi creare il nodo ROS utilizzando la client library rospy. Il processo di wrapping è molto semplice, si utilizzano le API messe a disposizione da PySwip per interrogare la base di conoscenza, si ottengono i risultati e si utilizzano per creare il nodo ROS. Qui di seguito riporto come è stato fatto il wrapping del codice per il mio progetto, il file `prolog_node.py` contiene il seguente codice:

```
from pyswip import Prolog
prolog = Prolog()
prolog.consult("block_world.pl")
result = list(prolog.query("get_blocks(Blocks)"))
result = list(prolog.query(
                "once(pillar(" + x + "," + y + "," + z +
                "," + h + "," + w + "," + d + ", Actions))"))
```

Il codice riportato non corrisponde riga per riga a quello completo, ho riportato soltanto dove viene utilizzato il bridge. Nella prima istruzione dalla libreria PySwip viene importata la classe Prolog, questa ci permette di interrogare la base di conoscenza. Viene istanziato quindi un oggetto della medesima classe e viene caricato il file prolog tramite il metodo `consult`. Nelle due righe successive vengono svolte due query: una serve a recuperare la lista dei blocchi presenti nella scena, l'altra invece serve a recuperare la lista di azioni da eseguire per risolvere il problema. Il metodo `query` restituisce un generatore, per questo motivo viene utilizzato il metodo `list` per convertirlo in una lista.

Come detto prima, il processo di wrapping non è complesso e, come si può ben vedere, è intuitivo e compatto. Per la realizzazione di questa milestone ho impiegato poco meno di una settimana, in questo caso i primi giorni sono serviti per ricercare una libreria che mi permettesse di fare il wrapping del codice prolog, mentre i restanti giorni sono serviti per capire come utilizzare la libreria e per scrivere il codice. Siccome al momento di realizzazione di questa milestone, circa il 18 giugno, l'architettura ROS non era stata ancora creata, ho dovuto creare un nodo di prova per testare il wrapping del codice prolog. Questo è consultabile nella mia repository ed è il file `python_node_poc.py`. Questo file conteneva una *proof of concept* di ciò che volevo realizzare, ovvero un nodo che interrogasse la base di conoscenza e che mostrasse i risultati ottenuti. Ho quindi realizzato il programma, questo chiedeva all'utente i dati in input per il predicato `pillar/7` e poi interrogava la base di conoscenza. I risultati poi venivano mostrati tramite un interfaccia grafica minimale realizzata con la libreria Tkinter.

Sviluppata questa milestone, mancava soltanto la creazione della rete ROS e la simulazione.

4.2.2 Quarta milestone: Simulazione

Questa è l'ultima milestone, la realizzazione della simulazione in ROS e gazebo. Questa, pur essendo secondaria al mio progetto di tesi, è stata molto importante perchè offre un esempio pratico del caso di studio. Per la realizzazione della simulazione ho utilizzato ROS Noetic e Gazebo, quest'ultimo è un simulatore fisico 3D open source che permette di simulare robot, oggetti e ambienti. La simulazione del UR5 è stata realizzata utilizzando il framework *locosim* [3], questo rende disponibile out-of-the-box il modello del robot con annessi controller per la simulazione in gazebo. Per questa milestone quindi ho dovuto sviluppare i due nodi ROS che si occupano di interrogare la base di conoscenza e di eseguire le azioni ottenute.

L'architettura ROS ideata per il mio caso di studio è illustrata in figura 4.3. Come si vede dalla figura i nodi sono 3: il task planner, il motion planner e il nodo per la simulazione del UR5. In particolare, quelli sviluppati da me sono appunto il nodo in prolog e quello di motion planning. Iniziamo descrivendo appunto quest'ultimo.

Motion planner

Il nodo di motion planning ha come obiettivo quello di pianificare ed eseguire i movimenti che il robot deve compiere per effettuare le azioni ottenute dal task planner. Ci sono varie strategie per

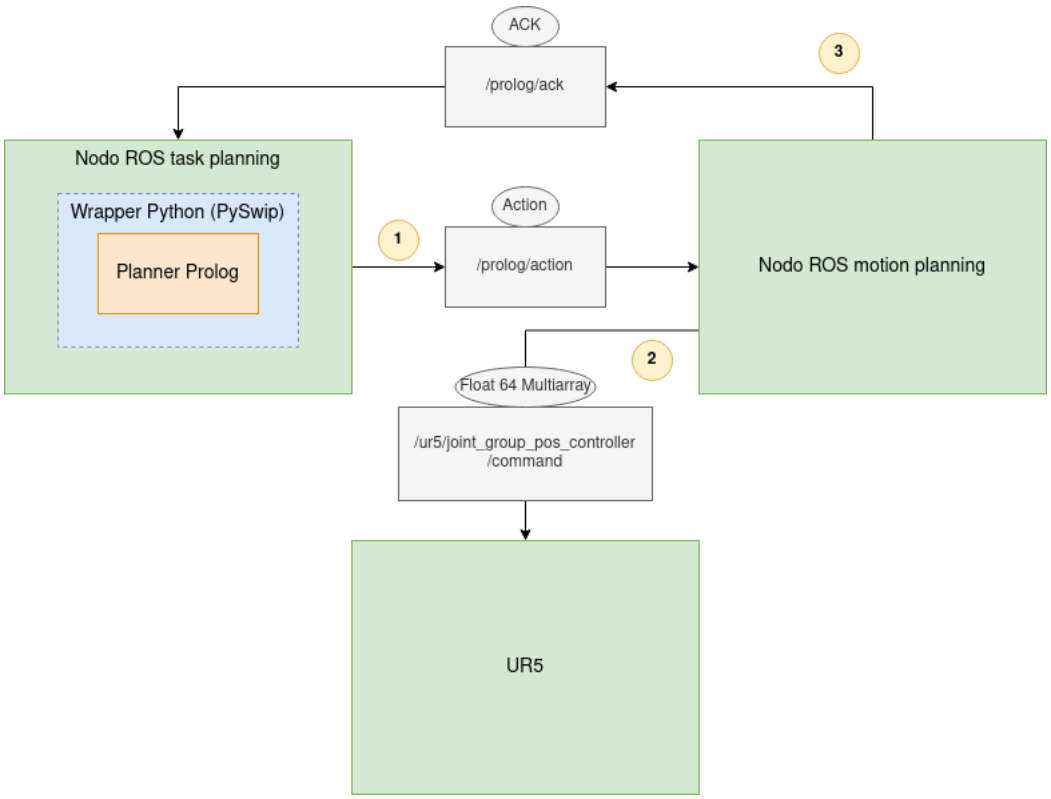

Figura 4.3: Architettura ROS. In verde ci sono i nodi mentre in grigio chiaro i topic con i relativi messaggi.

realizzarlo, io ho scelto di utilizzare una strategia relativamente semplice da eseguire ma allo stesso tempo completa per tutti i movimenti che il robot deve compiere. Essa è composta principalmente dagli algoritmi di cinematica e dalla interpolazione cubica di punti nello spazio. Partiamo descrivendo i primi due concetti, cinematica diretta e inversa, per poi passare alla generazione della traiettoria. La *cinematica diretta* è un problema che consiste nel trovare la posizione e l'orientamento dell'end effector[1] di un robot in funzione delle posizioni dei suoi giunti. Questa solitamente viene utilizzata per localizzare l'end effector nello spazio, io, però, non l'ho mai utilizzata siccome conoscevo a priori la sua configurazione di partenza dato che è definita dall'utente. Ho introdotto questo concetto solamente per completezza e per capire meglio la cinematica inversa. Questa, come suggerisce il nome, svolge il compito opposto, ovvero trovare le posizioni dei giunti in funzione della posizione e dell'orientamento del end effector del robot. Di questa ne ho fatto ampio uso nel mio nodo siccome mi permetteva di trovare la configurazione dei giunti in funzione della posizione del blocco da afferrare. Questo problema non ha una unica soluzione ma, nel nostro caso, ne ha 8. Quindi ho adottato la strategia di scelta della soluzione in base alla distanza rispetto alla configurazione precedente. Questo mi ha permesso di avere una continuità nei movimenti del robot. La distanza veniva calcolata attraverso la norma delle due configurazioni e la soluzione scelta era quella che minimizzava la distanza.

Minimizzata la distanza e trovata la configurazione dei giunti, il robot doveva eseguire il movimento. Per fare ciò ho utilizzato una interpolazione cubica di punti nello spazio. Questa serve a trovare tutte le configurazioni necessarie per arrivare da un punto A ad un punto B in un certo intervallo di tempo e compiendo un movimento fluido. Per la sua realizzazione mi sono servito del paper di Shuang Fang et al. [2], in particolare la parte in cui descriveva la creazione di, appunto, una traiettoria cubica.

Avendo la cinematica e la traiettoria funzionante ho dovuto solamente pensare ad un modo di gestire le azioni ottenute dal task planner. L'azione *move* l'ho voluta gestire con un semplice movimento con dei punti di controllo intermedi. In particolare ho voluto strutturare l'azione nel seguente modo:

1. Il braccio robotico approccia il blocco da afferrare stando ad una altezza prefissata.

[1] End effector: parte finale di un manipolatore robotico, nel mio caso un gripper a due dita.

2. Il braccio robotico si abbassa fino a toccare il blocco e lo afferra.

3. Il braccio torna a posizione 1.

4. Il braccio approccia la posizione finale ad una altezza prefissata.

5. Il braccio si abbassa fino a toccare il piano e rilascia il blocco.

6. Il braccio torna a posizione 4.

Questa è la strategia che ho voluto adottare per l'istruzione *move*.

Per l'istruzione *rotate*, invece, il movimento dipende dall'orientamento di partenza del blocco. Ogni orientamento avrà la sua strategia per ruotare in posizione corretta, cioè con la base verso il tavolo, il blocco.

Calcolate le posizioni dei giunti e la traiettoria, il nodo di motion planning invia i comandi al robot per eseguire il movimento. Eseguito quest'ultimo, il nodo di motion planning invia un messaggio al task planner per notificare l'esecuzione dell'azione e per permettere al task planner di proseguire con la prossima istruzione (punto 3 in figura 4.3).

La parte di motion planning dipende dalle istruzioni generate dal task planner, vediamo quindi ora come queste sono estrapolate dal programma prolog e come sono inviate al nodo di motion planning

Task planner

Il nodo di task planning ha come obiettivo quello di interrogare la base di conoscenza e di estrapolare le azioni da eseguire per portare a termine il programma prolog. Come visto prima, questo nodo è scritto in python e utilizza la libreria *pySwip* per interrogare la base di conoscenza. Ricevuto il risultato questo viene processato tramite la libreria di espressioni regolari *re* per estrapolare le azioni da eseguire. Estrapolate le azioni, queste vengono inviate con il messaggio appositamente creato al nodo di motion planning. Il messaggio in questione si chiama *action* e contiene il nome dell'azione da eseguire e i parametri necessari per eseguirla cioè le posizioni iniziali e finali e l'orienatamento iniziale. Inviato il messaggio, il nodo di task planning si mette in attesa di una risposta dal nodo di motion planning per poter proseguire con la prossima azione.

4.3 Difficoltà incontrate

Durante lo sviluppo della simulazione ho incontrato diverse difficoltà, molte se non la totalità di queste legate al simulatore gazebo. In questa sezione descriverò le principali difficoltà incontrate e come le ho risolte.

Durante lo sviluppo mi sono scontrato con due grandi imprevisti, tutti e due legati al motore di simulazione Gazebo. Il primo era legato alla simulazione del *grasping*, ovvero l'azione di afferrare un oggetto con il robot. Questo è un problema ben noto alla comunità di ricerca e sono stati sviluppati diversi metodi per risolverlo. Uno tra questi è l'utilizzo di un plugin chiamato *gazebo_grasp_plugin* che permette di simulare il grasping di un oggetto con il robot. Questo crea un *link* dinamico tra il robot e l'oggetto che permette di simulare il movimento di afferramento. Questo verrà creato quando il robot applicherà una certa forza al blocco e verrà eliminato quando questa forza verrà rimossa. Un accorgimento che ho dovuto prendere è stato quello di modificare i parametri fisici dei blocchetti aumentandone il loro peso. Così facendo il plugin riusciva a simulare in modo corretto il movimento di afferramento.

Una seconda difficoltà incontrata è sorta durante il momento di creazione della torre. I blocchi, se pur venivano posizionati nel modo corretto uno sopra l'altro, non rimanevano in equilibrio e iniziavano a "vibrare" compenetrandosi. Questo perchè il contatto tra i blocchi non era modellato nel modo corretto. Per risolvere questo problema ho dovuto modificare il file *.sdf* del modello del blocco aggiungendo le seguenti righe nel tag *collision*:

```
<surface>
    <contact>
        <ode>
```

```
            <min_depth>0.008</min_depth>
            <max_vel>0.0</max_vel>
        </ode>
    </contact>
</surface>
```

Il parametro *min_depth* indica la profondità minima di penetrazione del contatto. In altre parole, se due oggetti si scontrano, questo valore stabilisce la distanza minima tra le superfici prima che venga considerato un contatto valido. Il parametro *max_vel* invece indica la velocità massima di penetrazione. Se due oggetti si scontrano con una velocità maggiore di questo valore, il contatto non verrà considerato valido.

Questi fix sembrano essere sufficienti per risolvere il problema, un'altro modo per risolvere questi problemi sarebbe stato di utilizzare un altro simulatore come *CoppeliaSim* che permette di simulare in modo più preciso il contatto tra gli oggetti.

In questo capitolo abbiamo visto come ho strutturato la simulazione e come ho risolto i problemi che ho incontrato durante lo sviluppo. In questo momento il caso di studio è stato illustrato nella sua interezza, per maggiori approfondimenti e per vedere il codice sorgente si rimanda al repository [8]. Il prossimo capitolo trarrà le conclusioni rispetto a questo progetto proponendo anche dei possibili scenari di sviluppo futuri.

5 Conclusione

Bibliografia

[1] I. Bratko. *Prolog Programming for Artificial Intelligence*. International computer science series. Addison Wesley, 2001.

[2] Shuang Fang, Xianghua Ma, Yang Zhao, Qian Zhang, and Yaoyao Li. Trajectory planning for seven-dof robotic arm based on quintic polynormial. In *2019 11th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, volume 2, pages 198–201, 2019.

[3] Michele Focchi, Francesco Roscia, and Claudio Semini. Locosim: an open-source cross-platform robotics framework, 2023.

[4] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[5] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V Hermenegildo, Jose F Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, et al. Fifty years of prolog and beyond. *Theory and Practice of Logic Programming*, 22(6):776–858, 2022.

[6] Robert A Kowalski. Predicate logic as a programming language. *Proceedings of IFIP*, pages 569–574, 1974.

[7] Paul J. Krause. The art of prolog—second edition by leon sterling and ehud shapiro, mit press, cambridge, ma1994, pp. 509, £19.95 (paperback), £44.94 (hardback), isbn 0-262-19338-8. *The Knowledge Engineering Review*, 10(4):411–411, 1995.

[8] Davide De Martini. Prolog planner project. https://github.com/davidedema/prolog_planner. Repository GitHub del progetto.

[9] Fiorini Paolo Meli Daniele, Nakawala Hirenkumar. Logic programming for deliberative robotic task planning. *Artificial Intelligence Review*, 2023.

[10] Aram Mustafa and Tarik Rashid. Design an arm robot through prolog programming language. *Advances in Robotics and Automation*, 1:1–6, 01 2013.

[11] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 3, 01 2009.

[12] ROS. Ros wiki. http://wiki.ros.org/. Pagina del wiki di ROS.

[13] Yüce Tekol and contributors. PySwip v0.2.10, 2020.

[14] Unitn. Unitn collabora per watson. https://webmagazine.unitn.it/news/ateneo/7380/ad-unitrento-il-nuovo-ibm-faculty-award. Pagina del webgazine di UniTn che documenta la notizia.