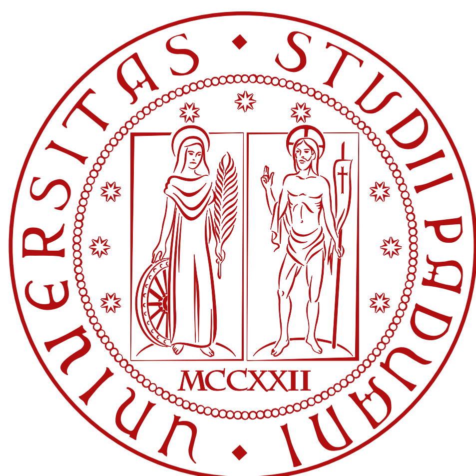

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Pianificare azioni intelligenti per robot assistivi via ROS-Plan

Relatrice
Prof.ssa Gloria Beraldo

Laureando
Tomasi Danny

ANNO ACCADEMICO 2022-2023

Data di laurea 29/09/2023

Sommario

I robot assistivi hanno la funzione di aiutare nell'arco della giornata un individuo che, per motivi quali vecchiaia, disabilità e non solo, non riesce ad essere indipendente.
Oltre a sostenere il paziente nella sua quotidianità, anche il caregiver ne beneficia, avendo un carico di lavoro minore, sia a livello fisico che psicologico.
Questa tesi ha lo scopo di studiare e testare ROS-Plan, un framework per la pianificazione automatica nel campo della robotica, per dotare il robot della capacità di pianificare una serie di azioni che permettano al robot di consegnare un farmaco al paziente, di garantire l'assunzione di tale farmaco e di ricordare al paziente di fare una chiamata al dottore.
In particolare, questa tesi vuole valutare le capacità di ROS-Plan in situazioni complesse, come ad esempio nel caso in cui la somministrazione del farmaco non abbia successo o nel caso in cui il paziente non chiami il dottore dopo aver ricevuto il promemoria.
In queste situazioni il sistema deve mantenere un comportamento cooperativo tra i moduli di pianificazione e di dispatch delle azioni, senza permettere sovrapposizioni e/o ritardi causati dalla necessità di ricalcolare il piano.
Infine, i risultati dei test delle demo hanno rivelato che le tempistiche della pianificazione dipendono, come atteso, dallo spostamento del paziente, ma il piano è stato riadattato coerentemente richiedendo circa 5 secondi.

Indice

1 Introduzione **1**

1.1 Robot Assistivi 2

1.2 Planning 3

1.3 Scopo della tesi 3

1.4 Struttura della tesi 4

2 Metodologia **5**

2.1 Robot Operating System 5

2.1.1 Nodi 6

2.1.2 Messaggi 6

2.1.3 Topic 6

2.1.4 Servizi 6

2.1.5 Parametri 7

2.1.6 Launch 7

2.1.7 Catkin 7

2.1.8 RViz 7

2.1.9 Gazebo 7

2.2 ROSPlan 7

2.2.1 Knowledge Base 9

2.2.2 Problem Interface 10

2.2.3 Planner Interface 10

2.2.4 Parsing Interface 11

2.2.5 Plan Dispatch 11

2.2.6 Action Interface 12

2.2.7 Sensing Interface 12

3 Sviluppo **15**

3.1 PDDL 15

3.1.1 Domain 15
3.1.2 Problem 18
3.2 Azioni Simulate 19
3.3 Coordinator 20
3.3.1 Aggiornamento della KB 20
3.3.2 Generazione del Piano 22
3.3.3 Esecuzione del Piano 23
3.3.4 Roslaunch 24
3.4 Introduzione di Oggetti Variabili 26
3.4.1 Settaggio Parametri 26
3.4.2 Patient Position Publisher 27
3.4.3 Sensing Interface Topics 28
3.4.4 Sensing Interface Services 30
3.5 Gestione dei Fallimenti 32

4 Analisi dei risultati ottenuti **35**

5 Conclusioni e Spunti per Lavori Futuri **39**
5.1 Conclusioni 39
5.2 Lavori Futuri 39

6 Appendici **41**
6.1 Nodo Coordinatore 41
6.2 Domain 52
6.3 Patient Position Publisher 56
6.4 Sensing Config 57
6.5 Sensing Functions 58

Bibliografia **63**

Capitolo 1

Introduzione

Nella società odierna, grazie al rapido progresso della medicina, la vita media si è allungata nonostante la vecchiaia, le disabilità e in generale qualsiasi condizione debilitante. Questo ha aumentato la porzione di popolazione che necessita di sostegno nell'arco della sua giornata. La figura che fornisce sostegno a questi individui prende il nome di caregiver.
Il caregiver è un individuo, un familiare, un amico, un volontario o un professionista specializzato (come in Figura 1.1, che assiste uno o più individui bisognosi durante la loro quotidianità [1].
È una definizione volutamente ampia dato che i compiti del caregiver possono essere molto diversi tra di loro, ad esempio potrebbe occuparsi della somministrazione di farmaci, di dare sostegno emotivo, di aiutare nelle faccende domestiche o anche di fungere da intermediario tra il paziente e professionisti sanitari, permettendogli di ricevere la miglior assistenza possibile.

Figura 1.1: Caregiver e paziente[2]

1.1 Robot Assistivi

La robotica è una disciplina che mira alla progettazione e alla costruzione di dispositivi in grado di svolgere attività in maniera autonoma o semi-autonoma, in modo tale da poter supportare o addirittura sostituire l'uomo nelle sue mansioni.
Un esempio di spicco è quello dei robot assistivi, cioè macchine sviluppate con lo scopo di fornire aiuto a persone con disabilità e/o problemi di autonomia nelle attività di ogni giorno[3] (Figura 1.2), il che porta ad alcuni vantaggi:

- il robot non è soggetto all'errore umano, dato per esempio dall'emotività; ciò garantisce affidabilità durante l'esecuzione di compiti critici

- potenzialmente può gestire una vasta gamma di situazioni, ad esempio nel caso in cui il paziente abbia problemi mnemonici o di deambulazione

- potrebbe essere in grado di ottenere dati più oggettivi sulla salute del paziente rispetto a quanto sarebbe possibile per un operatore umano.
  Questi dati potrebbero permettere al medico di avere un quadro generale più preciso

- permette al caregiver di delegare almeno una parte delle attività più semplici e meccaniche, in modo che si possa concentrare su altre attività finalizzate all'aumento della qualità della vita del paziente

- oltre ad alleviare il carico di lavoro fisico, un robot assistivo ridurrebbe l'impatto psicologico ed emotivo sul caregiver

Lo sviluppo di robot assistivi però non è banale; è essenziale, ad esempio, che il robot sia in grado di muoversi in autonomia e abbia una conoscienza dettagliata della disposizione di stanze e oggetti.
Inoltre l'ambiente in cui il robot dovrebbe idealmente operare è dinamico.
Il paziente potrebbe non seguire le istruzioni e/o spostarsi, imponendo al sistema di ricalibrare il piano d'azione.

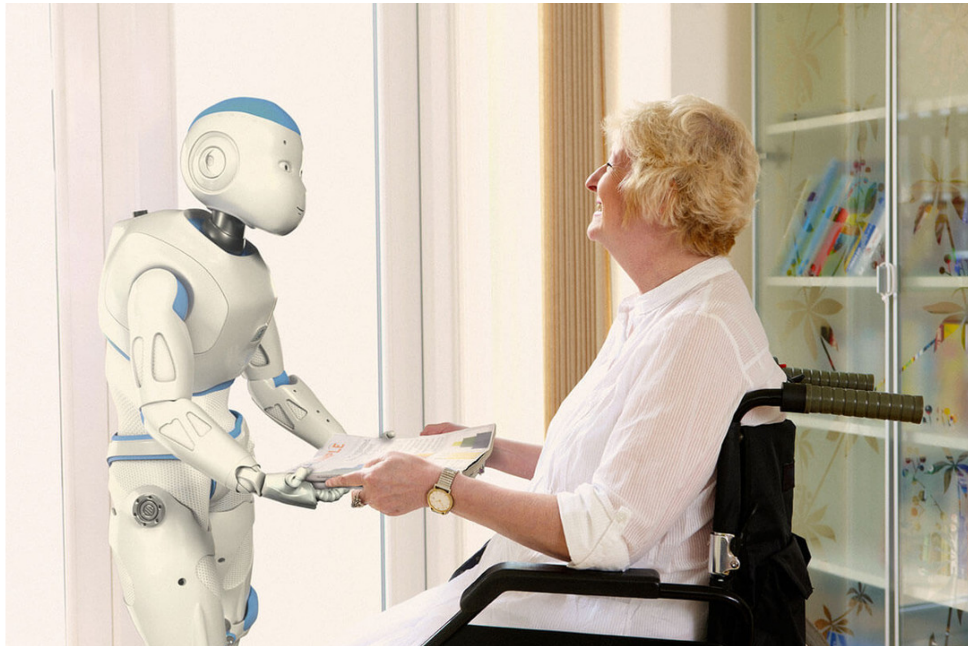

Figura 1.2: Robot assistivo[4]

1.2 Planning

Un modo di approcciarsi ai problemi citati precedentemente è l'utilizzo dei pianificatori.
La pianificazione è uno strumento utilizzato in molti ambiti della robotica e dell'automazione e prevede di stabilire una serie di azioni volte al raggiungimento di uno o più goal in modo efficiente, gestendo le risorse disponibili e rispettando i vincoli imposti[5].
Da questa definizione si può intuire la necessità di descrivere in maniera accurata l'ambiente in cui il robot si trova e le regole che lo governano.
Per rendere comprensibili queste informazioni al pianificatore si utilizzano dei linguaggi ad hoc. Nel nostro caso il linguaggio sarà il Planning Domain Definition Language (PDDL), uno standard nell'ambito della pianificazione automatica.

1.3 Scopo della tesi

Lo scopo di questa tesi è di utilizzare un framework chiamato ROSPlan su un sistema ROS (Robot Operating System) per controllare il comportamento di un robot assistivo all'interno di una simulazione.
L'obiettivo è di progettare un sistema che permetta al robot di eseguire alcuni compiti di assistenza al paziente.
I compiti scelti come esempi sono:

- prendere e consegnare un farmaco al paziente;
- garantire che il paziente assuma il farmaco;
- ricordare al paziente di chiamare il medico e assicurarsi che lo faccia.

Il primo passo sarà riuscire a descrivere il contesto e lo stato in cui il robot si troverà, definire alcune azioni possibili e specificare gli obiettivi da raggiungere.
Poi analizzeremo le fasi di pianificazione ed esecuzione del piano e, in un secondo momento, la relativa gestione in caso di fallimento. Infine analizzeremo i risultati ottenuti tramite varie esecuzioni dell'applicazione.

1.4 Struttura della tesi

Nel Capitolo 2 sarà presente una descrizione dei framework utilizzati, ovvero ROS e ROSPlan, indicando gli strumenti che mettono a disposizione, in particolare quelli utilizzati per questo progetto.
Successivamente, nel Capitolo 3, sarà illustrata l'implementazione dei vari moduli necessari alla nostra simulazione; per prima cosa la costruzione del file di dominio e del problema, seguita dall'analisi del coordinatore delle varie operazioni, dalla pianificazione all'esecuzione delle azioni. Una volta costruito il core ci focalizzeremo sulla stesura del codice relativo ai fallimenti e alla loro gestione.
Nel Capitolo 4 sarà presente una tabella con le tempistiche ottenute nella fase di test, associate al numero di ripianificazioni necessarie a causa dei fallimenti, e un paragrafo di valutazione dei risultati ottenuti.
Il Capitolo 5 conclude con alcune considerazioni sull'applicazione di ROS-Plan all'interno dello scenario assistivo esplorato in questa tesi.
Infine, nelle appendici, sarà presente il codice implementato per questo progetto.

Capitolo 2

Metodologia

2.1 Robot Operating System

Il Robot Operating System[1] è un framework open-source adibito alla creazione di software per applicazioni robotiche.
A dispetto di quanto suggerisca il nome, non è un vero e proprio sistema operativo, è invece un middleware che fornisce un vasto assortimento di strumenti e librerie per agevolare lo sviluppo di applicazioni capaci di generare comportamenti più o meno complessi di sistemi robotici.
La potenza di ROS risiede nella sua struttura modulare, che lo rende molto flessibile e facilmente scalabile, e nel suo continuo miglioramento dovuto all'ampia community[6].
ROS è stato concepito in seguito ad alcuni prototipi interni sviluppati dalla STanford AI Robot (STAIR) e da Personal Robots (PR) della Stanford University a partire dal 2000.
Nel 2007 il laboratorio di ricerca in robotica e incubatore tecnologico Willow Garage estese ulteriormente questi strumenti fornendo un insieme di implementazioni ben collaudate, battezzandole con il nome di ROS.
In generale, ROS segue la filosofia Unix, presenta perciò le seguenti caratteristiche:

- tools-based: consiste in molte piccole applicazioni, invece di avere una struttura monolitica.
- peer-to-peer: le applicazioni da cui è costituito interagiscono tra di loro tramite scambio di messaggi.
  Questi viaggiano in linea diretta da un'applicazione all'altra senza l'ausilio di una routine centrale.
- multi-language: ROS è compatibile con più linguaggi di programmazione, in modo da soddisfare esigenze differenti per diversi scopi.

[1] http://www.ros.org

- open-source: il codice sorgente di ROS è liberamente visionabile, il che permette alla sua community di aggiornare ed estendere il software.
  L'utilizzo di ROS è libero sia per progetti professionali che amatoriali.

La versione utilizzata per questo progetto è ROS Melodic.

2.1.1 Nodi

I processi indipendenti all'interno di ROS prendono il nome di nodi ed ognuno di loro esegue una specifica funzionalità.
Essi possono essere sviluppati e testati in modo autonomo. Come è stato detto prima, i nodi possono comunicare tramite messaggi.
I nodi hanno inoltre la possibilità di interagire tra di loro da macchine diverse, permettendo di distribuire il sistema ottenendo un miglioramento in termini computazionali. Questa particolare caratteristica rende possibile creare sistemi più complessi che richiedono una capacità di calcolo maggiore.

2.1.2 Messaggi

I messaggi sono delle strutture dati utilizzate per la comunicazione tra i nodi. ROS mette a disposizione un'ampia varietà di messaggi standard in modo tale da poter coprire molte tipologie di dati.
I nodi possono pubblicare dei messaggi (in questo caso prendono il nome di publisher) o riceverli (subscriber).

2.1.3 Topic

I messaggi viaggiano attraverso dei canali di comunicazione denominati topic.
I nodi possono iscriversi a dei topic, dove altri nodi publicheranno informazioni.
Questa architettura permette una gestione asincrona dei dati, portando ad avere una struttura decentralizzata, dato che non sarà necessario nessun modulo che gestisca la comunicazione.

2.1.4 Servizi

Non sempre la comunicazione asincrona fornita dai topic è la scelta migliore, per questo su ROS esistono anche i servizi, che seguono un modello client-server; un nodo (server) si mette a disposizione per eseguire determinate routine e altri nodi (client) possono richiedere tali routine, aspettando la comunicazione dell'esito dell'operazione dal server prima di continuare la loro esecuzione.

2.1.5 Parametri

ROS permette di mantenere in un server apposito delle variabili, chiamate parametri, e di manipolarle a piacimento.
L'utilità di questo strumento sta nella possibilità di avere rapido accesso ad informazioni utili a qualsiasi modulo lo richieda.

2.1.6 Launch

ROS presenta un sistema di lancio che permette di avviare e configurare un insieme di nodi in modo coordinato, senza doverlo fare manualmente.
Questo genere di file permette anche il lancio di ulteriori file .launch.
Sono scritti in XML.

2.1.7 Catkin

Catkin è il sistema di compilazione utilizzato da ROS.
Esso fornisce strumenti per la gestione dei progetti, la compilazione del codice sorgente dei vari nodi e le dipendenze tra i pacchetti.

2.1.8 RViz

RViz è uno strumento di visualizzazione tridimensionale che permette di stampare a video l'ambiente di lavoro in tempo reale.

2.1.9 Gazebo

Gazebo permette di simulare di testare i propri progetti in ambienti complessi senza il bisogno di eseguirli su hardware fisico.

2.2 ROSPlan

Rosplan[2] è una libreria progettata per la pianificazione automatica nell'ambito della robotica.
La stretta integrazione con ROS permette di sviluppare e testare software destinato a sistemi robotici in modo agevole[7].
Presenta una serie di nodi che incapsulano le fasi di generazione del problema, pianificazione

[2] https://kcl-planning.github.io/ROSPlan/

ed esecuzione del piano generato, ma permette anche l'utilizzo di strumenti esterni (Figura 2.1). La sua struttura comprende cinque elementi fondamentali:

- Knowledge Base
- Problem Interface
- Planner Interface
- Parsing Interface
- Plan Dispatcher

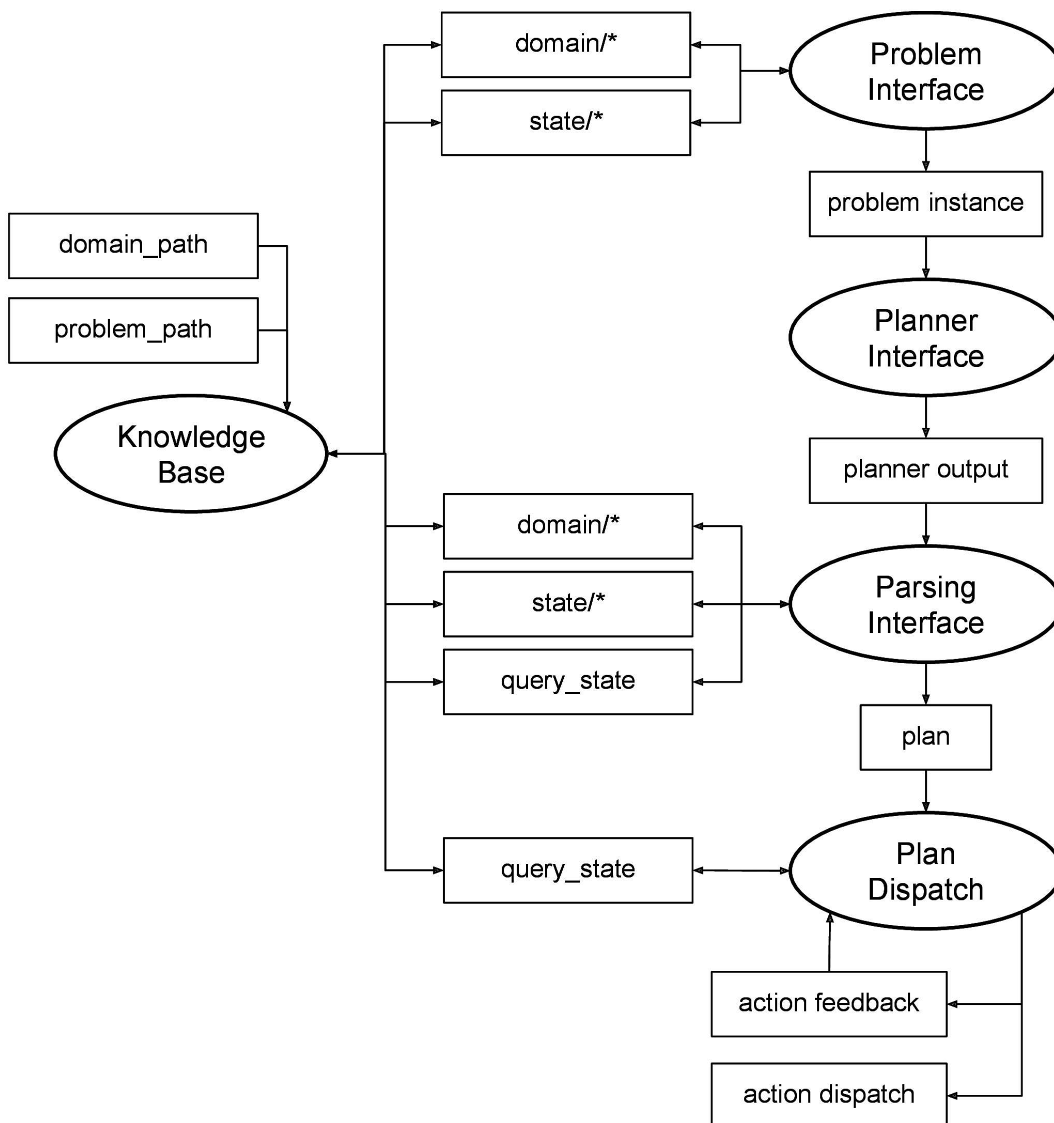

Figura 2.1: Struttura di ROSPlan[8]

2.2.1 Knowledge Base

La Knowledge Base (KB) è un database che ha lo scopo di conservare le informazioni relative all'ambiente in cui il robot dovrà operare, permettendo alle altre componenti di rosplan di generare un piano d'azione (Figura 2.2).
La KB è caratterizzata da un aggiornamento dinamico tramite messaggi inviati dai sensori, ad esempio in seguito alle azioni svolte o a delle rilevazioni automatiche.
Le informazioni contenute nella KB rientrano in tre macro categorie:

- lo stato attuale, che può comprendere la posizione di oggetti o del sistema robotico stesso, lo stato dei sensori e lo stato delle azioni in atto;
- le azioni possibili, i requisiti necessari per attuarle e i loro effetti;
- i goal da raggiungere.

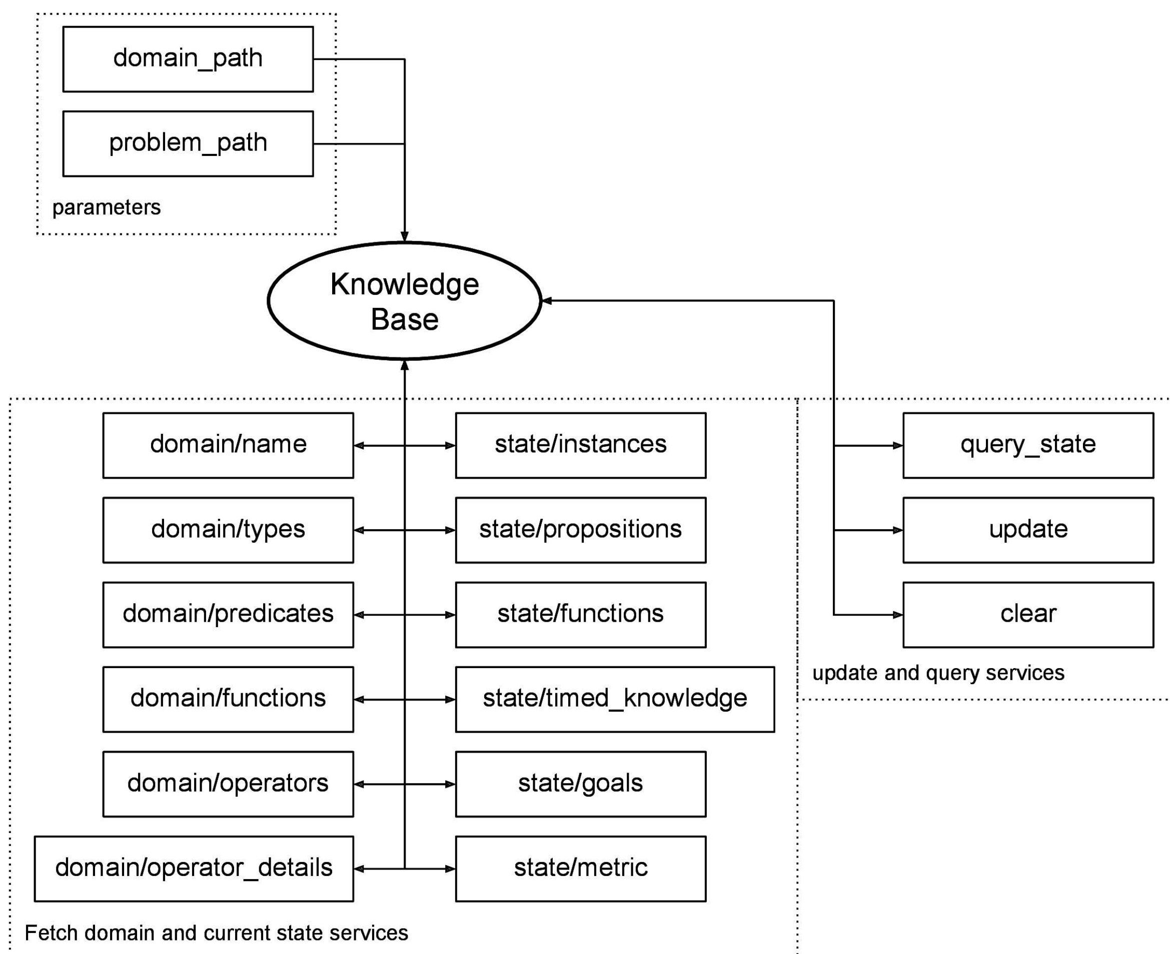

Figura 2.2: Knowledge Base[9]

2.2.2 Problem Interface

La Problem Interface (Figura 2.3) si occupa di generare l'istanza del problema in modo da renderlo fruibile al sistema di pianificazione.
In pratica questa componente estrapola le informazioni dai sensori e tramite interrogazioni della KB, per creare un file PDDL che verrà successivamente pubblicato su un topic apposito.

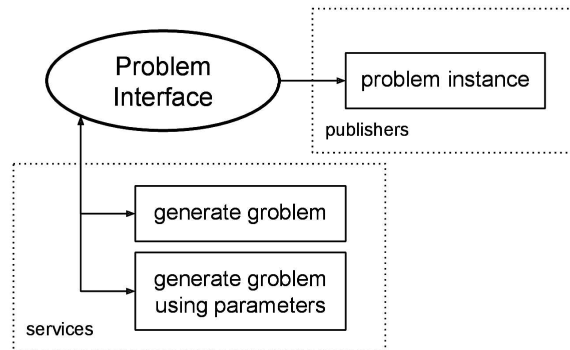

Figura 2.3: Problem Interface[10]

2.2.3 Planner Interface

La Planner Interface (Figura 2.4) è un wrapper per un AI Planner, che ha il compito di stilare il miglior piano d'azione per il robot.
Il piano viene ottenuto chiamando il servizio /rosplan_planner_interface/planning_server_params.
È necessario fornire il dominio e l'istanza del problema generati nella fase precedente; inoltre richiede di specificare una linea di comando, nel nostro caso:

```
timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM
```

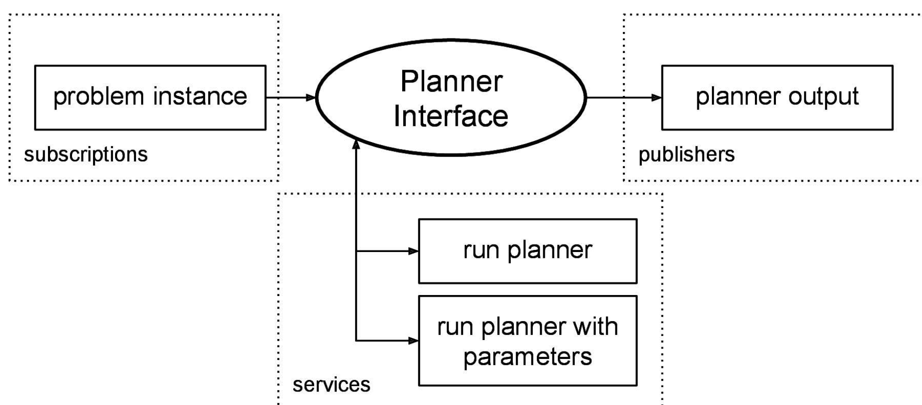

Figura 2.4: Planner Interface[11]

2.2.4 Parsing Interface

La Parsing Interface (Figura 2.5) ha lo scopo di tradurre il piano generato dalla Planner Interface in una rappresentazione eseguibile dello stesso.
In sostanza fa da tramite tra la Planning Interface e il Plan Dispatch. Può eseguire il suo compito sia prendendo in input ciò che riceve da un topic che da un file.

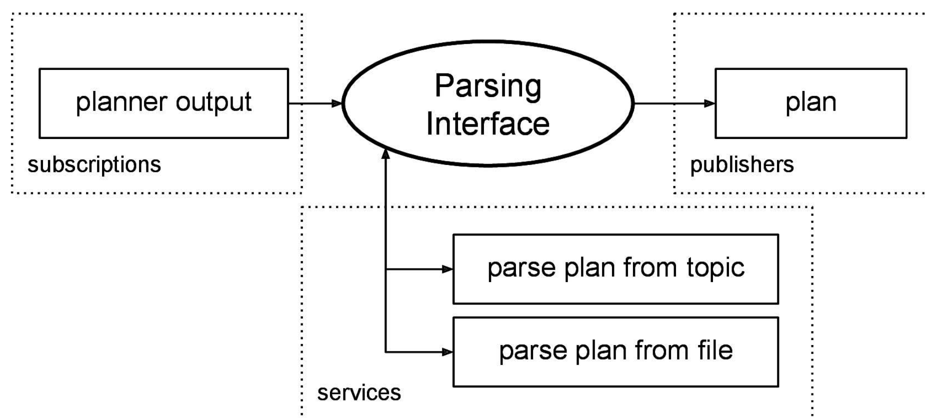

Figura 2.5: Parsing Interface[12]

2.2.5 Plan Dispatch

Quest'ultima componente (Figura 2.6) è responsabile dell'esecuzione delle azioni del piano, in quanto permette di chiamare i processi adibiti al loro svolgimento.
Dall'inizio alla fine dello svolgimento del piano il Plan Dispatch monitora in modo continuo l'ambiente e il sistema per rilevare potenziali deviazioni e intervenire in modo opportuno.
Comunica anche con la KB aggiornando lo stato corrente.

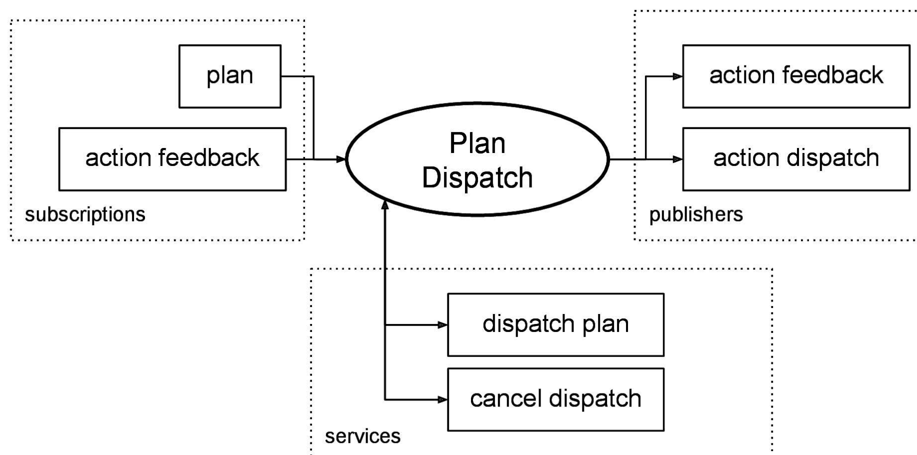

Figura 2.6: Plan Dispatch[13]

Esistono tuttavia ulteriori nodi responsabili dell'interazione del sistema con l'esterno, ovvero la Action Interface e la Sensing Interface.

2.2.6 Action Interface

La Action Interface è iscritta al topic pubblicato dal Plan Dispatch, chiamato Action Dispatch, e ha il dovere di far eseguire al sistema le azioni previste dal piano.
Permette anche di eseguire azioni simulate (Figura 2.7).

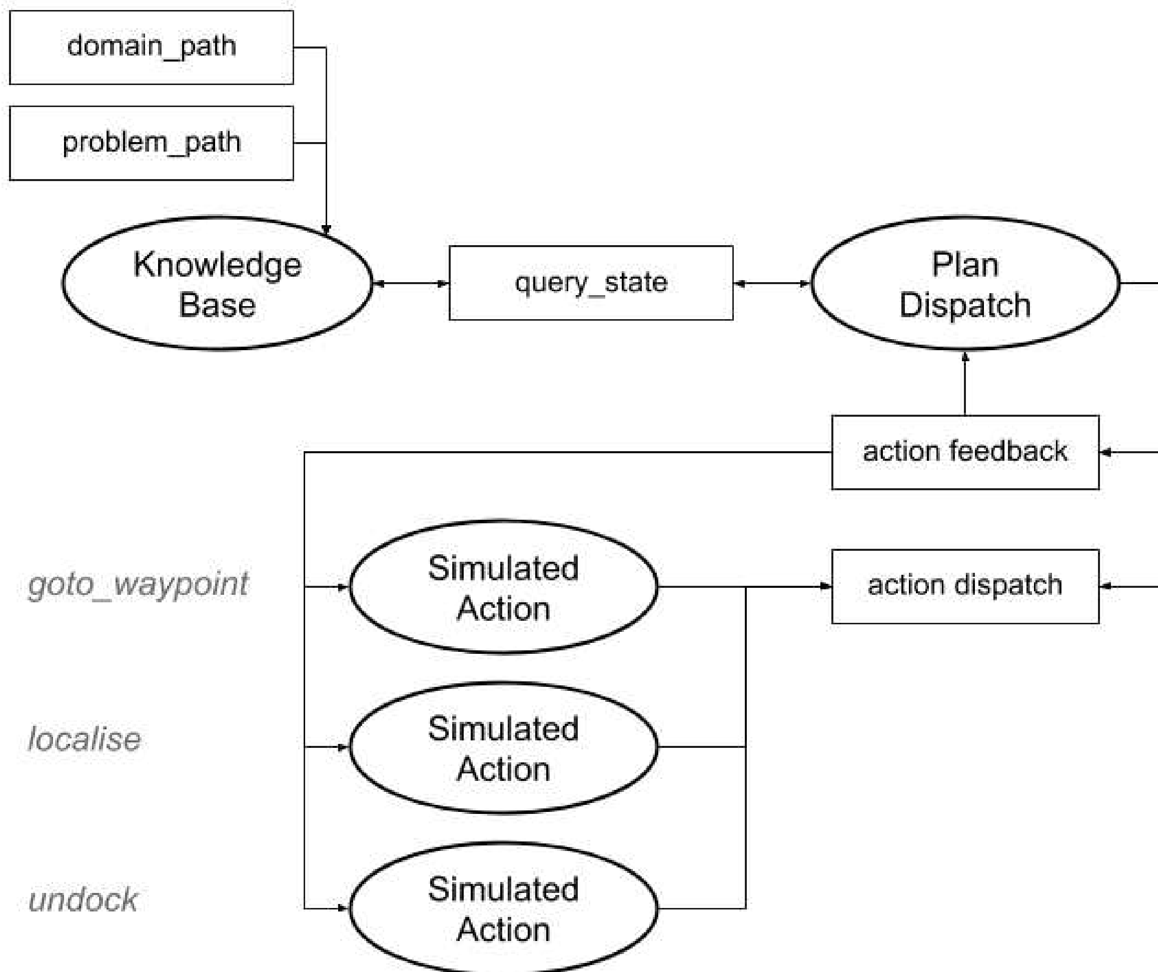

Figura 2.7: Simulated Actions[14]

2.2.7 Sensing Interface

Questa interfaccia (Figura 2.8) è responsabile della percezione dell'ambiente circostante per mezzo dei sensori e conseguentemente al continuo aggiornamento della KB.
Grazie alle sue caratteristiche la Sensing Interface permette anche di gestire le incertezze, veri-

ficando l'efficacia delle azioni eseguite.
Per il raggiungimento dei suoi scopi questa componente permette di modificare i predicati definiti nel dominio ascoltando i topic o di generare chiamate periodiche ai servizi che possono fornirci informazioni utili a tali modifiche.
Nel caso siano necessarie operazioni più o meno complesse prima di poter produrre il nuovo valore dei predicati, è possibile specificare delle funzioni che possano reagire adeguatamente in base ai dati ottenuti da topic o chiamate a servizio.

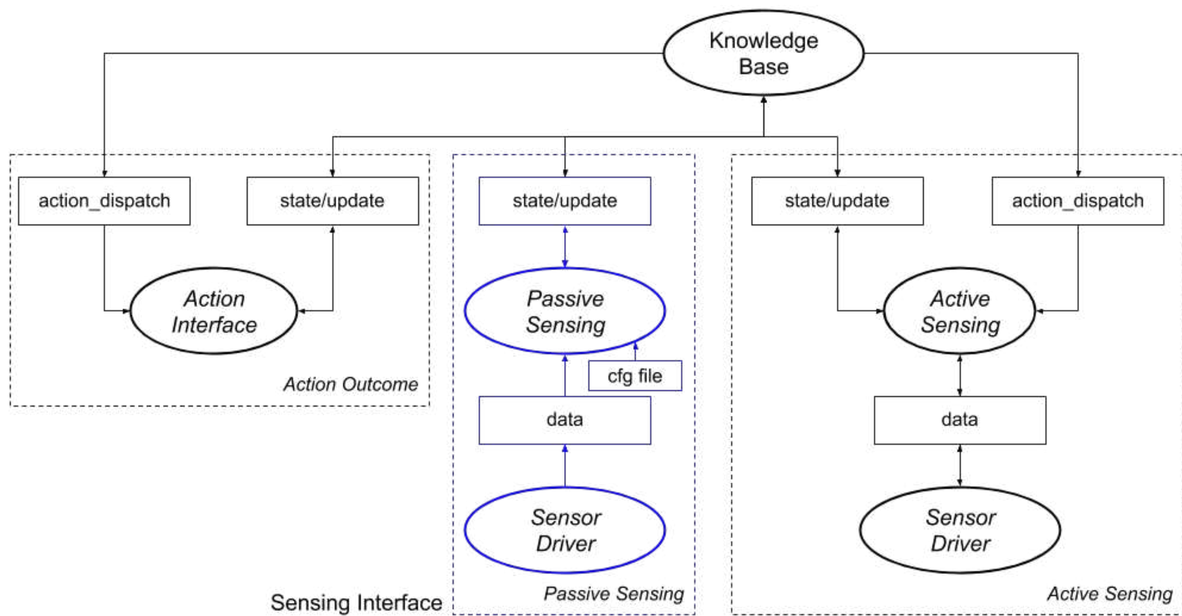

Figura 2.8: Sensing Interface[15]

Capitolo 3

Sviluppo

Per questo progetto è necessario sviluppare un'applicazione che controlli il robot assistivo all'interno di una simulazione; il robot deve garantire la somministrazione di un farmaco al paziente e deve ricordargli di fare una chiamata con il medico.
In questo capitolo verranno illustrati i passaggi per produrre tale applicazione, partendo dalla creazione dei file necessari alla prima fase, ovvero la generazione del problema.
Per la realizzazione ho preso spunto dalla demo rosplan_stage_exploration_demo, dato che il file di launch era provvisto dei nodi necessari al movimento e alla visualizzazione dello stage.

3.1 PDDL

I file di cui dovrà disporre la Problem Interface presentano l'estensione PDDL.
Il Planning Domain Definition Language (PDDL) è un linguaggio di specifica che ci permette di descrivere in modo formale la struttura del mondo del problema attraverso alcuni costrutti.
Per analizzare la struttura di ognuno analizzeremo i file utilizzati in questo progetto.

3.1.1 Domain

Il file domain contiene le informazioni del mondo e le azioni che il sistema può eseguire. È suddiviso in blocchi; di seguito analizzeremo quelli necessari per questa simulazione.

Types

Il primo costrutto permette di definire le classi di oggetti con cui il robot potrà interagire.
Per i nostri scopi sono necessari gli oggetti waypoint, robot, patient e medicine.

```
1     (:types
```

```
        waypoint
        robot
        patient
        medicine
    )
```

Listing 3.1: Types.

Predicates

Specificati i tipi possiamo definire i predicati, ovvero delle tuple di uno o più oggetti il cui valore può essere true o false.
In accordo ai valori assunti dai predicati si definisce lo stato in cui ci troviamo e quindi il sistema è in grado di capire le azioni che è possibile compiere.
I predicati sono provvisti di un nome e sono seguiti da una lista di oggetti di cui è necessario specificare il tipo.

```
    (:predicates
      ;positions
      (dock_at ?wp - waypoint)
      (robot_at ?v - robot ?wp - waypoint)
      (patient_at ?p - patient ?wp - waypoint)
      (medicine_at ?m - medicine ?wp - waypoint)
      ;robot flags
      (undocked ?v - robot)
      (docked ?v - robot)
      (localised ?v - robot)
      (robot_has_medicine ?v - robot ?m - medicine)
      (can_grab ?v - robot)
      (has_tried_to_give ?v - robot ?p - patient ?m - medicine)
      (has_told_to_take ?v - robot ?p - patient ?m - medicine)
      (has_told_to_call ?v - robot ?p - patient)
      ;patient flags
      (has_medicine ?p - patient ?m - medicine)
      (has_to_take ?p - patient ?m - medicine)
      (has_taken ?p - patient ?m - medicine)
      (has_to_call ?p - patient)
      (has_called ?p - patient)
    )
```

Listing 3.2: Predicates.

Nel nostro caso i primi quattro predicati servono ad indicare la posizione dei vari oggetti; nello specifico della basetta di ricarica, del robot, del paziente e della medicina. Seguono delle

flags per il robot che ci informano se è collegato o meno alla basetta, se ha eseguito l'azione localize, se ha preso la medicina e può prendere altre cose e infine se ha provate a svolgere i tre compiti che gli sono stati assegnati. Le flags del paziente ci servono per sapere se ha ricevuto la medicina, se deve prenderla e se l'ha fatto. In modo simile gli ultimi due predicati indicano se il paziente deve chiamare il dottore e se lo ha già fatto.

Functions

Le funzioni assomigliano ai predicati in termini di struttura, e come loro rappresentano lo stato attuale, ma sono provvisti di valori numerici e non booleani.

```
(:functions
  (distance ?a ?b - waypoint)
  (charge ?v - robot)
)
```

Listing 3.3: Functions.

Actions

Le azioni definiscono una trasformazione dello stato del mondo.
Questa trasformazione rappresenta un'azione realmente eseguibile dal sistema durante l'esecuzione del piano.
La tipologia standard fornita dal linguaggio PDDL è composta di tre parti: parametri, requisiti ed effetti.
Nei parametri è necessario specificare gli oggetti coinvolti nello svolgimento dell'azione, i requisiti sono una disgiunzione e/o congiunzione di predicati da soddisfare per poter eseguire l'azione ed infine gli effetti presentano le modifiche che subiranno i predicati a causa del compimento dell'azione.
In questo progetto per simulare al meglio l'esecuzione del piano sono presenti le azioni durative che, oltre alle parti specificate prima, necessitano di una componente ulteriore, ovvero la durata dell'azione.
L'esempio riportato qui sotto definisce l'azione goto_waypoint che permette al robot di spostarsi tra un waypoint e l'altro.

```
;; Move to any waypoint, avoiding terrain
(:durative-action goto_waypoint
  :parameters (?v - robot ?from ?to - waypoint)
  :duration ( = ?duration (distance ?from ?to))
```

```
5         :condition (and
6           (at start (robot_at ?v ?from))
7           (at start (localised ?v))
8           (over all (undocked ?v))
9           (over all (> (charge ?v) 0)))
10        :effect (and
11          (at end (robot_at ?v ?to))
12          (at start (not (robot_at ?v ?from)))
13          (at end (decrease (charge ?v) (distance ?from ?to))))
14      )
```

Listing 3.4: Durative action.

Nei parametri sono presenti gli oggetti robot, waypoint di partenza e waypoint di arrivo dato che sono gli unici coinvolti durante lo svolgimento dell'azione.
La scrittura all'interno della sezione duration stabilisce che la distanza dipende dal valore della funzione distance, che definisce la distanza tra i due waypoint presi in causa.
Nelle condizioni viene specificato che all'inizio dell'azione il robot dovrà trovarsi nel waypoint di partenza e dovrà già aver compiuto l'azione localize. Inoltre, durante tutta la durata dell'azione il robot dovrà essere carico e scollegato dalla base di ricarica.
Infine gli effetti impongono che il robot non si troverà più nel waypoint di partenza, ma sarà arrivato al waypoint di arrivo. Avrà anche perso una quantità di carica pari alla distanza percorsa.

3.1.2 Problem

Il file problem è associato al dominio corrispondente e contiene lo scenario nel quale la simulazione avrà inizio e gli obiettivi che il sistema dovrà cercare di raggiungere.
La sua struttura, come il domain file, è a blocchi.

Objects

In questa sezione vengono definiti gli oggetti presenti nel mondo, di cui bisogna specificare il tipo, scegliendo tra quelli specificati nel dominio.

```
1     (:objects
2         kenny - robot
3         larry - patient
4         oki - medicine
5     )
```

Listing 3.5: Objects.

Nel nostro caso sarà presente un robot chiamato kenny, un paziente di nome larry e la medicina da somministrare sarà semplicemente un oki.

Init

Init descrive lo stato iniziale specificando il valore dei predicati.
Se i predicati non sono presenti all'interno di init vengono registrati come falsi, a meno che non sia specificato diversamente nel momento del lancio della knowledge; in questo caso assumono il valore uknown.

```
1     (:init
2         (robot_at kenny wp0)
3         (docked kenny)
4         (dock_at wp0)
5         (can_grab kenny)
6         (has_to_take larry oki)
7         (has_to_call larry)
8         (= (charge kenny) 30)
9     )
```

Listing 3.6: Init.

Nel nostro caso il robot parte da un waypoint chiamato wp0 ed è sulla base di ricarica.
Il livello delle sue batterie è settato su 30 (valore fittizio che potrebbe riferirsi alla percentuale per esempio) ed è specificato che il paziente deve prendere una medicina e deve chiamare il dottore.
Le informazioni mancanti, come la posizione della medicina, verranno aggiunte in un'altra fase, attraverso un altro metodo, mostrato nella Sezione 3.3.1.

Goal

L'ultima componente contiene gli obiettivi da raggiungere nella forma di congiunzione e/o disgiunzione di predicati.
Come per alcune informazioni dello stato iniziale, in questo progetto i goal verranno aggiunti tramite il metodo mostrato nella Sezione 3.3.1.

3.2 Azioni Simulate

L'azione goto_waypoint è già definita ma per quanto riguarda le altre avremo bisogno di aggiungerle una per volta nel file di launch.

Vediamo come è possibile farlo per l'azione dock:

```
<include file="$(find rosplan_planning_system)/launch/includes/
simulated_action.launch" >
    <arg name="pddl_action_name" value="dock" />
</include>
```

Listing 3.7: Dock simulated action.

Tramite queste istruzioni viene incluso il file di launch specifico per le azioni simulate e viene precisato il nome dell'azione che si vuole aggiungere.
Nel nostro caso non è necessario specificare altro, dato che a livello visivo durante l'esecuzione delle varie azioni (ad esclusione di goto_waypoint) il robot sarà fermo sul posto.

3.3 Coordinator

Ora che disponiamo delle componenti fondamentali per avviare il processo di pianificazione, necessitiamo di un nodo in grado di coordinare le varie operazioni.
A questo scopo il file di launch è dotato di uno script (nel nostro caso in python) che possa automatizzare i processi di generazione del problema e successivamente del piano, della chiamata al parser e al dispatcher.
Dobbiamo apportare delle modifiche affinchè possa aggiungere le informazioni mancanti alla KB (cioè i predicati dello stato iniziale e i goal menzionati nella sezione 3.1.2), settare i parametri utili, chiamare in ordine i servizi forniti dalle varie interfacce e gestire le situazioni impreviste.

3.3.1 Aggiornamento della KB

Alcune informazioni non possono essere semplicemente scritte nel blocco init del problem file perché non sono certe, come ad esempio la posizione del paziente o delle medicine.
In questi casi è possibile aggiornare la KB tramite codice; in particolare utilizzando la chiamata al servizio resa possibile dalla libreria rospy.
Il servizio in questione è /rosplan_knowledge_base/update e necessita di una richiesta compilata opportunamente; nell'esempio riportato qui sotto vediamo come è possibile farlo per specificare che il predicato patient_at è vero se riferito a larry e al waypoint 65.

```python
kus_patient = KnowledgeUpdateServiceRequest()
kus_patient.update_type = 0
kus_patient.knowledge.knowledge_type = 1
kus_patient.knowledge.attribute_name = 'patient_at'
```

```
    kv_patient1 = KeyValue()
    kv_patient1.key = 'p'
    kv_patient1.value = 'larry'
    kus_patient.knowledge.values.append(kv_patient1)
    kv_patient2 = KeyValue()
    kv_patient2.key = 'wp'
    kv_patient2.value = 'wp65'
    kus_patient.knowledge.values.append(kv_patient2)
    kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
   KnowledgeUpdateService)
    if not kuc(kus_patient):
        rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
   ())
```

Listing 3.8: KB update

Per i nostri scopi bastano quattro attributi: update_type, knowledge_type, attribute_name e values.
Il primo termine permette di specificare il tipo di aggiornamento vogliamo fare.

```
    ADD_KNOWLEDGE=0
    ADD_GOAL=1
    REMOVE_KNOWLEDGE=2
    REMOVE_GOAL=3
    ADD_METRIC=4
    REMOVE_METRIC=5
```

L'attributo knowledge_type indica il tipo di informazione che stiamo trattando.

```
    INSTANCE=0
    FACT=1
    FUNCTION=2
    EXPRESSION=3
    INEQUALITY=4
```

Gli ultimi due valori precisano il nome e il valore dell'informazione.
Potrebbe risultare utile un quinto attributo, denominato is_negative, che può assumere i valori 0 e 1; se non specificato il valore di default è 0 (e quindi l'informazione è vera).
Oltre ai predicati necessari a definire lo stato iniziale, attraverso questo metodo aggiungiamo anche i goal.

3.3.2 Generazione del Piano

La generazione del piano si divide in due fasi e sono state raccolte all'interno di questa funzione:

```
1  def generate_problem_and_plan():
2
3      rospy.loginfo("KCL: (%s) Calling problem generation" % rospy.
   get_name())
4      pg = rospy.ServiceProxy('/rosplan_problem_interface/
   problem_generation_server', Empty)
5      if not pg():
6          rospy.logerr("KCL: (%s) No problem was generated!" % rospy.
   get_name())
7
8      rospy.loginfo("KCL: (%s) Calling planner" % rospy.get_name())
9      pi = rospy.ServiceProxy('/rosplan_planner_interface/
   planning_server_params', PlanningService)
10     pi_response = pi(domain_path, problem_path, data_path,
   planner_command, True)
11
12     if not pi_response:
13         rospy.logerr("KCL: (%s) No response from the planning server."
   % rospy.get_name())
14         return False
15     if not pi_response.plan_found:
16         rospy.loginfo("KCL: (%s) No plan could be found." % rospy.
   get_name())
17         return False
18     else:
19         rospy.loginfo("KCL: (%s) Plan was found." % rospy.get_name())
20         return True
```

Listing 3.9: Plan generation.

La prima fase riguarda la generazione dell'istanza del problema tramite il servizio fornito dalla Problem Interface.
Questa operazione utilizzerà la KB, il domain e il problem file per produrre un ulteriore file PDDL che chiameremo instance.
In pratica avrà la stessa struttura del problem file ma sarà fornito anche delle informazioni aggiunte dal nodo coordinatore e da altri nodi, come le distanze tra i waypoint connessi.
Notiamo che per utilizzare il servizio non è necessario specificare i file PDDL presi in input; questo perché sono già indicati tra gli argomenti del nodo della Problem Interface presente nel file di launch.

Successivamente alla generazione del problema avviene la generazione del piano per mezzo della Planner Interface. In questo caso è necessario compilare la richiesta di planning fornendo, oltre al path del dominio e del problema, anche il data path (dove verrà pubblicato il piano) e la riga di comando per il planner, cioè:

```
timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM
```

Siccome questa interfaccia è solo un wrapper, nel file di launch è necessario specificare il planner che si intende utilizzare; nel progetto verrà utilizzato il pianificatore standard di ROSPlan.

3.3.3 Esecuzione del Piano

```python
def execute_plan():

    rospy.loginfo("KCL: (%s) Calling plan parser" % rospy.get_name())
    pp = rospy.ServiceProxy('/rosplan_parsing_interface/parse_plan', Empty)
    if not pp():
        rospy.logerr("KCL: (%s) The plan was not parsed!" % rospy.get_name())
        return

    rospy.sleep(3)

    rospy.loginfo("KCL: (%s) Calling plan execution" % rospy.get_name())
    rospy.set_param('dispatch_called', True)
    pd = rospy.ServiceProxy('/rosplan_plan_dispatcher/dispatch_plan', DispatchService)
    pd_response = pd()
    if not pd_response:
        rospy.logerr("KCL: (%s) No response from the dispatch server." % rospy.get_name())
        return False
    if not pd_response.goal_achieved:
        rospy.loginfo("KCL: (%s) The execution was not successful." % rospy.get_name())
        return False
    else:
        rospy.loginfo("KCL: (%s) Plan was executed." % rospy.get_name())
        return True
```

Listing 3.10: Plan execution.

Anche nella funzione di esecuzione del piano vengono chiamati due servizi, ovvero /rosplan_parsing_interface/parse_plan e /rosplan_plan_dispatcher/dispatch_plan.
La Parsing Interface prende in input il piano generato dalla Planning Interface attraverso il planner topic definito nel file di launch (denominato /rosplan_planner_interface/planner_output) e produce il piano tradotto per il dispatcher.
Il Plan Dispatch riceve le informazioni attraverso il topic /rosplan_parsing_interface/complete_plan e indirizza le azioni da svolgere ai nodi corretti.

3.3.4 Roslaunch

A questo punto possiamo provare a lanciare il progetto. Per farlo sarà necessario usare il comando:

```
    roslaunch <ros package> launcher.launch
```

Nel terminale compariranno varie informazioni. Il setup iniziale prevede il setting di tutti i parametri del sistema seguito dall'inizializzazione dei vari nodi.
Successivamente avviene l'avvio dei processi e l'attesa dei servizi richiesti dalle componenti del sistema.
A questo punto vengono inizializzate le componenti principali di rosplan (KB, Problem Interface, etc).
Una volta che la KB è stata avviata vengono registrate tutte le informazioni del sistema relative alla simulazione ad opera del roadmap server.
Quando avrà terminato cominceranno le operazioni del nodo coordinatore:

```
(/rosplan_knowledge_base) Adding fact (patient_at larry wp65, 0)
(/rosplan_knowledge_base) Adding fact (medicine_at oki wp20, 0)
(/rosplan_knowledge_base) Adding fact (has_tried_to_give kenny larry oki,
    1)
(/rosplan_knowledge_base) Adding fact (has_medicine larry oki, 1)
(/rosplan_knowledge_base) Adding fact (has_told_to_take kenny larry oki, 1)
(/rosplan_knowledge_base) Adding fact (has_taken larry oki, 1)
(/rosplan_knowledge_base) Adding mission goal (has_tried_to_give kenny
    larry oki)
(/coordinator) Calling problem generation
(/rosplan_problem_interface) (instance.pddl) Generating problem file.
(/rosplan_knowledge_base) Adding fact (has_called larry, 0)
(/rosplan_problem_interface) (instance.pddl) The problem was generated.
(/rosplan_planner_interface) Problem received.
(/coordinator) Calling planner
(/rosplan_planner_interface) (instance.pddl) Writing problem to file.
```

```
15 (/rosplan_planner_interface) (instance.pddl) Running: timeout 10 /home/
     danny/tesi/src/rosplan/rosplan_planning_system/common/bin/popf /home/
     danny/tesi/src/rosplan_demos/rosplan_stage_exploration_demo/pddl/domain.
     pddl /home/danny/tesi/src/rosplan_demos/rosplan_stage_exploration_demo/
     pddl/instance.pddl > /home/danny/tesi/src/rosplan_demos/
     rosplan_stage_exploration_demo/pddl/plan.pddl
16 (/rosplan_planner_interface) (instance.pddl) Planning complete
17 (/rosplan_planner_interface) (instance.pddl) Plan was solved.
18 (/coordinator) Plan was found.
19 (/rosplan_parsing_interface) Planner output received.
20 (/coordinator) Calling plan parser
21 (/rosplan_parsing_interface) Parsing planner output.
22 (/rosplan_plan_dispatcher) Plan received.
23 (/coordinator) Calling plan execution
24 (/rosplan_plan_dispatcher) Dispatching plan.
```

Per prima cosa vengono aggiunte alla knowledge base le informazioni mancanti relative allo stato iniziale (righe 1-6) e il goal (riga 7).
Successivamente viene fatta generare l'istanza del problema dalla Problem Interface.
A questo punto la Planner Interface riceve il problema e chiama il pianificatore.
Se il planner riesce a generare un piano la Parsing Interface lo traduce e infine il dispatcher riceve le istruzioni e comincia ad assegnare le azioni da compiere ai rispettivi moduli.
Si apriranno inoltre altre finestre, ovvero la esterel plan viewer (Figura 3.1), dove sarà possibile vedere il piano generato sottoforma di flowchart, e rviz che mostrerà lo stage dove opera il robot (Figura 3.2).

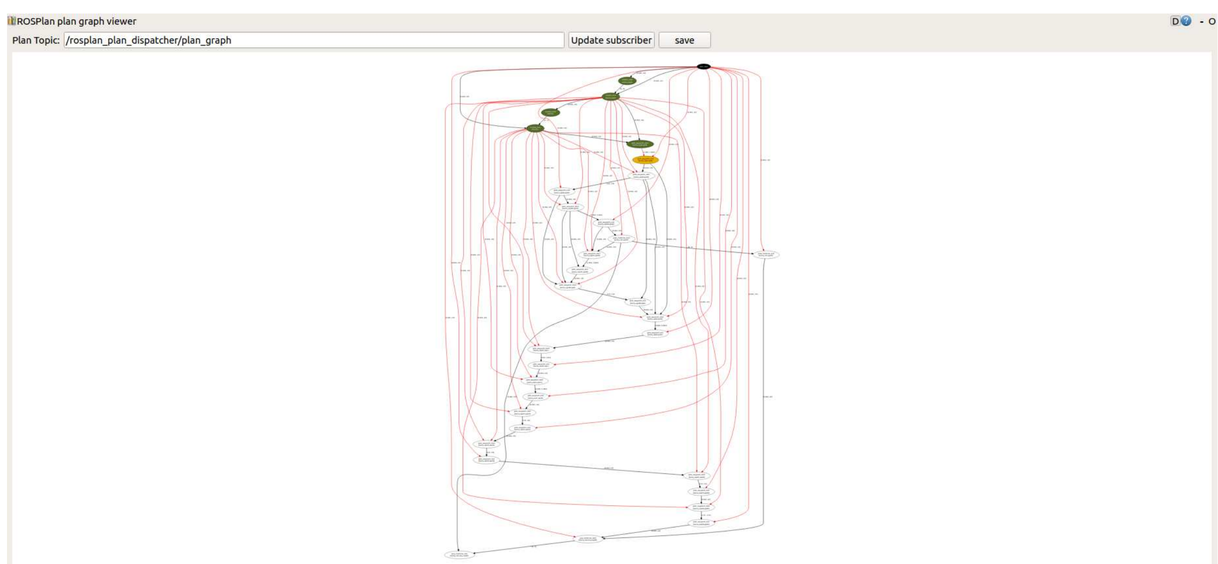

Figura 3.1: Esterel Plan Viewer

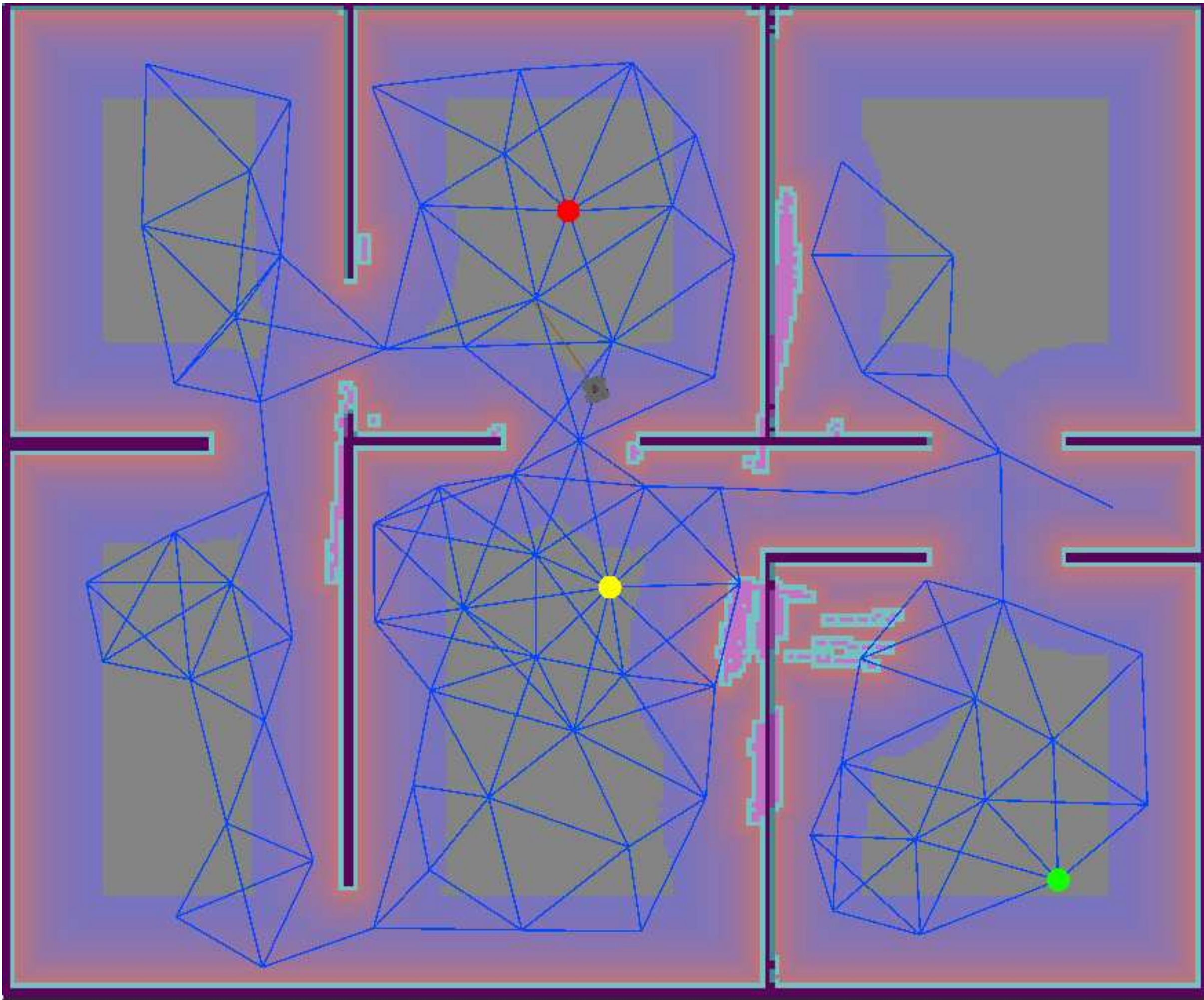

Figura 3.2: Ambiente di simulazione della demo; i waypoint evidenziati in giallo, verde e rosso rappresentano, rispettivamente, le posizioni della basetta di ricarica, del paziente e della medicina

3.4 Introduzione di Oggetti Variabili

Per rendere più realistica la simulazione introduciamo degli elementi variabili nel problema, nello specifico la possibilità che il paziente si sposti dalla sua posizione attuale e che possa non eseguire le istruzioni dettate dal robot.
Per farlo utilizziamo la Sensing Interface, simulando dei potenziali sensori; prima però avremo bisogno di alcuni parametri e di un nodo publisher.

3.4.1 Settaggio Parametri

Come già detto, i parametri sono uno strumento utile per condividere facilmente informazioni tra i vari nodi del programma; ad esempio, uno dei parametri inizializzati dal

nodo coordinatore è la posizione del paziente, in modo che sia accessibile senza il bisogno di generare una chiamata al servizio adibito all'interrogazione della KB, ovvero /rosplan_knowledge_base/state/propositions.

3.4.2 Patient Position Publisher

Per permettere al paziente di spostarsi ho aggiunto un nodo che pubblica la sua nuova posizione su un nuovo topic ad intervalli semi-randomici; il codice di seguito.

```
1  rospy.init_node('patient_position_node')
2  patient_pos_pub = rospy.Publisher('patient_position_topic', String,
       queue_size = 2)
3
4  rospy.loginfo(" (%s) : Publisher node for patient position initialized!" %
       rospy.get_name())
5
6  waypoints = 0
7  while not rospy.has_param("/rosplan_demo_waypoints/wp"):
8       rospy.sleep(0.2)
9
10 while not rospy.has_param("/rosplan_demo_waypoints/edge"):
11     rospy.sleep(0.2)
12
13 while not rospy.has_param('patient_pos'):
14     rospy.sleep(0.2)
15
16 waypoints = len(rospy.get_param("/rosplan_demo_waypoints/wp"))
17
18 while not rospy.get_param('goals_achieved') and not rospy.is_shutdown():
19     time_to_wait = random.randint(30,120)
20     rospy.rostime.wallsleep(time_to_wait)
21     curr_wp = rospy.get_param('patient_pos')
22     new_wp = -1
23     while not rospy.has_param('/rosplan_demo_waypoints/edge/wp' + str(
       curr_wp) + '/wp' + str(new_wp)):
24             new_wp = random.randint(0,waypoints)
25     patient_pos_pub.publish(String('wp' + str(new_wp)))
26     rospy.rostime.wallsleep(1.0)
```

Listing 3.11: Patient position publisher.

Il nodo aspetta che siano disponibili i parametri necessari (righe 7-14) ed estrapola il numero di waypoint (riga 16) e la posizione del paziente (riga 21).

Successivamente sceglie un waypoint casuale e se è connesso a quello attualmente occupato dal paziente (righe 23-24) lo pubblica nel topic patient_position_topic (riga 26).

3.4.3 Sensing Interface Topics

Per aggiornare la KB quando il nostro sensore simulato (patient position publisher) pubblica la nuova posizione dobbiamo modificare la Sensing Interface.
Per prima cosa è necessario compilare il file sensing_config.yaml. Sotto la voce topics aggiungiamo:

```
patient_at:
    params:
        - larry
        - '*'
    topic: /patient_position_topic
    msg_type: std_msgs/String
```

Listing 3.12: Sensing config.

Dobbiamo specificare il nome del predicato che verrà aggiornato, cioè patient_at, e specificarne i parametri.
Il secondo parametro “*” è una wildcard e quindi rappresenta qualsiasi valore generico, anche se deve comunque essere un oggetto presente nel file instance e di uno dei tipi definiti nel dominio.
Successivamente bisogna specificare il topic da ascoltare e il tipo di messaggi che arriveranno.
Per conoscere la tipologia di messaggio è sufficiente usare da terminale il comando rosmsg show.
Se il messaggio in arrivo è già un valore accettabile per il predicato questa modifica alla Sensing Interface è sufficiente; tuttavia nel nostro caso abbiamo bisogno di una funzione apposita che andrà specificata all’interno di un file indicato sotto la voce functions in sensing_config.yaml.
La funzione in questione deve condividere il nome con il predicato, nel nostro caso patient_at, e deve avere come parametri il messaggio che si riceve dal topic e i parametri del predicato.
La funzione verrà eseguita ogni volta che il patient position publisher pubblicherà un nuovo messaggio; di seguito il codice:

```
rospy.loginfo(" (%s) : Patient has moved!" % rospy.get_name())

if rospy.get_param('dispatch_called'):
    cd = rospy.ServiceProxy('/rosplan_plan_dispatcher/cancel_dispatch',
   Empty)
```

```
    cd()
    rospy.set_param('dispatch_called', False)
    move_base_pub = rospy.Publisher('move_base_reset', Bool, queue_size =
   2)
    msg_navigation = Bool()
    msg_navigation.data = True
    move_base_pub.publish(msg_navigation)

ret_value = []

# Find current patient_location
attributes = get_kb_attribute("patient_at")
curr_wp = ''
for a in attributes:
    if not a.is_negative:
        curr_wp = a.values[1].value
        break

if curr_wp != '':
    kus_remove = KnowledgeUpdateServiceRequest()
    kus_remove.update_type = 0
    kus_remove.knowledge.knowledge_type = 1
    kus_remove.knowledge.is_negative = 1
    kus_remove.knowledge.attribute_name = 'patient_at'
    kv_remove_patient = KeyValue()
    kv_remove_patient.key = 'p'
    kv_remove_patient.value = 'larry'
    kus_remove.knowledge.values.append(kv_remove_patient)
    kv_remove_waypoint = KeyValue()
    kv_remove_waypoint.key = 'wp'
    kv_remove_waypoint.value = curr_wp
    kus_remove.knowledge.values.append(kv_remove_waypoint)
    kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
   KnowledgeUpdateService)
    if not kuc(kus_remove):
        rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
   ())

    kus_add = KnowledgeUpdateServiceRequest()
    kus_add.update_type = 0
    kus_add.knowledge.knowledge_type = 1
    kus_add.knowledge.attribute_name = 'patient_at'
    kv_add_patient = KeyValue()
    kv_add_patient.key = 'p'
```

```
    kv_add_patient.value = 'larry'
    kus_add.knowledge.values.append(kv_add_patient)
    kv_add_waypoint = KeyValue()
    kv_add_waypoint.key = 'wp'
    kv_add_waypoint.value = msg.data
    kus_add.knowledge.values.append(kv_add_waypoint)
    kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
KnowledgeUpdateService)
    if not kuc(kus_add):
        rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
())

    rospy.set_param('patient_pos',str(msg.data))
    rospy.set_param('patient_has_moved', True)
return ret_value
```

Listing 3.13: Sensing interface function.

Se il Plan Dispatch è attivo il codice lo blocca insieme all'esecuzione delle azioni già in esecuzione e aggiorna il parametro dispatch_called.
In seguito estrapola la posizione attuale del paziente; questa informazione ci serve per settare a False il predicato patient_at larry <posizione attuale>.
Normalmente per settare a True il predicato con la nuova posizione basta compilare correttamente la variabile ret_value ma in questo caso, come per l'operazione precedente, aggiorneremo manualmente la KB attraverso il servizio /rosplan_knowledge_base/update, come abbiamo già visto nella Sezione 3.3.1; questo perché altrimenti i parametri patient_pos e patient_has_moved verrebbero modificate prima dell'effettivo update della KB.
Dobbiamo comunque ritornare ret_value alla fine della funzione perché risulti valida, dato che dovrebbe contenere l'aggiornamento del predicato.

3.4.4 Sensing Interface Services

Per quanto riguarda l'esito delle azioni give_medicine, tell_to_take e tell_to_call utilizziamo sempre la Sensing Interface ma, invece di sfruttare un publisher e un topic, scegliamo i servizi.
Dobbiamo ancora una volta modificare il file sensing_config.yaml, questa volta sotto la voce services.
Analizziamo il caso del predicato has_medicine (che indica se il paziente abbia ricevuto la medicina).

```
has_medicine:
    params:
```

```
        - larry
        - oki
    service: /rosplan_knowledge_base/state/propositions
    srv_type: rosplan_knowledge_msgs/GetAttributeService
    time_between_calls: 10
    operation: 'not res.attributes[0].is_negative if res != None else False
   '
```

Listing 3.14: Sensing config.

Possiamo notare che, come nel caso dei topic, è necessario specificare il predicato e i parametri relativi ad esso.
Le voci successive sono differenti; bisogna indicare il servizio che si intende interrogare e la sua tipologia.
Similmente alla variante dei topic è possibile scoprire il tipo di servizio tramite il comando rossrv show.
La prossima informazione da inserire è il tempo che deve intercorrere tra una chiamata e l'altra; questo valore viene espresso in secondi.
L'ultima voce è l'operazione (in linguaggio python) da eseguire con il risultato della chiamata di servizio, al quale potremo riferirci con il nome res.
Se la richiesta che vogliamo fare è di tipo standard dobbiamo aggiungere la voce request tra time_between_calls e operation e indicarne il tipo.
Nel nostro caso vogliamo effettuare una richiesta personalizzata, quindi la aggiungeremo al file precisato nella sezione functions. La funzione dovrà essere chiamata req_<nome del predicato> e non avrà parametri.
Per continuare il nostro esempio analizziamo req_has_medicine (per gli altri predicati la struttura è analoga):

```
try:
    result = get_kb_attribute("has_medicine")
except:
    return None
if len(result) > 0:
    if result[0].is_negative:
        number = random.randint(0,10)
        if number < 6:
            return GetAttributeServiceRequest('has_tried_to_give')
        else:
            return GetAttributeServiceRequest('has_medicine')
    else:
        return GetAttributeServiceRequest('has_medicine')
```

```
14 else:
15     return None
```

Listing 3.15: Sensing interface function.

Se il fatto non esiste ancora, la funzione ritorna il valore None che, come possiamo vedere nella voce operation dello snippet precedente, risulterà nel valore False del predicato.
Qualora il fatto esista già, mantiene il suo valore se è positivo altrimenti, con una percentuale del 70%, il predicato diventa vero quando il robot prova a dare la medicina al paziente.
Nei casi restanti il predicato mantiente il suo stato invariato.
Ho scelto una percentuale di successo del 70% perché sono azioni che il paziente deve svolgere quotidianamente, e quindi con il quale ha familiarità.

3.5 Gestione dei Fallimenti

Quanto fatto finora ci permette di avere una certa imprevedibilità riguardo la buona riuscita di alcune azioni, il che rende necessario gestire le situazioni avverse.
Per fare ciò amplieremo il nodo coordinatore.
Invece di imporre subito il goal finale, spezziamo la pianicazione in problemi più piccoli, specificando dei goals intermedi da raggiungere.
Raggiunto il goal il coordinatore aspetta l'aggiornamento da parte della Sensing Interface e, in caso di buona riuscita, passa al problema successivo, altrimenti resetta a False i predicati che indicano il tentativo del robot e ripianifica.
Questo algoritmo funziona per le azioni che coinvolgono i predicati has_medicine, has_taken e has_called.
Per gestire correttamente anche la situazione in cui il paziente si sposta controlliamo l'esito della funzione execute; se il Dispatcher non è terminato raggiungendo il goal significa che è stato interrotto a causa della Sensing Interface e quindi risulta necessaria la ripianificazione, essendo cambiato un dato del problema.
Di seguito il codice che gestisce la pianificazione e l'esecuzione del piano per riuscire a dare la medicina al paziente:

```
1 gas = rospy.ServiceProxy('/rosplan_knowledge_base/state/propositions',
      GetAttributeService)
2 res = True
3 while res:
4     plan_found = False
5
6     while not plan_found
```

```
7         plan_found = generate_problem_and_plan()
8         rospy.sleep(1)
9 
10     execute_plan()
11 
12     rospy.sleep(10)
13 
14     res = gas.call(GetAttributeServiceRequest('has_medicine')).attributes
    [0].is_negative
15     if res:
16         rospy.set_param('actual_goal', 1)
17         rospy.set_param('goal1_achieved', False)
18 
19         kus_httg1 = KnowledgeUpdateServiceRequest()
20         kus_httg1.update_type = 0
21         kus_httg1.knowledge.knowledge_type = 1
22         kus_httg1.knowledge.is_negative = 1
23         kus_httg1.knowledge.attribute_name = 'has_tried_to_give'
24         kv_httg4 = KeyValue()
25         kv_httg4.key = 'v'
26         kv_httg4.value = 'kenny'
27         kus_httg1.knowledge.values.append(kv_httg4)
28         kv_httg5 = KeyValue()
29         kv_httg5.key = 'p'
30         kv_httg5.value = 'larry'
31         kus_httg1.knowledge.values.append(kv_httg5)
32         kv_httg6 = KeyValue()
33         kv_httg6.key = 'm'
34         kv_httg6.value = 'oki'
35         kus_httg1.knowledge.values.append(kv_httg6)
36         kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
37         if not kuc(kus_httg1):
38             rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.
    get_name())
```

Listing 3.16: Coordinatore.

Il codice richiama la funzione di generazione del piano finché non ne trova uno e successivamente chiama la funzione di esecuzione. Attende 10 secondi prima di controllare lo stato del predicato has_medicine in modo tale che la Sensing Interface abbia il tempo di aggiornarne il valore.

Se il robot non è riuscito a consegnare la medicina al paziente viene resettato il valore del predicato has_tried_to_give a False; in questo modo durante la prossima esecuzione il robot ritenterà

l’azione give_medicine.
Il codice ripete questo processo finché il robot non riesce a consegnare la medicina al paziente.

Capitolo 4

Analisi dei risultati ottenuti

I test sono stati fatti nello scenario descritto nella Figura 3.2, con la posizione della medicina e la posizione iniziale del paziente fisse.
Ho eseguito 40 test prendendo i tempi della fase di setup (ovvero tutto ciò che avviene prima della chiamata al dispatcher), i tempi di esecuzione e il numero di fallimenti dati dallo spostamento del paziente e dalla non riuscita delle azioni give_medicine, tell_to_take_medicine e tell_to_call.
I risultati ottenuti (Tabella 4.1) mostrano che i tempi di setup rimangono molto simili tra di loro, questo perché la taglia del problema non cambia.
Le tempistiche dell'esecuzione invece, sono molto variabili, dato che in questo lasso di tempo possono verificarsi varie ripianificazioni causate dallo spostamento del paziente o dal fallimento di un'azione. È infatti possibile notare che i tempi di esecuzioni crescono insieme al numero di ripianificazioni.
Nonostante questa caratteristica è facile accorgersi che i tempi di esecuzioni di prove differenti non necessariamente coincidono a parità di ripianificazioni. Questo accade per due ragioni: la prima è che il sistema di navigazione del robot tende a sovrastimare la vicinanza delle pareti, e quindi ha difficoltà nel trovare un percorso per raggiungere il waypoint dell'obiettivo. La seconda è dovuta al fatto che le chiamate della Sensing Interface ai servizi per aggiornare i predicati potrebbero venire eseguite dopo che la ripianificazione abbia avuto inizio. In questo modo il planner non riesce a trovare un piano coerentemente con gli aggiornamenti ed è necessario ripianificare ulteriormente.
Infine, notiamo che le ripianificazioni dovute al fallimento di un'azione (# Fallimenti) incidono maggiormente sulle tempistiche rispetto a quelle dovute allo spostamento del paziente (# Spostamenti). Questo perché per sapere se l'azione è andata a buon fine è necessario aspettare l'aggiornamento dei predicati da parte della Sensing Interface, cosa che richiede molto più tempo di uno spostamento (come si può vedere confrontando ad esempio le run 36 e 37, dove il

numero totale di ripianificazioni è invariato ma un numero maggiore di fallimenti delle azioni, bilanciato con un numero inferiore di spostamenti, porta comunque ad un'aumento del tempo di circa 30 secondi) ; infatti, una volta raggiunto il paziente, sarà molto più facile per il robot seguirlo, dato che si sposterà in un waypoint vicino.

Tabella 4.1: Risultati dei test (in secondi)

# Esecuzione	Setup	Esecuzione	Totale	# Spostamenti	# Fallimenti	# Ripianificazioni
1	18,4	223,7	242,1	3	3	6
2	18,1	394,9	413	7	7	14
3	18,8	293,6	312,4	5	4	9
4	18,5	258,1	276,6	3	3	6
5	18,5	232,8	251,3	4	2	6
6	18,9	231,8	250,7	3	3	6
7	19,5	375,6	395,1	6	5	11
8	19	225,5	244,5	3	3	6
9	19,2	295,2	314,4	5	5	10
10	19	210	229	3	2	5
11	19,6	286,2	305,8	3	4	7
12	19,6	236,3	255,9	5	4	9
13	19,9	222,5	242,4	2	2	4
14	18,8	202	220,8	2	2	4
15	19,1	312,9	332	3	5	8
16	19,4	322,9	342,3	4	5	9
17	19,6	271,7	291,3	3	5	8
18	18,6	405,8	424,4	4	4	8
19	19,4	221,4	240,8	3	2	5
20	18,7	346,1	364,8	6	5	11
21	18,8	334,8	353,6	4	4	8
22	19,1	209,5	228,6	3	2	5
23	19,2	253,7	272,9	2	3	5
24	18,9	374,1	393	6	7	13
25	19,1	188,6	207,7	2	1	3
26	18,9	252	270,9	5	4	9
27	18,2	248,3	266,5	3	3	6
28	18,6	259,8	278,4	3	3	6
29	18,6	338,7	357,3	5	5	10
30	18,7	265,5	284,2	3	3	6
31	19,4	214,4	233,8	2	2	4
32	19,1	300	319,1	5	6	11
33	18,7	186,5	205,2	3	2	5
34	18,8	221,7	240,5	2	2	4
35	19,2	349,6	368,8	4	6	10
36	18,5	305,8	324,3	6	5	11
37	19,3	339	358,3	5	6	11
38	18,6	227,3	245,9	3	2	5
39	19,6	251,8	271,4	5	4	9
40	19,6	414,8	434,4	6	8	14
Media	18,99	277,62	296,61	3,85	3,83	7,68

Capitolo 5

Conclusioni e Spunti per Lavori Futuri

5.1 Conclusioni

La struttura modulare di rosplan è sicuramente un suo grande punto di forza, in quanto mi ha permesso di poter testare il progetto in corso d'opera, senza che fosse completamente ultimato. Grazie alle demo disponibili le parti relative allo stage e al movimento erano già implementate, anche se quest'ultima presenta dei problemi; in particolare il robot sovrastima la vicinanza delle pareti, cerca di raggiungere i waypoint con un'angolazione precisa e fa un giro su se stesso ogni volta che ci riesce.
Il linguaggio PDDL ha dimostrato il suo valore sia per la semplicità della sua sintassi, che mi ha permesso di produrre e modificare il file di dominio e del problema senza difficoltà, sia per la buona documentazione presente sul web.
La parte più difficile da implementare è stata quella relativa all'imprevedibilità dell'esito delle azioni e relativa gestione del fallimento, in quanto ha richiesto di sincronizzare la Sensing Interface con il nodo coordinatore e il Plan Dispatch.
Il problema è stato risolto tramite l'utilizzo di alcuni parametri che ho potuto utilizzare come semafori.
Le limitazioni più grandi di ROSPlan che ho potuto riscontrare sono state le scarse indicazioni in caso di errori, specialmente nello specificare a quale modulo appartenesse il problema, e i tempi relativamente lunghi di pianificazione, cosa che rende il framework inadatto a tasks che richiedono una certa reattività.

5.2 Lavori Futuri

Per continuare lo sviluppo di questo progetto sarebbe necessario migliorare il sistema di navigazione, risolvendo i casi in cui il robot si incastra o trova degli ostacoli; in questo modo sarebbe

possibile permettere al robot di muoversi anche in presenza di altre persone e in un ambiente più realistico.
Inoltre uno sviluppo futuro potrebbe prevedere di testare il sistema in un ambiente reale con un robot mobile fisico.
Seguendo la struttura del dominio già sviluppato si potrebbe ampliare il set di azioni disponibili e la quantità di tasks attuabili dal robot; ad esempio il robot potrebbe essere in grado di dare supporto motorio al paziente e di mantenere in memoria la posizione di vari oggetti utili, oltre alle medicine.

Capitolo 6

Appendici

6.1 Nodo Coordinatore

```
#!/usr/bin/env python
import rospkg
import rospy
import sys
import time
import os
import time
import random

from std_msgs.msg import String
from std_srvs.srv import Empty, EmptyResponse
from rosplan_knowledge_msgs.srv import *
from rosplan_interface_mapping.srv import CreatePRM
from rosplan_dispatch_msgs.srv import DispatchService,
    DispatchServiceResponse, PlanningService, PlanningServiceResponse
from diagnostic_msgs.msg import KeyValue

############
# THE REST #
############

# get path of pkg
rospy.init_node("coordinator")

# load parameters
max_prm_size = rospy.get_param('~max_prm_size', 1000)
planner_command = rospy.get_param('~planner_command', "")
domain_path = rospy.get_param('~domain_path', "")
```

```
28 problem_path = rospy.get_param('~problem_path', "")
29 data_path = rospy.get_param('~data_path', "")
30
31 # wait for services
32 rospy.wait_for_service('/rosplan_roadmap_server/create_prm')
33 rospy.wait_for_service('/rosplan_problem_interface/
    problem_generation_server')
34 rospy.wait_for_service('/rosplan_planner_interface/planning_server_params')
35 rospy.wait_for_service('/rosplan_parsing_interface/parse_plan')
36 rospy.wait_for_service('/rosplan_plan_dispatcher/dispatch_plan')
37
38 def make_prm(size):
39     # generate dense PRM
40     rospy.loginfo("KCL: (%s) Creating PRM of size %i" % (rospy.get_name(),
    size))
41     prm = rospy.ServiceProxy('/rosplan_roadmap_server/create_prm',
    CreatePRM)
42     if not prm(size,0.8,1.6,1.8,60,200000):
43         rospy.logerr("KCL: (%s) No PRM was made" % rospy.get_name())
44
45 def generate_problem_and_plan():
46
47     rospy.loginfo("KCL: (%s) Calling problem generation" % rospy.get_name()
    )
48     pg = rospy.ServiceProxy('/rosplan_problem_interface/
    problem_generation_server', Empty)
49     if not pg():
50         rospy.logerr("KCL: (%s) No problem was generated!" % rospy.get_name
    ())
51
52     rospy.loginfo("KCL: (%s) Calling planner" % rospy.get_name())
53     pi = rospy.ServiceProxy('/rosplan_planner_interface/
    planning_server_params', PlanningService)
54     pi_response = pi(domain_path, problem_path, data_path, planner_command,
     True)
55
56     if not pi_response:
57         rospy.logerr("KCL: (%s) No response from the planning server." %
    rospy.get_name())
58         return False
59     if not pi_response.plan_found:
60         rospy.loginfo("KCL: (%s) No plan could be found." % rospy.get_name
    ())
61         return False
```

```
62     else:
63         rospy.loginfo("KCL: (%s) Plan was found." % rospy.get_name())
64         return True
65 
66 def execute_plan():
67 
68     rospy.loginfo("KCL: (%s) Calling plan parser" % rospy.get_name())
69     pp = rospy.ServiceProxy('/rosplan_parsing_interface/parse_plan', Empty)
70     if not pp():
71         rospy.logerr("KCL: (%s) The plan was not parsed!" % rospy.get_name
    ())
72         return
73 
74     rospy.sleep(3)
75 
76     rospy.loginfo("KCL: (%s) Calling plan execution" % rospy.get_name())
77     rospy.set_param('dispatch_called', True)
78     pd = rospy.ServiceProxy('/rosplan_plan_dispatcher/dispatch_plan',
    DispatchService)
79     pd_response = pd()
80     rospy.set_param('dispatch_called', False)
81     if not pd_response:
82         rospy.logerr("KCL: (%s) No response from the dispatch server." %
    rospy.get_name())
83         rospy.set_param('goal_achieved', False)
84         return False
85     if not pd_response.goal_achieved:
86         rospy.loginfo("KCL: (%s) The execution was not successful." % rospy
    .get_name())
87         rospy.set_param('goal_achieved', False)
88         return False
89     else:
90         rospy.loginfo("KCL: (%s) Plan was executed." % rospy.get_name())
91         rospy.set_param('goal' + str(rospy.get_param('actual_goal')) + '
    _achieved', True)
92         rospy.set_param('actual_goal', rospy.get_param('actual_goal') + 1)
93         return True
94 
95 try:
96     rospy.sleep(3)
97     make_prm(max_prm_size)
98 
99     # wait for odom to publish to sensing interface
```

```
100     sps = rospy.ServiceProxy('/rosplan_knowledge_base/state/propositions',
    GetAttributeService)
101     count = 0
102     while count<1:
103         rospy.sleep(1)
104         gas = GetAttributeServiceRequest()
105         gas.predicate_name = 'robot_at'
106         facts = sps(gas)
107         if not facts:
108             rospy.logwarn("KCL: (%s) Proposition service not available." %
    rospy.get_name())
109         count = 0
110         for k in facts.attributes:
111             if not k.is_negative:
112                 count = count + 1
113 
114     # add knowledge
115     kus_patient = KnowledgeUpdateServiceRequest()
116     kus_patient.update_type = 0
117     kus_patient.knowledge.knowledge_type = 1
118     kus_patient.knowledge.attribute_name = 'patient_at'
119     kv_patient1 = KeyValue()
120     kv_patient1.key = 'p'
121     kv_patient1.value = 'larry'
122     kus_patient.knowledge.values.append(kv_patient1)
123     kv_patient2 = KeyValue()
124     kv_patient2.key = 'wp'
125     kv_patient2.value = 'wp65'
126     kus_patient.knowledge.values.append(kv_patient2)
127     kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
128     if not kuc(kus_patient):
129         rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
    ())
130 
131 
132     kus_medicine = KnowledgeUpdateServiceRequest()
133     kus_medicine.update_type = 0
134     kus_medicine.knowledge.knowledge_type = 1
135     kus_medicine.knowledge.attribute_name = 'medicine_at'
136     kv_medicine1 = KeyValue()
137     kv_medicine1.key = 'm'
138     kv_medicine1.value = 'oki'
139     kus_medicine.knowledge.values.append(kv_medicine1)
```

```
140     kv_medicine2 = KeyValue()
141     kv_medicine2.key = 'wp'
142     kv_medicine2.value = 'wp20'
143     kus_medicine.knowledge.values.append(kv_medicine2)
144     kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
145     if not kuc(kus_medicine):
146         rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
    ())
147 
148     kus_httg = KnowledgeUpdateServiceRequest()
149     kus_httg.update_type = 0
150     kus_httg.knowledge.knowledge_type = 1
151     kus_httg.knowledge.is_negative = 1
152     kus_httg.knowledge.attribute_name = 'has_tried_to_give'
153     kv_httg1 = KeyValue()
154     kv_httg1.key = 'v'
155     kv_httg1.value = 'kenny'
156     kus_httg.knowledge.values.append(kv_httg1)
157     kv_httg2 = KeyValue()
158     kv_httg2.key = 'p'
159     kv_httg2.value = 'larry'
160     kus_httg.knowledge.values.append(kv_httg2)
161     kv_httg3 = KeyValue()
162     kv_httg3.key = 'm'
163     kv_httg3.value = 'oki'
164     kus_httg.knowledge.values.append(kv_httg3)
165     kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
166     if not kuc(kus_httg):
167         rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
    ())
168 
169     kus_hm = KnowledgeUpdateServiceRequest()
170     kus_hm.update_type = 0
171     kus_hm.knowledge.knowledge_type = 1
172     kus_hm.knowledge.is_negative = 1
173     kus_hm.knowledge.attribute_name = 'has_medicine'
174     kv_hm1 = KeyValue()
175     kv_hm1.key = 'p'
176     kv_hm1.value = 'larry'
177     kus_hm.knowledge.values.append(kv_hm1)
178     kv_hm2 = KeyValue()
179     kv_hm2.key = 'm'
```

```
180     kv_hm2.value = 'oki'
181     kus_hm.knowledge.values.append(kv_hm2)
182     kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
        KnowledgeUpdateService)
183     if not kuc(kus_hm):
184         rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
        ())
185
186     kus_httt = KnowledgeUpdateServiceRequest()
187     kus_httt.update_type = 0
188     kus_httt.knowledge.knowledge_type = 1
189     kus_httt.knowledge.is_negative = 1
190     kus_httt.knowledge.attribute_name = 'has_told_to_take'
191     kv_httt1 = KeyValue()
192     kv_httt1.key = 'v'
193     kv_httt1.value = 'kenny'
194     kus_httt.knowledge.values.append(kv_httt1)
195     kv_httt2 = KeyValue()
196     kv_httt2.key = 'p'
197     kv_httt2.value = 'larry'
198     kus_httt.knowledge.values.append(kv_httt2)
199     kv_httt3 = KeyValue()
200     kv_httt3.key = 'm'
201     kv_httt3.value = 'oki'
202     kus_httt.knowledge.values.append(kv_httt3)
203     kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
        KnowledgeUpdateService)
204     if not kuc(kus_httt):
205         rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
        ())
206
207     kus_ht = KnowledgeUpdateServiceRequest()
208     kus_ht.update_type = 0
209     kus_ht.knowledge.knowledge_type = 1
210     kus_ht.knowledge.is_negative = 1
211     kus_ht.knowledge.attribute_name = 'has_taken'
212     kv_ht1 = KeyValue()
213     kv_ht1.key = 'p'
214     kv_ht1.value = 'larry'
215     kus_ht.knowledge.values.append(kv_ht1)
216     kv_ht2 = KeyValue()
217     kv_ht2.key = 'm'
218     kv_ht2.value = 'oki'
219     kus_ht.knowledge.values.append(kv_ht2)
```

```
220     kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
221     if not kuc(kus_ht):
222         rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.get_name
    ())
223 
224     #setting params
225     rospy.set_param('goals_achieved', False)
226     rospy.set_param('actual_goal', 1)
227     rospy.set_param('goal1_achieved', False)
228     rospy.set_param('goal2_achieved', False)
229     rospy.set_param('patient_has_moved', False)
230     rospy.set_param('patient_pos', 'wp65')
231     rospy.set_param('dispatch_called', False)
232 
233     # add goals to the KB
234     kus_goal1 = KnowledgeUpdateServiceRequest()
235     kus_goal1.update_type = 1
236     kus_goal1.knowledge.knowledge_type = 1
237     kus_goal1.knowledge.attribute_name = 'has_tried_to_give'
238     kv_goal11 = KeyValue()
239     kv_goal11.key = 'v'
240     kv_goal11.value = 'kenny'
241     kus_goal1.knowledge.values.append(kv_goal11)
242     kv_goal12 = KeyValue()
243     kv_goal12.key = 'p'
244     kv_goal12.value = 'larry'
245     kus_goal1.knowledge.values.append(kv_goal12)
246     kv_goal13 = KeyValue()
247     kv_goal13.key = 'm'
248     kv_goal13.value = 'oki'
249     kus_goal1.knowledge.values.append(kv_goal13)
250     kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
251     if not kuc(kus_goal1):
252         rospy.logerr("KCL: (%s) Goal was not added!" % rospy.get_name())
253 
254     gas = rospy.ServiceProxy('/rosplan_knowledge_base/state/propositions',
    GetAttributeService)
255     res = True
256     while res:
257         plan_found = False
258 
259         while not plan_found:
```

```
260             plan_found = generate_problem_and_plan()
261             rospy.sleep(1)
262 
263         execute_plan()
264 
265         rospy.sleep(10)
266 
267         res = gas.call(GetAttributeServiceRequest('has_medicine')).
    attributes[0].is_negative
268         if res:
269             rospy.set_param('actual_goal', 1)
270             rospy.set_param('goal1_achieved', False)
271 
272             kus_httg1 = KnowledgeUpdateServiceRequest()
273             kus_httg1.update_type = 0
274             kus_httg1.knowledge.knowledge_type = 1
275             kus_httg1.knowledge.is_negative = 1
276             kus_httg1.knowledge.attribute_name = 'has_tried_to_give'
277             kv_httg4 = KeyValue()
278             kv_httg4.key = 'v'
279             kv_httg4.value = 'kenny'
280             kus_httg1.knowledge.values.append(kv_httg4)
281             kv_httg5 = KeyValue()
282             kv_httg5.key = 'p'
283             kv_httg5.value = 'larry'
284             kus_httg1.knowledge.values.append(kv_httg5)
285             kv_httg6 = KeyValue()
286             kv_httg6.key = 'm'
287             kv_httg6.value = 'oki'
288             kus_httg1.knowledge.values.append(kv_httg6)
289             kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
290             if not kuc(kus_httg1):
291                 rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.
    get_name())
292 
293     print("Il robot e' riuscito a consegnare la medicina")
294 
295     # add goals to the KB
296     kus_goal2 = KnowledgeUpdateServiceRequest()
297     kus_goal2.update_type = 1
298     kus_goal2.knowledge.knowledge_type = 1
299     kus_goal2.knowledge.attribute_name = 'has_told_to_take'
300     kv_goal21 = KeyValue()
```

```
301        kv_goal21.key = 'v'
302        kv_goal21.value = 'kenny'
303        kus_goal2.knowledge.values.append(kv_goal21)
304        kv_goal22 = KeyValue()
305        kv_goal22.key = 'p'
306        kv_goal22.value = 'larry'
307        kus_goal2.knowledge.values.append(kv_goal22)
308        kv_goal23 = KeyValue()
309        kv_goal23.key = 'm'
310        kv_goal23.value = 'oki'
311        kus_goal2.knowledge.values.append(kv_goal23)
312        kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
       KnowledgeUpdateService)
313        if not kuc(kus_goal2):
314            rospy.logerr("KCL: (%s) Goal was not added!" % rospy.get_name())
315
316        res = True
317        while res:
318            plan_found = False
319            while not plan_found:
320                plan_found = generate_problem_and_plan()
321                rospy.sleep(1)
322
323            execute_plan()
324
325            rospy.sleep(10)
326
327            res = gas.call(GetAttributeServiceRequest('has_taken')).attributes
       [0].is_negative
328            if res:
329                rospy.set_param('actual_goal', 2)
330                rospy.set_param('goal2_achieved', False)
331
332                kus_httt1 = KnowledgeUpdateServiceRequest()
333                kus_httt1.update_type = 0
334                kus_httt1.knowledge.knowledge_type = 1
335                kus_httt1.knowledge.is_negative = 1
336                kus_httt1.knowledge.attribute_name = 'has_told_to_take'
337                kv_httt4 = KeyValue()
338                kv_httt4.key = 'v'
339                kv_httt4.value = 'kenny'
340                kus_httt1.knowledge.values.append(kv_httt4)
341                kv_httt5 = KeyValue()
342                kv_httt5.key = 'p'
```

```
343             kv_htttt5.value = 'larry'
344             kus_htttt1.knowledge.values.append(kv_htttt5)
345             kv_htttt6 = KeyValue()
346             kv_htttt6.key = 'm'
347             kv_htttt6.value = 'oki'
348             kus_htttt1.knowledge.values.append(kv_htttt6)
349             kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
350             if not kuc(kus_htttt1):
351                 rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.
    get_name())
352
353             kus_htt = KnowledgeUpdateServiceRequest()
354             kus_htt.update_type = 0
355             kus_htt.knowledge.knowledge_type = 1
356             kus_htt.knowledge.attribute_name = 'has_to_take'
357             kv_htt1 = KeyValue()
358             kv_htt1.key = 'p'
359             kv_htt1.value = 'larry'
360             kus_htt.knowledge.values.append(kv_htt1)
361             kv_htt2 = KeyValue()
362             kv_htt2.key = 'm'
363             kv_htt2.value = 'oki'
364             kus_htt.knowledge.values.append(kv_htt2)
365             kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
366             if not kuc(kus_htt):
367                 rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.
    get_name())
368
369     print("Il paziente ha preso la medicina")
370
371     # add goals to the KB
372     kus_goal3 = KnowledgeUpdateServiceRequest()
373     kus_goal3.update_type = 1
374     kus_goal3.knowledge.knowledge_type = 1
375     kus_goal3.knowledge.attribute_name = 'has_told_to_call'
376     kv_goal31 = KeyValue()
377     kv_goal31.key = 'v'
378     kv_goal31.value = 'kenny'
379     kus_goal3.knowledge.values.append(kv_goal31)
380     kv_goal32 = KeyValue()
381     kv_goal32.key = 'p'
382     kv_goal32.value = 'larry'
```

```
383     kus_goal3.knowledge.values.append(kv_goal32)
384     kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
385     if not kuc(kus_goal3):
386         rospy.logerr("KCL: (%s) Goal was not added!" % rospy.get_name())
387 
388     res = True
389     while res:
390         plan_found = False
391         while not plan_found:
392             plan_found = generate_problem_and_plan()
393             rospy.sleep(1)
394 
395         execute_plan()
396 
397         rospy.sleep(10)
398 
399         res = gas.call(GetAttributeServiceRequest('has_called')).attributes
    [0].is_negative
400         if res:
401             rospy.set_param('actual_goal', 3)
402             rospy.set_param('goal3_achieved', False)
403 
404             kus_httc = KnowledgeUpdateServiceRequest()
405             kus_httc.update_type = 0
406             kus_httc.knowledge.knowledge_type = 1
407             kus_httc.knowledge.is_negative = 1
408             kus_httc.knowledge.attribute_name = 'has_told_to_call'
409             kv_httc1 = KeyValue()
410             kv_httc1.key = 'v'
411             kv_httc1.value = 'kenny'
412             kus_httc.knowledge.values.append(kv_httc1)
413             kv_httc2 = KeyValue()
414             kv_httc2.key = 'p'
415             kv_httc2.value = 'larry'
416             kus_httc.knowledge.values.append(kv_httc2)
417             kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
418             if not kuc(kus_httc):
419                 rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.
    get_name())
420 
421             kus_htc = KnowledgeUpdateServiceRequest()
422             kus_htc.update_type = 0
```

```
423             kus_htc.knowledge.knowledge_type = 1
424             kus_htc.knowledge.attribute_name = 'has_to_call'
425             kv_htc1 = KeyValue()
426             kv_htc1.key = 'p'
427             kv_htc1.value = 'larry'
428             kus_htc.knowledge.values.append(kv_htc1)
429             kv_htc2 = KeyValue()
430             kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
431             if not kuc(kus_htc):
432                 rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.
    get_name())
433 
434     print("Il paziente ha chiamato il dottore")
435 
436     rospy.set_param('goals_achieved', True)
437     print("Tutti i goal sono stati raggiunti")
438 
439 except rospy.ServiceException as e:
440     rospy.logerr("KCL: (%s) Service call failed: %s" % (rospy.get_name(), e
    ))
```

6.2 Domain

```
1 (define (domain dom) ;; speeded up version of turtlebot domain, note that
    action durations are short
2 
3 (:requirements :strips :typing :fluents :disjunctive-preconditions
    :durative-actions)
4 
5 (:types
6   waypoint
7   robot
8   patient
9   medicine
10 )
11 
12 (:predicates
13   ;positions
14   (dock_at ?wp - waypoint)
15   (robot_at ?v - robot ?wp - waypoint)
16   (patient_at ?p - patient ?wp - waypoint)
17   (medicine_at ?m - medicine ?wp - waypoint)
```

```
  ;robot flags
  (undocked ?v - robot)
  (docked ?v - robot)
  (localised ?v - robot)
  (robot_has_medicine ?v - robot ?m - medicine)
  (can_grab ?v - robot)
  (has_tried_to_give ?v - robot ?p - patient ?m - medicine)
  (has_told_to_take ?v - robot ?p - patient ?m - medicine)
  (has_told_to_call ?v - robot ?p - patient)
  ;patient flags
  (has_medicine ?p - patient ?m - medicine)
  (has_to_take ?p - patient ?m - medicine)
  (has_taken ?p - patient ?m - medicine)
  (has_to_call ?p - patient)
  (has_called ?p - patient)
)

(:functions
  (distance ?a ?b - waypoint)
  (charge ?v - robot)
)

;; Move to any waypoint, avoiding terrain
(:durative-action goto_waypoint
  :parameters (?v - robot ?from ?to - waypoint)
  :duration ( = ?duration (distance ?from ?to))
  :condition (and
    (at start (robot_at ?v ?from))
    (at start (localised ?v))
    (over all (undocked ?v))
    (over all (> (charge ?v) 0)))
  :effect (and
  ; (at end (visited ?to))
    (at end (robot_at ?v ?to))
    (at start (not (robot_at ?v ?from)))
    (at end (decrease (charge ?v) (distance ?from ?to))))
)

;; Localise
(:durative-action localise
  :parameters (?v - robot)
  :duration ( = ?duration 1)
  :condition (over all (undocked ?v))
  :effect (at end (localised ?v))
```

```
62 )
63
64 ;; Dock to the charge base
65 (:durative-action dock
66   :parameters (?v - robot ?wp - waypoint)
67   :duration ( = ?duration 3)
68   :condition (and
69     (over all (dock_at ?wp))
70     (at start (robot_at ?v ?wp))
71     (at start (undocked ?v))
72     (over all (> (charge ?v) 0)))
73   :effect (and
74     (at end (docked ?v))
75     (at start (not (undocked ?v)))
76     (at end (assign (charge ?v) 100)))
77 )
78
79 ;;undock from the charge base
80 (:durative-action undock
81   :parameters (?v - robot ?wp - waypoint)
82   :duration ( = ?duration 2)
83   :condition (and
84     (over all (dock_at ?wp))
85     (at start (docked ?v)))
86   :effect (and
87     (at start (not (docked ?v)))
88     (at end (undocked ?v)))
89 )
90
91 ;;take the medicine to carry it
92 (:durative-action grab_medicine
93   :parameters (?v - robot ?m - medicine ?wp - waypoint)
94   :duration (= ?duration 1)
95   :condition (and
96     (at start (robot_at ?v ?wp))
97     (at start (medicine_at ?m ?wp))
98     (at start (can_grab ?v))
99     (over all (undocked ?v))
100     (over all (> (charge ?v) 0))
101   )
102   :effect (and
103     (at end (robot_has_medicine ?v ?m))
104     (at start (not (medicine_at ?m ?wp)))
105     (at start (not (can_grab ?v)))
```

```
106     (at end (decrease (charge ?v) 1))
107   )
108 )
109
110 ;;give the medicine to the patient
111 (:durative-action give_medicine
112   :parameters (?v - robot ?m - medicine ?p - patient ?wp - waypoint)
113   :duration (= ?duration 1)
114   :condition (and
115     (at start (robot_at ?v ?wp))
116     (at start (robot_has_medicine ?v ?m))
117     (at start (patient_at ?p ?wp))
118     (at start (has_to_take ?p ?m))
119     (over all (undocked ?v))
120     (over all (> (charge ?v) 0))
121   )
122   :effect (and
123     (at end (can_grab ?v))
124     (at end (has_tried_to_give ?v ?p ?m))
125     (at end (decrease (charge ?v) 1))
126   )
127 )
128
129 ;;tell to the patient to take the medicine
130 (:durative-action tell_to_take_medicine
131   :parameters (?v - robot ?p - patient ?m - medicine ?wp - waypoint)
132   :duration (= ?duration 1)
133   :condition (and
134     (at start (robot_at ?v ?wp))
135     (over all (patient_at ?p ?wp))
136     (at start (has_to_take ?p ?m))
137     (at start (has_medicine ?p ?m))
138     (over all (undocked ?v))
139     (over all (> (charge ?v) 0))
140   )
141   :effect (and
142     (at end (has_told_to_take ?v ?p ?m))
143     (at end (not (has_medicine ?p ?m)))
144     (at end (decrease (charge ?v) 1))
145   )
146 )
147
148 ;;tell to the patient to call the doctor
149 (:durative-action tell_to_call
```

```
150   :parameters (?v - robot ?p - patient ?m - medicine ?wp - waypoint)
151   :duration (= ?duration 1)
152   :condition (and
153     (at start (robot_at ?v ?wp))
154     (over all (patient_at ?p ?wp))
155     (at start (has_to_call ?p))
156     (at start (has_taken ?p ?m))
157     (over all (undocked ?v))
158     (over all (> (charge ?v) 0))
159   )
160   :effect (and
161     (at start (has_told_to_call ?v ?p))
162     (at start (not (has_to_call ?p)))
163     (at end (decrease (charge ?v) 1))
164   )
165 )
166
167
168 )
```

Listing 6.1: types.

6.3 Patient Position Publisher

```
1 #!/usr/bin/env python
2 import rospy
3 import random
4 from std_msgs.msg import String
5 from rosplan_knowledge_msgs.srv import *
6
7 def patient_position():
8
9     rospy.init_node('patient_position_node')
10    patient_pos_pub = rospy.Publisher('patient_position_topic', String,
     queue_size = 2)
11
12    rospy.loginfo(" (%s) : Publisher node for patient position initialized!
     " % rospy.get_name())
13
14    waypoints = 0
15    while not rospy.has_param("/rosplan_demo_waypoints/wp"):
16        rospy.sleep(0.2)
17
```

```python
    while not rospy.has_param("/rosplan_demo_waypoints/edge"):
        rospy.sleep(0.2)

    while not rospy.has_param('patient_pos'):
        rospy.sleep(0.2)

    waypoints = len(rospy.get_param("/rosplan_demo_waypoints/wp"))

    while not rospy.has_param('goal1_achieved'):
        rospy.sleep(0.2)

    while not rospy.get_param('goals_achieved') and not rospy.is_shutdown()
    :
        time_to_wait = random.randint(30,120)
        rospy.rostime.wallsleep(time_to_wait)
        curr_wp = rospy.get_param('patient_pos')
        new_wp = -1
        while not rospy.has_param('/rosplan_demo_waypoints/edge/wp' + str(
    curr_wp) + '/wp' + str(new_wp)):
                new_wp = random.randint(0,waypoints)
        patient_pos_pub.publish(String('wp' + str(new_wp)))
        rospy.rostime.wallsleep(1.0)

patient_position()
```

6.4 Sensing Config

```yaml
functions:
    - $(find rosplan_stage_exploration_demo)/scripts/
    pddl_exploration_sensing_functions.py
topics:
    robot_at:
        params:
            - kenny
            - '*'
        topic: /amcl_pose
        msg_type: geometry_msgs/PoseWithCovarianceStamped

    patient_at:
        params:
            - larry
            - '*'
        topic: /patient_position_topic
```

```
        msg_type: std_msgs/String
services:
    has_medicine:
        params:
            - larry
            - oki
        service: /rosplan_knowledge_base/state/propositions
        srv_type: rosplan_knowledge_msgs/GetAttributeService
        time_between_calls: 10
        operation: 'not res.attributes[0].is_negative if res != None else
    False'
    has_taken:
        params:
            - larry
            - oki
        service: /rosplan_knowledge_base/state/propositions
        srv_type: rosplan_knowledge_msgs/GetAttributeService
        time_between_calls: 10
        operation: 'not res.attributes[0].is_negative if res != None else
    False'
    has_called:
        params:
            - larry
        service: /rosplan_knowledge_base/state/propositions
        srv_type: rosplan_knowledge_msgs/GetAttributeService
        time_between_calls: 10
        operation: 'not res.attributes[0].is_negative if res != None else
    False'
```

6.5 Sensing Functions

```python
#!/usr/bin/env python
import numpy as np
import random
from math import sqrt


import rospkg
import rospy
import sys
import time
import os
import time
```

```
13 import random
14
15 from std_msgs.msg import String
16 from std_srvs.srv import Empty, EmptyResponse
17 from rosplan_knowledge_msgs.srv import *
18 from std_msgs.msg import Bool
19 from rosplan_interface_mapping.srv import CreatePRM
20 from rosplan_dispatch_msgs.srv import DispatchService,
     DispatchServiceResponse, PlanningService, PlanningServiceResponse
21 from diagnostic_msgs.msg import KeyValue
22 from threading import Thread
23
24 def robot_at(msg, params):
25     assert(msg.header.frame_id == "map")
26     assert(len(params) == 2)
27
28     ret_value = []
29     attributes = get_kb_attribute("robot_at")
30     curr_wp = ''
31
32     # Find current robot_location
33     for a in attributes:
34         if not a.is_negative:
35             curr_wp = a.values[1].value
36             break
37
38     for robot in params[0]:
39         distance = float('inf')
40         closest_wp = ''
41
42         waypoints = []
43         while not rospy.has_param("/rosplan_demo_waypoints/wp"):
44             rospy.sleep(0.2)
45
46         waypoints = rospy.get_param("/rosplan_demo_waypoints/wp")
47
48         for wp in waypoints:
49             pose = rospy.get_param("/rosplan_demo_waypoints/wp/"+wp)
50             assert(len(pose) > 0)
51             x = pose[0] - msg.pose.pose.position.x
52             y = pose[1] - msg.pose.pose.position.y
53             d = sqrt(x**2 + y**2)
54             if d < distance:
55                 closest_wp = wp
```

```
56                  distance = d
57          if curr_wp != closest_wp:
58              ret_value.append((robot + ':' + closest_wp, True))  # Set new
    wp to true
59              if curr_wp != '':
60                  ret_value.append((robot + ':' + curr_wp, False)) # Set
    current waypoint to false
61
62      return ret_value
63
64  def patient_at(msg, params):
65
66      rospy.loginfo(" (%s) : Patient has moved!" % rospy.get_name())
67
68      if rospy.get_param('dispatch_called'):
69          cd = rospy.ServiceProxy('/rosplan_plan_dispatcher/cancel_dispatch',
     Empty)
70          cd()
71          rospy.set_param('dispatch_called', False)
72          move_base_pub = rospy.Publisher('move_base_reset', Bool, queue_size
     = 2)
73          msg_navigation = Bool()
74          msg_navigation.data = True
75          move_base_pub.publish(msg_navigation)
76
77      ret_value = []
78
79      attributes = get_kb_attribute("patient_at")
80      curr_wp = ''
81
82      # Find current patient_location
83      for a in attributes:
84          if not a.is_negative:
85              curr_wp = a.values[1].value
86              break
87
88      if curr_wp != '':
89          kus_remove = KnowledgeUpdateServiceRequest()
90          kus_remove.update_type = 0
91          kus_remove.knowledge.knowledge_type = 1
92          kus_remove.knowledge.is_negative = 1
93          kus_remove.knowledge.attribute_name = 'patient_at'
94          kv_remove_patient = KeyValue()
95          kv_remove_patient.key = 'p'
```

```
        kv_remove_patient.value = 'larry'
        kus_remove.knowledge.values.append(kv_remove_patient)
        kv_remove_waypoint = KeyValue()
        kv_remove_waypoint.key = 'wp'
        kv_remove_waypoint.value = curr_wp
        kus_remove.knowledge.values.append(kv_remove_waypoint)
        kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
        if not kuc(kus_remove):
            rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.
    get_name())

        kus_add = KnowledgeUpdateServiceRequest()
        kus_add.update_type = 0
        kus_add.knowledge.knowledge_type = 1
        kus_add.knowledge.attribute_name = 'patient_at'
        kv_add_patient = KeyValue()
        kv_add_patient.key = 'p'
        kv_add_patient.value = 'larry'
        kus_add.knowledge.values.append(kv_add_patient)
        kv_add_waypoint = KeyValue()
        kv_add_waypoint.key = 'wp'
        kv_add_waypoint.value = msg.data
        kus_add.knowledge.values.append(kv_add_waypoint)
        kuc = rospy.ServiceProxy('/rosplan_knowledge_base/update',
    KnowledgeUpdateService)
        if not kuc(kus_add):
            rospy.logerr("KCL: (%s) Knowledge was not added!" % rospy.
    get_name())

        rospy.set_param('patient_pos',str(msg.data))
        rospy.set_param('patient_has_moved', True)

    return ret_value

def req_has_medicine():
    try:
        result = get_kb_attribute("has_medicine")
    except:
        return None
    if len(result) > 0:
        if result[0].is_negative:
            number = random.randint(0,10)
            if number < 6:
```

```
136                 return GetAttributeServiceRequest('has_tried_to_give')
137             else:
138                 return GetAttributeServiceRequest('has_medicine')
139         else:
140             return GetAttributeServiceRequest('has_medicine')
141     else:
142         return None
143 
144 
145 def req_has_taken():
146     try:
147         result = get_kb_attribute("has_taken")
148     except:
149         return None
150     if len(result) > 0:
151         if result[0].is_negative:
152             number = random.randint(0,10)
153             if number < 6:
154                 return GetAttributeServiceRequest('has_told_to_take')
155             else:
156                 return GetAttributeServiceRequest('has_taken')
157         else:
158             return GetAttributeServiceRequest('has_taken')
159     else:
160         return None
161 
162 def req_has_called():
163     try:
164         result = get_kb_attribute("has_called")
165     except:
166         return None
167     if len(result) > 0:
168         if result[0].is_negative:
169             number = random.randint(0,10)
170             if number < 6:
171                 return GetAttributeServiceRequest('has_told_to_call')
172             else:
173                 return GetAttributeServiceRequest('has_called')
174         else:
175             return GetAttributeServiceRequest('has_called')
176     else:
177         return None
```

Bibliografia

[1] C. K. Hunt, «Concepts in caregiver research,» *Journal of nursing scholarship*, vol. 35, n. 1, pp. 27–32, 2003.

[2] nursetimes, *Il ruolo del caregiver nella gestione assistenziale*. indirizzo: https://www.google.com/url?sa=i&url=https%3A%2F%2Fnursetimes.org%2Fil-ruolo-del-caregiver-nella-gestione-assistenziale%2F49829&psig=AOvVaw2n2zR-ZJLAf_cL8_VVy4v7&ust=1695700533179000&source=images&cd=vfe&opi=89978449&ved=0CBAQjRxqFwoTCMDWkfHuxIEDFQAAAAAdAAAAABAE.

[3] D. P. Miller, «Assistive robotics: an overview,» *Assistive Technology and Artificial Intelligence: Applications in Robotics, User Interfaces and Natural Language Processing*, pp. 126–136, 2006.

[4] linkedin, *Healthcare Assistive Robots Market: Caring hands and Technology enabled service*. indirizzo: https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.linkedin.com%2Fpulse%2Fhealthcare-assistive-robots-market-caring-hands-enabled-archit-ajmera&psig=AOvVaw0s3SNYigHs1NYGffUP-Est&ust=1695686083810000&source=images&cd=vfe&opi=89978449&ved=0CBAQjRxqFwoTCPCVttr4xIEDFQAAAAAdAAAAABAE.

[5] E. Karpas e D. Magazzeni, «Automated planning for robotics,» *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 3, pp. 417–439, 2020.

[6] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng et al., «ROS: an open-source Robot Operating System,» in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[7] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos e M. Carreras, «Rosplan: Planning in the robot operating system,» in *Proceedings of the international conference on automated planning and scheduling*, vol. 25, 2015, pp. 333–341.

[8] kcl planning, *Rosplan Overview*. indirizzo: https://kcl-planning.github.io/ROSPlan/documentation/images/rosplan_demo_system.png.

[9] ——, *Knowledge Base*. indirizzo: https://kcl-planning.github.io/ROSPlan/documentation/images/rosplan_knowledge_base.png.

[10] ——, *Problem Interface*. indirizzo: https://kcl-planning.github.io/ROSPlan/documentation/images/rosplan_problem_interface.png.

[11] ——, *Planner Interface*. indirizzo: https://kcl-planning.github.io/ROSPlan/documentation/images/rosplan_planner_interface.png.

[12] ——, *Parsing Interface*. indirizzo: https://kcl-planning.github.io/ROSPlan/documentation/images/rosplan_parsing_interface.png.

[13] ——, *Plan Dispatch*. indirizzo: https://kcl-planning.github.io/ROSPlan/documentation/images/rosplan_dispatch_interface.png.

[14] ——, *Simulated Actions*. indirizzo: https://kcl-planning.github.io/ROSPlan//tutorials/tutorial_04.png.

[15] ——, *Sensing Interface*. indirizzo: https://kcl-planning.github.io/ROSPlan//tutorials/sensing.png.