UNIVERSITA' DEGLI STUDI ROMA TRE

FACOLTA' DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica

Tesi di laurea

PROGETTAZIONE E IMPLEMENTAZIONE DI UNA INTERFACCIA GRAFICA PER SISTEMI DI CONTROLLO IN SCENARI DI EMERGENZA

Relatore

Prof. Carla Limongelli

Correlatore

Ing. Andrea Orlandini

Candidato

Fabrizio Mosconi

200738/61

Anno Accademico 2005/2006

ai miei genitori

Indice

1 **Introduzione** 8

2 **HRI: Human Robot Interaction** 12

2.1 Introduzione 12
2.2 Ecologia del Sistema HRI 12
2.2.1 Le Persone 12
2.2.2 La Scena del Soccorso 14
2.2.3 I Robot 16
2.3 Un Modello di Flusso per l'HRI 17
2.3.1 Prendere Decisioni 18
2.3.2 Dai Dati alle Informazioni 19
2.3.3 LOVR 21
2.3.4 Problem Holder 22
2.4 I Soccorsi alle Torri Gemelle 22

3 **HRI: Human Robot Interface** 26

4 **Prolog e Golog** 30

4.1 Introduzione 30
4.2 Situation Calculus 30
4.3 Rappresentazione Del Mondo 31

4.4 Prolog 34
4.5 Golog 36
4.5.1 Concurrent Temporal Golog 38

5 Mappe Per Robot Mobili **39**
5.1 Rappresentazione Topologica 39
5.2 Tipologie di Rappresentazione Spaziale 40
5.2.1 Decomposizione Spaziale 41
5.2.2 Rappresentazione Geometrica 42
5.3 Tipologie Di Mappe 43
5.4 Path Planning 44
5.4.1 A* Per Mappe 44
5.5 Inseguimento del Cammino 46
5.6 I Campi Potenziali 47
5.6.1 Introduzione 47
5.6.2 Calcolo delle Distanze 49
5.6.3 I Minimi Locali 49
5.6.4 Wavelet Plannet: Fronti d'Onda 50
5.6.5 Algoritmo di Trasformazione delle Distanze 52
5.6.6 Robot on the Edge: Attenti ai Bordi 54
5.6.7 Conclusioni 55
5.7 I Diagrammi di Voronoi 56
5.7.1 Introduzione 56
5.7.2 Sketch Map 58
5.7.3 Algoritmo di Voronoi 59
5.7.4 Mappa Topologica 59

6 SLAM **61**

6.1 SLAM: Introduzione 61

6.2 SLAM: Descrizione 63

6.3 SLAM: il Filtro di Kalman 65

6.3.1 Modello Matematico del Filtro 66

6.3.2 Il Filtro di Kalman Esteso 68

7 Architettura Doro **71**

7.1 Introduzione 71

7.2 Livello Esecutivo 73

7.3 Livello Funzionale 75

8 Iniziativa Mista **79**

8.1 Interazione Robot-Operatore 79

8.2 Livello Fisico 82

8.3 Pianificatore 83

8.4 Execution Monitor 86

8.5 Processo C++ 86

8.6 Iniziativa Mista: Introduzione 87

8.7 Gestione del Futuro 88

8.8 Verifica del Piano 91

9 Visualizzatore **98**

9.1 Introduzione 98

9.2 OnInit() 99

9.2.1 MyFrame() 100

9.3 Lettura da File .txt 100

9.4 Costruttore MyFrame() 100

9.5 I Due Pannelli 101

9.5.1 BigPanel() 101
9.5.2 StartPanel() 103
9.6 Le Funzioni 105
9.6.1 OnBarDbClick() 105
9.7 LeftClickFrame() 107
9.8 ReadPlan() 108
9.8.1 LockModify()/UnLockModify() 111
9.8.2 LockBar()/UnlockBar() 112
9.8.3 Funzioni richiamate da ReadPlan() 112
9.8.4 AddTick 114
9.8.5 GetSlamPlan()/GetNavPlan()/GetVisioPlan() 114
9.9 BigGrid 115
9.9.1 GoGrid() 117
9.10 Lettura da File di Testo Nomi dei Task 117
9.11 AddTask() 119
9.12 AddPlan() 122
9.12.1 GetPlanTaskString 125
9.12.2 DeletePlan() 126
9.13 EndTask() 127
9.13.1 LockNewPlan() 128
9.14 OnRightClick() 128
9.14.1 Tick() 130
9.15 Start()/ZoomIn()/ZoomOut()/Fit() 131
9.16 Il Piano 131
9.17 Le Macro 133

10 Conclusioni 134

11 Appendice **136**

11.1 Fireball.h 136

11.2 Fireball.cpp 145

12 Ringraziamenti **189**

Bibliografia **191**

12.1 Codice 192

Elenco delle figure

2.1 Hot Zone, Warm Zone, Cold Zone 14
2.2 Modello di Flusso per l'HRI 17
2.3 Flusso di Informazioni 20

5.1 Diagrammi di Voronoi: Ostacoli 57
5.2 Triangolazione di Delaunay 57

6.1 SLAM: Regioni di interesse 62

7.1 Architettura DO.RO. - Domestic Robot 71

9.1 Visualizzatore 99

Capitolo 1

Introduzione

L'obiettivo di questa tesi è stato quello di realizzare un'interfaccia grafica per un robot che opera in scenari di emergenza, ovvero in aree di soccorso urbano, da solo o in team con dei soccorritori umani.

Spesso, nelle zone in cui si verificano catastrofi naturali o altre sciagure, i soccorritori umani non riescono ad accedere a causa di problemi strutturali, quali le limitate dimensioni, l'inagibilità, il pericolo di crolli degli ambienti coinvolti nel disastro, o per la presenza di fattori ad alto rischio per l'uomo, come incendi, gas tossici e radiazioni. Per tutti questi motivi, i team di soccorso utilizzano dei robot di soccorso. Questi robot permettono loro di esplorare le zone inaccessibili e pericolose, per rilevare eventuali vittime e superstiti, e prestare i primi soccorsi.

Il team di soccorso controlla il robot a distanza e spesso si verifica il problema della perdita di controllo del mezzo. Questi problemi di comunicazione possono verificarsi a causa dei segnali che non arrivano a destinazione, oppure a causa di mancanza di governabilità, causata dalla limitata percezione da parte degli operatori dell'effettiva situazione, in cui si trova il robot.

Per questi motivi ha assunto una importanza fondamentale la proget-

tazione e l'implementazione di interfacce grafiche che permettano, in un feedback continuo, il controllo del robot da parte dell'operatore, che, visivamente, può rendersi conto della situazione del robot, e intervenire per fornire istruzioni e correzioni al suo operato.

L'architettura utilizzata è quella DO.RO. (Domestic Robot), testato più volte durante le passate edizioni della RoboCup Real Rescue Competition [7]. Durante queste prove si è potuto constatare che uno dei fattori principali di buon esito di una missione di soccorso è dato dall'interazione operazione-robot. L'architettura comprende una Gui, il robot virtuale Pioneer, lo State Manager, il Task Manager e il Pianificatore (*Planner*). Quest'ultimo è lo strumento che permette all'operatore di interagire con il robot.

Allo scopo dunque di facilitare l'interazione operatore-robot, lo scopo della tesi è stato quello di progettare e realizzare un visualizzatore, che permetta la visualizzazione dei task svolti dal robot ed inseriti dall'operatore.

Tale visualizzatore è stato progettato e implementato con l'ausilio delle librerie grafiche, open source, *wxWidgets*. Queste, molto versatili da un punto di vista grafico, sono utilizzate mediante il linguaggio C++ nell'ambiente di sviluppo Microsoft .Net, e hanno permesso la realizzazione del visualizzatore, che permette l'analisi dei task relativi al piano di navigazione, nei moduli di SLAM, Navigazione e Visione.

In questo modo, con l'interazione tra la Gui e il visualizzatore, l'operatore può effettivamente rendersi conto di quello che il robot sta eseguendo, in modo tale da poter effettuare delle modifiche e correzioni durante l'esplorazione. Questo è fondamentale se si opera con il robot in scene soccorso, dove è importante avere un feedback quasi istantaneo dell'operato del robot.

Nel primo capitolo della tesi, è stato analizzato lo scenario dei soccorsi in ambito urbano, approfondendo in particolare la composizione del gruppo di soccorso. E' stata analizzata la sua composizione; è stata esaminata l'interazione fra i suoi membri e il flusso delle informazioni in una scena del soccorso; è stato descritto l'utilizzo dei robot di soccorso, e l'influenza che le loro esplorazioni hanno sulle decisioni del team dei soccorritori; sono stati evidenziati i punti di forza e di debolezza dei team di soccorso, in relazione con gli scenari che devono affrontare. In particolare è stata riportata l'analisi svolta in [1, 2] dello scenario occorso durante i soccorsi prestati dopo il ben noto attentato alle Torri Gemelle a New York, l'11 settembre 2001.

Nel capitolo successivo, è stata descritta e analizzata la struttura generale di una interfaccia grafica, che permetta di guidare un robot a distanza, allo scopo di evidenziarne le caratteristiche più comuni e salienti, per realizzare una struttura di facile uso per l'operatore del robot.

Nel capitolo 4, sono stati descritti i due linguaggi di Intelligenza artificiale Prolog e Golog, che con le loro istruzioni permettono una navigazione intelligente al robot. E' stata offerta una panoramica sulle tecniche e strategie di pianificazione, con interesse particolare relativo ai metodi di costruzione delle mappe per robot mobili e al loro utilizzo.

In particolare è stato descritto dal punto di vista applicativo e dal punto di vista formale, il metodo di SLAM, "*simultaneous localization and map-building*", che permette al robot la navigazione, l'esplorazione e la costruzione della mappa dell'ambiente in cui è utilizzato.

Nell'ambito più propriamente pratico, è stata descritta l'architettura DO.RO. nelle sue componenti utilizzate nell'interfaccia: la Gui di controllo del robot, il robot virtuale Pioneer, lo State Manager, il Task Manager, il Pianificatore. Quest'ultimo permette, attraverso un menù di poter inserire e cancellare delle

azioni, per modificare o confermare il piano di navigazione del robot.

Nel capitolo successivo è stata descritta in dettaglio la struttura e il funzionamento del visualizzatore, con le relative funzioni.

Capitolo 2

HRI: Human Robot Interaction

2.1 Introduzione

HRI è l'acronimo per Human Robot Interaction. Gli argomenti di interesse principale nell'HRI sono: i task; l'ambiente in cui avvengono i soccorsi, e si muovono i soccorritori e i robot; i soccorritori e la loro interazione [2].

2.2 Ecologia del Sistema HRI

2.2.1 Le Persone

In [2] è stata presentata una analisi degli scenari di emergenza in cui sono state definite le categorie di persone coinvolte nelle operazioni di soccorso: i soccorritori e i sopravvissuti stessi. Il team di persone che prestano i soccorsi, e i loro mezzi, compresi i robot, vengono denominati team USAR (Urban Search And Rescue). I team USAR svolgono attività di ritrovamento, salvataggio e primo soccorso alle vittime. In riferimento ai soccorsi prestati dopo l'attentato delle Torri Gemelle, da parte di squadre di soccorso federali,

è stato possibile costruire dei modelli per i team USAR. Questi team di solito hanno diversi scopi:

- Ricerca (Search);
- Salvataggio (Rescue);
- Cure (Medical);
- Bonifica (HazMat);
- Logistica (Logistic);
- Pianificazione (Planning).

Ogni team USAR è guidato da un Manager, che è affiancato da specialisti. Se vengono usati dei robot, essi fanno riferimento al capo del team a cui sono assegnati.

Di norma, i componenti dei team di soccorso sono dei pompieri della città interessata, maschi, con vari livelli di istruzione e possono essere sospettosi delle nuove tecnologie, soprattutto perché timorosi di essere soppiantati dai robot nel loro lavoro.

Inoltre dobbiamo considerare il fatto che una buona parte del team di soccorso può essere composta da specialisti civili.

Tutti questi fattori giocano un ruolo fondamentale nell'accettazione dei robot come parti integranti del team di soccorso e non come una parte separata dell'operazione.

2.2.2 La Scena Del Soccorso

Nel momento dell'intervento l'USAR team opera in tre zone tra loro comunicanti: Hot Zone, Warm Zone e Cold Zone.

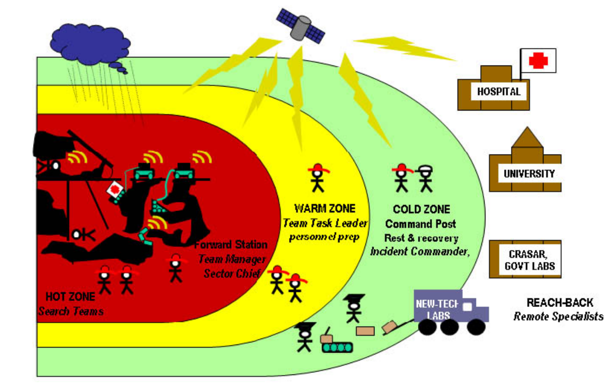

Figura 2.1: Hot Zone, Warm Zone, Cold Zone

- **Hot Zone**: la zona calda, l'area dell'attuale devastazione. L'accesso a questa zona deve essere controllato e limitato, se possibile.

  In questa zona operano, tra gli altri, il Task Force Leader e il Search Team Manager;

- **Warm Zone**: l'area limitrofa alla zona calda, dove il team di soccorso si raduna e si organizza. Per entrare in questa zona, i soccorritori devono passare attraverso una serie di decontaminazioni. Quest'area serve anche per organizzare evacuazioni di emergenza;

- **Cold Zone**: la zona fredda, che circonda la Warm Zone ed è ristretta ai soccorritori. È la zona del Quartier Generale dei soccorritori, il luogo dove si forniscono informazioni ai media, si fanno attendere i parenti delle vittime, si ristorano, riposano e ricaricano i soccorritori. Risorse

remote possono essere accedute tramite comunicazioni via telefono o internet.

La zona principale delle ricerche è la Hot Zone, e al suo interno bisogna ricercare le cavità o le fessure tra le macerie. I primi tre giorni sono fondamentali nelle ricerche, in quanto finalizzati alla ricerca di eventuali superstiti, in particolare le prime 48 ore.

Cavità (Void): esse sono fondamentali perché rappresentano l'unico accesso all'interno delle strutture urbane e perché rappresentano l'unica via per poter ritrovare vittime e sopravvissuti. Proprio a causa della loro dimensione e della posizione spesso poco accessibile, si prestano all'uso dei robot di soccorso.

Le cavità vengono suddivise a loro volta a seconda delle loro condizioni. Ci possono essere spazi semi-strutturati, che ancora ricordano, seppur parzialmente parti di edifici, come porte o trombe di scale, e possono ancora sopportare l'ingresso delle persone; spazi ristretti: l'accesso è ancora possibile da parte di persone, ma molto rischioso; spazi chiusi, inaccessibili ai soccorritori umani e animali. Le ultime due classificazioni sono quelle che interessano l'uso dei robot di salvataggio.

All'interno della **Hot Zone** bisogna tener conto anche di altri aspetti non secondari. Questa zona rappresenta a tutti gli effetti un pericolo costante anche per i soccorritori (crolli, strutture pericolanti, materiale metallico sporgente, etc), e quindi i soccorritori devono a loro volta proteggersi dai pericoli circostanti.

Questa situazione può ostacolare, insieme alla difficoltà di accesso, in modo rilevante il lavoro dei soccorritori. Ad esempio, per un operatore è difficile muoversi con tuta, respiratore e protezioni su mani e piedi.

A ciò si aggiunge anche la difficoltà nel percepire le condizioni reali della Hot Zone, che può essere buia o poco illuminata, ricoperta di polvere, e non permette di capirne l'effettiva pericolosità.

Un'altra fonte di stress per i soccorritori è dovuta alla mancanza di comunicazione. Gli operatori infatti sono dispiegati in zone molto vaste e i palazzi e le rovine sono spesso degli schermi alle onde radio.

Ne consegue che spesso i team di soccorso devono lavorare in modo isolato l'uno dall'altro, comunicando solo le informazioni strettamente necessarie e tutto ciò aumenta la sensazione di isolamento e di pericolo.

2.2.3 I Robot

Attualmente, nessun robot è costruito in modo specifico per gli scenari USAR. Infatti la specificità delle situazioni di soccorso non dà garanzie sulla configurazione dei robot da usare nelle operazioni di soccorso, per penetrare all'interno di cavità.

In generale, è possibile distinguere i robot in mini-robot e in micro-robot. I primi hanno dimensioni ridotte (per esempio possono essere trasportati in alcune scatole), cosa assai utile, perché i soccorritori devono avere le mani libere per lavorare durante le operazioni di intervento. I secondi, invece, possono essere anche contenuti in una scatola di scarpe e possono entrare in zone non accessibili alle persone.

2.3 Un Modello di Flusso per L'HRI

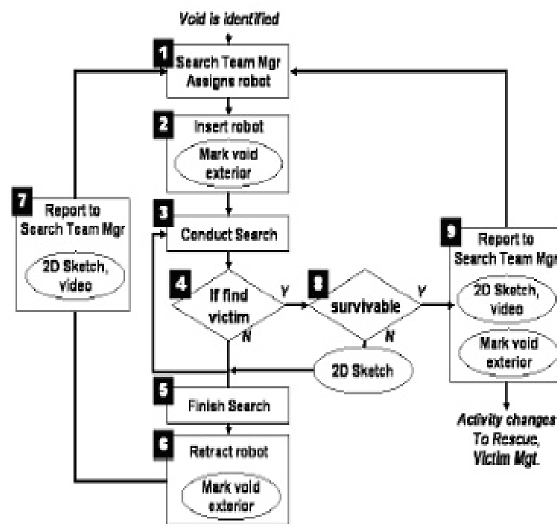

Figura 2.2: Modello di Flusso per l'HRI

In base al modello in figura 2.2, possiamo evidenziare gli aspetti principali delle operazioni di ricerche e salvataggio di un team di soccorso. Nella figura seguente è possibile osservare il flusso di interazioni tra i membri del team, in particolare i robot e i suoi operatori.

Nella figura, notiamo per primo il flusso principale delle informazioni in un team USAR. I rettangoli rappresentano le varie attività, gli ovali le informazioni dirette ad altri, i rombi le decisioni prese o le alternative.

Iniziando dall'alto, quando viene identificata una cavità, il Search Team Manager, ovvero il responsabile del team di ricerche, deve decidere quali risorse impegnare nell'esplorazione. Sta a lui anche la scelta del robot da impiegare, scelta che dipende dalla situazione che si presenta, dal parere degli eventuali esperti presenti e dalla disponibilità dei robot affidati al gruppo.

Questa è una fase critica, poiché i componenti del gruppo potrebbero non essere addestrati a interagire con il robot. In questo caso si suppone che gli utilizzatori del robot siano accompagnati da un esperto.

Nella figura, il flusso principale di informazioni va dal passo 1) al passo 6): dall'assegnazione del robot da parte del Search Team Manager al recupero del robot. Prima di questi passi, il team si occupa di marcare la cavità, con segnali che evidenzino che la ricerca è in in fase di inizio, e inseriscono il robot.

La ricerca va avanti finché il robot non incontra una vittima 4). A questo punto il flusso si sposta sul ramo decisionale con i passi 8) e 9); oppure il robot può essere arrivato in un punto in cui non può avanzare 5), e quindi la ricerca è conclusa. Quando il robot ha esplorato la cavità viene riportato indietro 6), e la cavità è di nuovo marcata con segnali che indichino l'avvenuta esplorazione. Infine viene compilato un rapporto per il Search Team Manager, e vengono esaminati i filmati, quando il robot rientra alla base delle operazioni 7).

Come si evince dalla figura, la conoscenza acquisita sul campo deve essere trasformata in informazioni, per i membri del team di soccorso. In particolare le informazioni raccolte sul campo possono essere comunicate tramite segni esterni, lasciati nelle zone già esplorate, utili per team di ricerca che non sanno se hanno raggiunto un'area più o meno esplorata; tramite comunicazioni a voce o scritti. Queste informazioni permettono anche di ricostruire il percorso effettuato dal robot o dal team di soccorso qualora non si abbiano informazioni sufficienti a riguardo.

2.3.1 Prendere Decisioni

Nella figura 2.2, al punto 4), il nodo decisionale si verifica quando incontriamo una vittima. Deve essere appurato quale sia il suo stato (co-

sciente, svenuta, etc), punto 8). Mentre i superstiti coscienti sono facili da riconoscere, fare considerazioni sullo stato nel caso in cui non ci siano segni tangibili di vita, comporta decisioni delicate che devono essere affrontate sfruttando appieno le capacità umane e tecnologiche a disposizione. Nel caso in cui si pensa di non poter fare niente in soccorso alla vittima incontrata, viene lasciato un segno sul luogo del rinvenimento, e la ricerca va avanti.

Se viene valutata la presenza di un sopravvissuto si informa un team di esperti, e a questo punto bisogna effettuare importanti scelte. Quello che è più importante è decidere come ottenere più informazioni possibili, per accertare se effettivamente il robot si trova davanti a un sopravvissuto, e in questo caso può essere decisivo il team di esperti. Se questa ipotesi è verificata, la strategia di ricerca cambia, punto 9), e bisogna usarne un'altra.

In questo caso è opportuno fare uno disegno della zona in cui è presente il sopravvissuto, nel caso in cui il team di soccorso debba evacuare la sua zona; successivamente interverranno gli uomini addetti all'effettivo recupero e cura del sopravvissuto.

2.3.2 Dai Dati alle Informazioni

E' possibile individuare inoltre un altro modello di flusso, che evidenzia la trasformazione dei dati, raccolti dai vari team, in informazioni valide per tutti i soccorritori.

La sorgente dei dati è il team di soccorso. In particolare, il robot fornisce dati raccolti sul campo trasformati in informazioni per il team di ricerca. Come vediamo dalla figura l'interazione tra i vari team, il problem holder, gli operatori e il robot è continua.

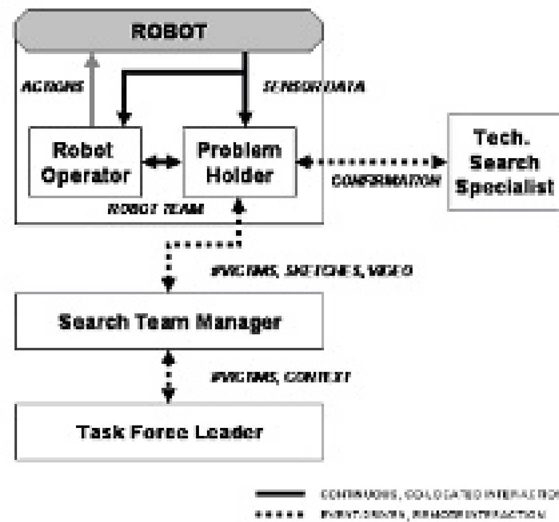

Figura 2.3: Flusso di Informazioni

Gli operatori usano i dati raccolti dal robot stesso per farlo muovere. Il problem holder usa gli stessi dati per ricercare eventuali segni di vittime o superstiti. L'obiettivo primario del team di ricerca è ritrovare eventuali sopravvissuti. In questo caso è il problem holder che interpreta i dati del robot e si preoccupa di trasformarli in informazioni per i team di ricerca tramite sketch o filmati.

I due eventi che possono verificarsi sono:

1. ritrovamento di una vittima;
2. completa esplorazione di una cavità.

Mentre il team di ricerca di un singolo robot si preoccupa soltanto della zona assegnatagli, il **Search Team Manager** e il **Task Force Leader** si preoccupano di tutta la Hot Zone. Il Search Team Manager si preoccupa di

raccogliere le informazioni da tutti i singoli team di ricerca. Il Task Force Leader è quello che, una volta ricevute le informazioni, decide la strategia di soccorso per ogni zona, in base a quelle informazioni, e alle risorse disponibili.

2.3.3 LOVR

Data la loro importanza le informazioni principali sono di tipo visivo. Una delle *strategie di ricerca* principali è la LOVR, che è un acronimo inglese *Localize, Observe General Surroundings, Look Specifically for Victims, Report* [2]. Essa è strutturata, come indica il nome stesso, in quattro operazioni.

Usando questa strategia, l'operatore indirizza il robot nelle cavità e stima distanza e direzione. Negli spazi ristretti, spesso non c'è modo di scegliere la direzione, ma se questa possibilità esiste, allora si segue un algoritmo di tipo "*right-wall following*", simile alla ricerca che effettuerebbe un uomo.

I due operatori osservano il nuovo insieme di elementi visibili dalla nuova posizione, e i sopravvissuti.

Questa osservazione a sua volta aiuta nel riconoscimento di altre caratteristiche della cavità, come ad esempio soffitti pericolanti, altrimenti non rilevabili.

Il robot è poi guidato a eseguire un'ulteriore osservazione dell'area, in ogni direzione, con sensori termici, o altre modalità, per trovare eventuali vittime. A questo punto, l'operatore principale che guida il robot comunica all'altro operatore ciò che sta visualizzando il robot. Il secondo operatore si occupa di disegnare una mappa del persorso (*path*) del robot e degli ambienti incontrati.

Questa comunicazione tra i due permette di ridurre il disorientamento dell'operatore principale e di accertare che la ricerca sia completa.

Il LOVR è eseguito ogni 2-5 minuti, a seconda della complessità dell'ambiente esplorato.

2.3.4 Problem Holder

Il Problem Holder fornisce uno stato della situazione circostante, diverso dallo stato del robot, ovvero mantiene in memoria lo stato della zona in cui opera, opposta allo stato in cui si trova il robot. In particolare memorizza segnali di probabile presenza di vittime o sopravvissuti; fonti di calore, colori, suoni, movimenti; inoltre memorizza le mappe che il robot ha costruito nella sua esplorazione, controlla i punti di vista e cerca di sintetizzare il reale stato della situazione affrontata.

E' interessante analizzare alcuni aspetti pratici della Urban Search and Rescue, durante un evento tragico come l'attentato delle Torri Gemelle a New York, l'11 Settembre 2001.

2.4 I Soccorsi alle Torri Gemelle

Nei soccorsi prestati durante l'attentato alle Torri Gemelle, l'11 settembre 2001, sono stati utilizzati dei robot di salvataggio. Durante le operazioni di soccorso, sono stati realizzati dei filmati, che, una volta esaminati, hanno permesso di raccogliere ulteriori informazioni per migliorare il lavoro dei team di Urban Search and Rescue [1].

L'osservazione di quel tragico evento ha permesso di evidenziare dei comportamenti non corretti da parte degli uomini impiegati. Ha inoltre eviden-

ziato dei difetti di progettazione nei robot.

Nel controllare un qualsiasi robot di salvataggio sono necessarie due persone o due team di persone. I compiti principali infatti sono due. Il primo è l'individuazione di segni di vita di eventuali sopravvissuti. Il secondo è il controllo del robot. E' evidente che questi due compiti devono essere svolti in maniera separata, ma parallela.

Questi due team devono dunque comunicare tra loro. Inoltre tutti i team di controllo dei robot devono comunicare tra loro. I dati raccolti da ogni singola macchina devono essere inviati attraverso opportuni protocolli a un server centrale. Quest'ultimo, situato in una zona esterna all'evento, deve provvedere alla loro conservazione. Inoltre è da ricordare che la comunicazione tra i vari team non può e non deve limitarsi soltanto allo scambio dei dati raccolti da robot.

Dal punto di vista dei miglioramenti delle funzioni dei robot, l'esperienza sul campo ha dimostrato che le macchine sono avvantaggiate rispetto a eventuali soccorritori umani e animali in certe condizioni. Infatti i robot possono addentrarsi in anfratti e superfici non raggiungibili, date le piccole dimensioni, e non risentono di particolari condizioni ambientali, come incendi e aria irrespirabile o tossica. I robot vengono utilizzati anche per accertare le condizioni di stabilità della struttura, prima di eventuali soccorsi da parte dell'uomo.

Per quanto riguarda i dati raccolti e il modo in cui tali dati venivano analizzati dagli operatori, è stata evidentemente la mancanza di un adeguato

numero di sensori su ogni robot. Questo ha portato gli operatori a fidarsi troppo delle immagini visive raccolte dalle telecamere montate sui robot.

Tra i difetti riscontrati nella progettazione dei rescue robot, i seguenti sono i più comuni:

1. I robot vengono progettati in modo da arrampicarsi, ma spesso non sono in grado di scendere attraverso i cunicoli nel terreno oppure all'interno delle strutture crollate.

2. Molti algoritmi di navigazione e mappatura necessitano di sensori di una certa dimensione. Molto spesso queste non sono adatte alle esigenze di questi scenari, in cui i robot devono intrufolarsi in anfratti e cavità, e le cui dimensioni non possono essere più grandi di una scatola di scarpe.

3. L'uso delle telecamere è molto utile per aumentare la percezione della visuale del robot, ma sono ancora troppo imprecise, e fonte di errori di valutazione da parte degli operatori.

4. Problemi di comunicazione. Questi sono spesso causati dalla volatilità dei mezzi di trasmissione. Durante i soccorsi, i robot che sfruttano la comunicazione wireless sono stati gli unici a essere persi. Infatti, non erano dotati di sistemi intelligenti in grado di riattivare il robot per ristabilire il contatto.

5. I robot spesso non sono protetti dall'azione di agenti chimici, biologici e radioattivi e quindi vulnerabili a danneggiamenti causati da questi materiali.

Le osservazioni precedenti evidenziano due tipi di relazioni. La prima è tra il team che guida il robot e il team dei soccorsi. La seconda è tra il team

dei soccorsi e i robot sulla scena dei soccorsi. Il primo team in generale si occupa di guidare il robot, rilevandone la posizione e analizzandone i dati. Il secondo, interagendo con il primo, cerca eventuali segni di sopravvissuti o di vittime.

Capitolo 3

HRI: Human Robot Interface

La Human Robot Interface (HRI) é l'interfaccia con la quale l'operatore entra in contatto con un robot e tutti i suoi dispositivi. Un umano (operatore) può prendere atto della posizione del robot, delle sue rilevazioni e ne può modificare, eventualmente, il comportamento.

Nel progetto di una HRI bisogna tenere conto di diverse caratteristiche:

1. L'aspetto dell'interfaccia dell'operatore.
2. Le funzionalità del sistema di controllo.
3. Stabilire una connessione tra il robot e il team di guida.
4. Familiarizzare con i comandi del robot, e farlo attraverso l'interfaccia che si è implementata.
5. Specificare e implementare le funzionalità specifiche, tipiche del robot, in particolare:
    - visualizzare la mappa che il robot costruisce durante la sua esplorazione;

    - permettere all'operatore di poter guidare il robot, con un click sull'obiettivo visualizzato sulla mappa;
    - sviluppare una navigazione libera da eventuali collisioni, verso il target specificato dall'operatore.

6. Integrazione della macchina per le riprese con l'interfaccia di visualizzazione.

7. Documentare il lavoro svolto.

L'interfaccia, che dovrà essere utilizzata dall'operatore, è la parte visivamente più importante di un sistema HRI. Quello che deve visualizzare può essere descritto da questi quattro punti:

- **World Modeling**: il modello del mondo, ovvero l'area dove lavora il robot e la posizione del robot stesso;
- **Deliberation**: è un pannello che permette di seguire i processi ad alto livello, come path planning, strategie, monitoraggio;
- **Perception**: in questa sezione i dati raccolti sono trasformati in informazioni sugli oggetti e gli ostacoli, che il robot incontra nel suo cammino;
- **Control**: permette di seguire i movimenti a basso livello. Questi movimenti possono essere, ad esempio, lo spostarsi in un corridoio, oppure azioni più specifiche, ad esempio, la raccolta di un oggetto;
- **Virtual Robot**: è una rappresentazione del robot, nelle sue componenti interne e nei suoi sensori esterni.

Ognuna di queste componenti deve poter comunicare e scambiare dati e informazioni l'una con l'altra.

Un altro componente dell'interfaccia è il **local perceptual space**, che mantiene una rappresentazione dell'area in cui si trova a lavorare il robot.

Andando più nel dettaglio, esistono diversi tipi di rappresentazione delle mappe. Uno dei modelli più usati è quello della griglia [4]. La griglia è rappresentata da una matrice quadrata. Ogni singola cella della matrice, le cui dimensioni sono fissate a priori, contiene la rappresentazione di una certa area di spazio, in cui opera il robot.

Dal punto di vista dell'operatore che lo guida, è fondamentale sapere se in quell'area della mappa sono presenti o meno degli ostacoli. Questo dato è rappresentato dal valore zero o dal valore uno nella struttura della matrice. In particolare, di queste mappe ce ne sono tre. Ogni casella ha tre valori che rappresentano rispettivamente l'eventualità che la cella sia vuota, l'eventualità che la cella sia occupata e una combinazione delle prime due.

L'aggiornamento dei valori delle celle è possibile tramite l'uso di laser, sonar o altri sensori che permettono di percepire tali informazioni. In particolare, possiamo asserire che se il robot occupa una certa area, sulla mappa, ovvero una cella nella matrice, quell'area sarà sicuramente libera da ostacoli. Usando un sonar, ad esempio, si possono ottenere due informazioni che permettono di riempire i valori della prima e della seconda mappa: la rilevazione di un ostacolo permetterà di settare ad uno la cella nella mappa che contiene gli ostacoli, mentre la certezza dell'assenza di altri ostacoli permette di settare le celle tra il robot e l'ostacolo con tanti zeri.

Un'altra funzione specifica è quella di **path planning**, ovvero la pianifi-

cazione del percorso che deve seguire il robot. Si tratta di assegnare al robot una serie di coordinate, fino al punto esatto in cui il robot dovrebbe dirigersi. In generale il path che si vuole seguire consiste in molti piccole destinazioni (*goal*) successive.

Capitolo 4

Prolog e Golog

4.1 Introduzione

La pianificazione rappresenta l'attività finalizzata alla scoperta di una sequenza di azioni che, se eseguite, portano dalla situazione iniziale a una situazione obiettivo. Le informazioni necessarie per effettuare la pianificazione sono:

- Stato iniziale
- Descrizione delle azioni, con le condizioni della loro applicabilità
- Descrizione dell'obiettivo nel situation calculus un problema di pianificazione si risolve dimostrando che: $Assiomi| - ?sGoal(s)$

4.2 Situation Calculus

Uno dei formalismi più noti utilizzati per descrivere un agente ossia a qualunque cosa in grado di percepire e agire in un ambiente, un dominio dinamico, è il situation calculus [7]. Tale formalismo concepisce il mondo come

una sequenza di situazioni ognuna delle quali è il risultato dell'applicazione di un'azione su una situazione precedente. Lo stato del mondo, o meglio della realtà di interesse, è l'effetto della sequenza di azioni eseguite su di esso a partire da uno stato iniziale. Il situation calculus esprime i cambiamenti del mondo nella logica di primo ordine.

Nel situation calculus un problema di pianificazione si risolve dimostrando che: $Assiomi|-?sGoal(s)$. La soluzione sarà del tipo $S = do(a_1, do(a_2, do(a_3, S_0)))$ e quindi il piano sarà $< a_3, a_2, a_1 >$: per raggiungere l'obiettivo a partire dalla situazione iniziale, è necessario eseguire nell'ordine le azioni a_3, a_2 e a_1.

4.3 Rappresentazione Del Mondo

Il primo passo da fare nella fase di rappresentazione del mondo è quello di descrivere le relazioni o proprietà che non sono soggette al cambiamento. Per questo scopo è stato definito un gruppo di assiomi che rappresentano:

- **Le proprietà dei predicati statici**. In questo caso, le proprietà non necessitano dell'argomento situazione aggiuntivo, che abbiamo visto precedentemente riguardo ai fluenti:

  $\forall x(supermarket(x) \rightarrow sells(x, milk))$.

- **Le proprietà dei fluenti**:

  $\forall x \forall y \forall s(next(x, y, s) \rightarrow next(y, x, s))$

Quindi è necessario definire lo stato iniziale e l'obiettivo da raggiungere. In generale lo stato iniziale è rappresentato in forma dichiarativa con una congiunzione di formule atomiche che esprime la **situazione di partenza**:

$On(A, B, S_0) \wedge onTable(B, S0) \wedge onTable(C, S_0)$.

Allo stesso modo la **descrizione dell'obiettivo** è rappresentata dalla congiunzione dei fluenti che devono essere veri nella situazione obiettivo: $\exists s(on(A, B, s) \wedge on(B, C, s) \wedge onTable(C, s))$.

Di fondamentale importanza è la **descrizione delle azioni**. Per questo scopo vengono utilizzati in primo luogo gli **assiomi delle precondizioni**.

La formula seguente esprime le condizioni di applicabilità di un'azione in una determinata situazione:

$$\forall s \forall x_1, \cdots, \forall a(precond(x1, \cdots, x_n, s) \rightarrow Poss(a(x_1, \cdots, x_n)s))$$

dove a rappresenta l'azione considerata, le x rappresentano tutti i parametri di a, s rappresenta la situazione in cui può essere eseguita a, $precond(x_1, .., x_n)$ rappresenta le precondizioni per l'eseguibilità e $Poss(a, s)$ rappresenta il fatto che è possibile eseguire l'azione a nella situazione s.

In secondo luogo le azioni devono essere descritte specificando le proprietà della situazione, che risulta a seguito dell'esecuzione dell'azione stessa. Per rappresentare come il mondo cambia a partire da una situazione alla successiva, si utilizzano gli **assiomi degli effetti**.

Il formalismo utilizzato è il seguente:

$$Poss(a(x_1, \cdots, xn), s) \rightarrow postconda(x_1, \cdots, xn, do(a, s)).$$

Sfortunatamente questo tipo di assiomi non è sufficiente per descrivere in modo completo lo stato risultante dall'esecuzione di un'azione. Per fornire una descrizione completa è necessario introdurre, per ogni azione, anche una descrizione di ciò che rimane invariato a seguito della sua esecuzione. Pertanto risulta indispensabile introdurre gli **assiomi dei frame** che descrivono

come il mondo rimane lo stesso e sono del tipo:

$$\forall a \forall x \forall s(onTable(x,s) \wedge x \neq x' \wedge (a \neq afferra) \rightarrow onTable(x, do(a,s))).$$

Nel loro insieme gli assiomi degli effetti e quelli dei frame forniscono una descrizione completa di come il mondo evolve in risposta alle azione dell'agente. Tuttavia l'introduzione degli assiomi dei frame comporta la necessità di introdurre un numero di assiomi dell'ordine 2*|Fluenti|*| Azioni|; il problema da affrontare dunque è determinato dal fatto che questo numero può assumere valori elevati.

Pertanto per ottenere una rappresentazione più elegante, possiamo combinare gli assiomi degli effetti e dei frame al fine di ottenere un singolo assioma che descrive il modo di calcolare un predicato per l'intervallo di tempo successivo, dato il suo valore per l'intervallo di tempo corrente. Un tale assioma è necessario per ogni predicato che può cambiare valore col passare del tempo; prende il nome di assioma di stato successore ed è della forma:

Un predicato sarà vero:
dopo ⇔ *[un'azione lo ha reso vero*
∨ *già vero e nessun'azione lo ha reso falso]*

Quindi secondo l'assioma di stato successore, per ogni fluente $R(x1, \cdots, xn, s)$ si determinano le formule:

- $GR+(a, s, x1, \cdots, xn)$ che rappresenta tutte le condizioni che possono causare il passaggio di R da falso a vero;

- $GR-(a, s, x_1, \cdots, x_n)$ che rappresenta tutte le condizioni che possono causare il passaggio di R da vero a falso.

Quindi l'assioma di stato successore per $R(x_1, \cdots, x_n, s)$ sarà:

$\forall a, \forall x1, \cdots, \forall xnPoss(a, s) \rightarrow$

$R(x1, \cdots, xn, s), do(a, s)) \equiv$

$GR + (a, s, x_1, \cdots, x_n) \vee (GR + (x1, \cdots, x_n, s) \wedge (GR - (a, s, x_1, \cdots, x_n))$

Per ogni azione a, per ogni situazione s e per tutti gli oggetti $x_1, \cdots, x_n$, $R(x_1, \cdots, xn)$ vale nella situazione $do(a, s)$ se e solo se a è una delle azioni che provocano $R(x1, \cdots, x_n)$ oppure $R(x_1, \cdots, xn)$ è vero in s e a non lo modifica.

Per descrivere per esempio, il fluente *aperta* con un assioma di stato successore, si potrebbe utilizzare una formula del tipo:

$\forall taperta(x, do(a, s)) \Leftrightarrow a = aprire \vee (aperta(x, s) \wedge a \neq chiudere)$

4.4 Prolog

Il Prolog è un linguaggio di programmazione logica (PROgrammazione LOGica). È un linguaggio ideato per esigenze di intelligenza artificiale, pertanto si concentra più sul *cosa* invece che sul *come* si attiva il comportamento intelligente [7].

Un programma Prolog consiste in un insieme di procedure espresse in *clausole di Horn*, ossia clausole contenenti al massimo un letterale positivo, ed attivate da un'asserzione iniziale d'obiettivo. È costituito da un insieme di fatti che dichiarano un certo stato di cose, da un'insieme di regole che definiscono relazioni fra stati e cose, e da obiettivi a cui rispondere.

- I fatti e le regole sono clausole definite: contengono entrambe un letterale positivo ma le seconde si distinguono perchè contengono almeno un letterale negativo.

  - Fatto A (A è un atomo)
  - Regola $A \wedge \neg B_1 \vee \cdots \vee \neg B_n$

- Gli obiettivi invece sono clausole negative e come tali contengono soltanto letterali negativi; nel caso in cui un obiettivo non contenesse letterali, si parlerebbe di clausola vuota.
  - Obiettivo $\neg B \ \vee \cdots \vee \neg B_n$

Un programma logico viene eseguito a fronte di un goal: $? - G$.

Il significato di un programma Prolog definisce se un dato obiettivo è vero rispetto ad un dato programma e, in caso affermativo, per quali istanze di variabili esso risulta vero.

Per trovare le risposte alle domande, il Prolog esamina il programma partendo dall'inizio e cerca un fatto che possa essere uguale a quello contenuto nella domanda, sostituendo alla variabile un valore opportuno.

Il processo per rendere uguali i due termini si chiama **unificazione**; una variabile che assume un valore si dice istanziata. Nell'esaminare il programma, il Prolog fa riferimento al principio di risoluzione: dimostrare un teorema ricavando la clausola vuota, a partire dalle ipotesi (fatti o regole).

Il Prolog inoltre sfrutta un metodo di **refutazione**: il problema di determinare se, dato un insieme di formule S e una formula $A, S| = A$, viene ricondotto al problema di determinare se $S \bigcup \neg A$ è insoddisfacibile; fornisce una risposta se c'è refutazione e risponde *no* altrimenti.

Per dedurre logicamente un'informazione negativa da un programma sono necessarie regole speciali. La regola più importante è quella di negazione come fallimento: è una regola completa e corretta, controparte implementativa dell'assunzione di mondo chiuso ed è definita: *se un atomo chiuso A non è conseguenza logica di un programma, allora si inferisce notA.*

Secondo questa assunzione il mondo è chiuso nel senso che ogni cosa che esiste si trova nel programma o può venire derivata da esso. Di conseguenza, se qualcosa non si dovesse trovare nel programma, allora essa non risulterebbe vera, ma risulterà vera la sua negazione.

Uno dei modi per provare che un predicato non è conseguenza logica dei fatti e delle regole contenute nel programma, e quindi poter inferire la sua negazione, è mostrare che esiste una deduzione che fallisce per la negazione di quel predicato. La regola di negazione come fallimento è definita nel seguente modo: dato il predicato *not*(*Obiettivo*), se *Obiettivo* ha successo, allora *not*(*Obiettivo*) fallisce, altrimenti *not*(*Obiettivo*) ha successo.

Nei linguaggi imperativi un programma è costituito da una serie di passi che specificano in che modo il computer deve lavorare per risolvere il problema.

Essendo il Prolog, invece, un linguaggio descrittivo, un programma Prolog è costituito dalla descrizione del problema. Il Prolog inoltre è considerato all'interno della programmazione funzionale nella quale si rinuncia al concetto di comando e, in particolare, al comando di assegnamento. Di conseguenza viene a mancare anche il concetto di *oggetto*, poiché non esistono operazioni in grado di modificare lo stato.

4.5 Golog

Il Golog è un linguaggio di programmazione di alto livello, basato sul situation calculus, che integra il ragionamento, la percezione e le azioni. I programmi Golog pertanto sono costituiti da azioni primitive δ e azioni complesse: quest'ultime saranno valutate dall'interprete (Prolog) al fine di ottenere una serie di azioni primitive, le quali potranno essere eseguite fisicamente.

Le azioni complesse sono definite attraverso simboli *extralogici* (*while*, *if*, ...) che funzionano come abbreviazioni per espressioni complesse nel linguaggio del situation calculus.

Il linguaggio è basato su una versione estesa del situation calculus che permette una descrizione formale di azioni complesse del tipo:

- **Sequenze di azioni**: prima si esegua A e poi B: $(\delta_1; \delta_2)$.
- **Test**: ϕ è vera nella situazione corrente $(\phi?)$.
- **Azioni non deterministiche**: si esegue A oppure B $(\delta_1|\delta_2)$.
- **Indeterminismo** nel numero di iterazioni: un'azione viene eseguita zero o più volte δ^*.
- **Istruzioni condizionali** $if(p, \delta_1, \delta_2)$ dove p è un'espressione booleana. Se p è vera viene eseguita δ_1, altrimenti δ_2.
- **Iterazione**: $while(p, \delta)$ dove p è un'espressione booleana. Finche p è vera viene eseguita δ.
- **Scelta non deterministica** dell'argomento dell'azione: $(\pi x)\delta$ dove x è una variabile Golog e δ è un'azione che usa x. Questa istruzione sceglie un valore da assegnare a x ed esegue δ con tale valore.
- **Procedure**: qualunque delle precedenti espressioni costituisce il corpo di una procedura Golog (*proc(nome, corpo)*). Pertanto un'interprete Prolog per il Golog, contiene la definizione dei costruttori di tali azioni complesse.

Finora tuttavia, non sono state ancora considerate le proprietà fondamentali delle azioni reali:

- si sviluppano nel tempo,
- presentano una durata e spesso sono concorrenti $(\delta_1||\delta_2)$.

4.5.1 Concurrent Temporal Golog

Il **Concurrent Temporal Golog** è una estensione del Golog che mira a colmare queste lacune attraverso l'estensione degli assiomi del situation calculus sequenziale, non temporale, al fine di gestire il tempo. L'interprete è arricchito con alcune definizioni per gestire non solo singole azioni, ma anche insiemi di azioni. poiché ci possono essere azioni che occorrono simultaneamente. Ogni azione presenta un parametro aggiuntivo che esprime il tempo in cui essa comincia. Con questa semantica per rappresentare un'azione con una certa durata si utilizzano due azioni istantanee: una indicante l'inizio e l'altra la fine dell'azione. Per esempio per rappresentare un'azione con una certa durata, ad esempio $go(p_1, p_2)$, si utilizzano: $startGo(p_1, t_1)$ e $endGo(p_2, t_2)$.

Il TCGolog offre anche simboli funzionali per gestire in modo semplice il tempo; per esempio $time(\delta)$ denota l'istante in cui occorre l'azione δ, mentre $start(s)$ denota l'istante in cui inizia la situazione s. Golog offre vantaggi significativi: permette all'utente di concentrarsi sulle specifiche di alto livello e gestire domini complessi, offrendo la possibilità di utilizzare piani temporali flessibili. Esperimenti condotti hanno indotto ad usare tale linguaggio per costruire sistemi di controllo per robot, per la simulazione di sistemi dinamici, e per la programmazione di agenti intelligenti.

Capitolo 5

Mappe Per Robot Mobili

5.1 Rappresentazione Topologica

L'uomo si basa su mappe cognitive [4]: esse sono mappe di ambienti in larga scala, la cui conoscenza è data dall'osservazione ripetuta nel tempo, per trovare dei percorsi e per determinare la posizione relativa degli oggetti nell'ambiente stesso.

In generale questa rappresentazione non ha la forma di una mappa per orientarsi, così come siamo abituati a pensarla, ma è piuttosto un luogo dove immagazzinare la conoscenza acquisita di tali ambienti. Tali mappe sono frammentarie, irregolari, possono contenere imprecisioni, contengono tipi eterogenei di informazioni, sono gerarchiche, sono piuttosto selettive e soggettive.

Per un robot invece sono molto poche le informazioni che occorrono per la navigazione:

- Dove sono?
- Dove sono gli altri luoghi?

- Come posso arrivarci?
- Dove sono stato?
- Dove non sono stato?

Queste informazioni devono essere facilmente reperibili, in genere dai sensori, devono poter essere aggiornate facilmente, devono permettere un *path planning* efficace e di conseguenza anche un modo facile di spostarsi all'interno dell'ambiente e tra ambienti contigui.

E' possibile un feedback tra il robot e l'uomo, per capire differenze e affinità di percezione e rappresentazione dello spazio tra i due soggetti.

5.2 Tipologie di Rappresentazione Spaziale

Scegliere il tipo di rappresentazione dello spazio, che sarà adottata per la navigazione del robot è fondamentale, perché influenzerà il modo in cui l'operatore potrà interagire con il robot stesso. Questa rappresentazione può essere dei seguenti tipi:

- **Rappresentazione Metrica**: usa un sistema di coordinate assolute per la descrizione delle caratteristiche dell'ambiente;
- **Decomposizione Spaziale:** lo spazio è suddiviso in celle. Ogni cella contiene l'informazione se sia occupata o meno;
- **Rappresentazione Geometrica**: usa primitive geometriche per rappresentare lo spazio;
- **Rappresentazione Topologica**: usa una struttura a grafo. I nodi del grafo corrispondono a determinati posti e i rami sono i collegamenti tra questi posti;

- **Rappresentazione Ibrida**: una struttura con entrambe le caratteristiche.

Tra queste tipologie, due sono le più comuni.

5.2.1 Decomposizione Spaziale

La decomposizione spaziale utilizza una rappresentazione dell'ambiente sotto forma di griglia. Ogni cella ha un valore che specifica, in modo binario, se è occupata o meno. Possiamo parlare anche di una griglia di occupazione, i cui valori, per ogni cella, sono dati dal teorema di Bayes. Possiamo associare le celle a dei sensori, inizializzare le celle a un valore di 0.5, né occupate né vuote e aspettare il responso del sensore. Così sembrerebbe anche semplice, ma dobbiamo tener conto che, già durante l'acquisizione di questi dati, bisogna conoscere la posizione del robot nell'ambiente.

E' un doppio problema di localizzazione del robot nell'ambiente e di mappatura dell'ambiente in cui si trova il robot. Per questi motivi spesso insorgono incongruenze nei dati raccolti, spesso a causa di ritardi temporali, poiché le due operazioni richiedono tempo.

In base a queste osservazioni possiamo dedurre che la griglia di occupazione dovrà essere realizzata in modo da poter integrare facilmente e velocemente i dati acquisiti dai sensori. Mantenere i dati in memoria e aggiornarli comporta lo sviluppo di algoritmi di una certa complessità: occorrerà conoscere in maniera esatta la posizione del robot. Tutto questo comporta un gran consumo di spazio e genera problemi di inconsistenza dei dati e quindi inefficienze nel path planning.

Per evitare il consumo eccessivo di spazio e di risorse di calcolo, si può usare l'idea del *quadtree*, che sfrutta la suddivisione dello spazio in celle rettangolari (quadrate), finché non è interamente ricoperto.

5.2.2 Rappresentazione Geometrica

Usa delle primitive geometriche, che non sono altro che punti, linee, poligoni. Queste figure sono rappresentate da coordinate, che mostrano la loro posizione in un sistema di coordinate globali.

Questo tipo di rappresentazione è spesso usata per lo scan matching.

L'obiettivo è quello di determinare il numero di traslazioni e rotazioni necessarie per raggiungere o evitare quella determinata forma.

Compio due operazioni:

- **Preprocessamento**: pulitura e aggiustamento dell'immagine;
- **Matching** vero e proprio che è effettuato con tecniche di misurazione dell'errore.

Devo poter acquisire un numero sufficiente di informazioni, per evitare problemi di consistenza dei dati. Posso effettuare delle semplificazioni, partendo dalla posizione iniziale del robot e cercando poi localmente per il match migliore. Ricordiamo che queste operazioni vengono effettuate mentre il robot è in movimento, quindi devono essere effettuate in modo accurato per evitare che il robot si smarrisca. Inoltre il mondo e gli oggetti in esso presenti possono cambiare.

Quest'ultimo problema può essere risolto definendo formalmente la rappresentazione del mondo per il robot.

Consideriamo un grafo $G = (V, E)$.

Ogni nodo del grafo corrisponde a un determinato posto. Il robot deve poter sapere quando arriva in uno di quei posti, ovvero in uno di quei nodi. I bordi, edges, corrispondono alle connessioni tra questi nodi, ovvero i percorsi che il robot può seguire durante i suoi spostamenti.

Il grafo dovrà contenere delle informazioni per consentire al robot di spostarsi da un posto all'altro, e le distanze relative tra essi.

Una rappresentazione con grafi è ad esempio il **diagramma di Voronoi**, uno scheletro dell'ambiente che il robot dovrà attraversare. Per questo tipo di rappresentazione abbiamo bisogno di un path planning molto efficiente, ma senza la necessità di localizzazione esatta dei posti. L'ideale sarebbe utilizzare una rappresentazione ibrida che medi tra la rappresentazione geometrica e quella topologica.

5.3 Tipologie Di Mappe

In generale, i robot possono usare due tipi di mappe: mappe globali e mappe locali [4].

- **Mappa Globale**: mappa totale dell'ambiente; è efficace durante la pianificazione del percorso, per ottimizzarlo, ma sfortunatamente impegna numerose risorse di calcolo.
- **Mappa Locale**: più rapidi dal punto di vista dei conti, ma meno ricchi di informazioni, non permettono di ottimizzare il percorso.

Un robot può memorizzare un percorso in una regione sconosciuta. La memorizzazione può avvenire per mezzo di una struttura a grafo.

Questa struttura è costruita ricordando i punti salienti del percorso, un numero sufficiente di punti che ricoprano in modo significativo lo spazio libero.

5.4 Path Planning

Gli algoritmi di pianificazione descritti nelle sezioni precedenti, combinati con la tipologia di mapping che si giudica più soddisfacente, possono avere diverse applicazioni nella determinazione del percorso che si vuole assegnare al robot.

Per questi problemi, la pianificazione è ottenere un percorso ottimo, da un punto di partenza iniziale e a un punto di destinazione. Minimo è inteso come la distanza minima o il minimo numero di rotazioni fino alla destinazione.

Dal punto di vista delle tipologie, gli **algoritmi di path planning** possono essere così divisi:

- Metodi di ricerca nello spazio libero: mantenimento della massima distanza dagli ostacoli;
- Metodi di rappresentazione della geometria degli ostacoli;
- Metodi basati sulla discretizzazione dello spazio;
- Metodi basati sui campi potenziali artificiali.

Gli algoritmi esaminati nelle sezioni precedenti possono essere adattati ai tipi di mappature esaminati in questa parte. Ad esempio, A* può essere utilizzato sia dal al metodo di Voronoi sia dal metodo dei campi potenziali.

5.4.1 A* Per Mappe

A* è uno degli algoritmi che può essere utilizzato per il path planning. L'utilizzo di questo algoritmo è subordinato alla posizione del robot nello spazio libero. Tutto dipende, in questo tipo di rappresentazioni dall'orientamento del robot. Bisogna avere mappe bidimensionali per ogni orientamento θ_i del robot.

L'accrescimento degli ostacoli viene eseguito in modo più preciso.

Abbiamo il robot orientato in un certo modo, e mantenendo tale orientamento, vogliamo cercare un cammino per il robot. La mappa verrà creata mantenendo fissa la direzione del robot. A questo punto vengono accresciuti gli ostacoli secondo la seguente tecnica:

- Si prende un punto R, ovvero un vertice, sul poligono che rappresenta il robot;
- Si fa scorrere il robot sull'ostacolo che viene poi espanso;
- Tenendo fisso R nell'origine si effettuano due rotazioni sull'asse x e sull'asse y;
- · Si fa poi coincidere R con tutti i vertici dell'ostacolo e si aggiungono i vertici del robot, ottenendo una lista di vertici;
- Unendo tutti i vertici esterni si ottiene l'ostacolo esteso.

Una volta effettuata l'espansione dell'ostacolo, ottenuto il grafo dei vertici, si può costruire il grafo di visibilità che consiste nel collegare il punto di start, quello di goal e tutti i vertici tra loro visibili degli ostacoli.

Tale grafo contiene tutti cammini da start a goal, e a esso possiamo applicare A*.

Tra gli svantaggi del grafo di visibilità c'è la sua scarsa adattabilità a mondi dinamici. I cammini sono vicini agli ostacoli. Una variante che migliora la situazione è lo sfruttamento dei vertici per usare una triangolazione di Delaunay utilizzando come punti iniziali i vertici degli ostacoli espansi. Questo metodo fornisce il cammino collegando i punti di mezzo dei lati dei triangoli: il risultato è un cammino nel mezzo dello spazio libero.

Un altro algoritmo per la costruzione del cammino è l'**inseguimento del cammino**.

5.5 Inseguimento del Cammino

Il cammino prodotto è una lista di punti, ovvero un percorso totalmente geometrico: $start, p_1, p_2, ..., p_{n-1}, p_n, goal$.

Le caratteristiche del cammino dipendono dalla mappa:

- Presenza di ostacoli imprevisti;
- Ostacoli mobili;
- Perdita di localizzazione nel robot;
- Problemi dinamici per le troppe rotazioni;
- ... ...

Possiamo sfruttare un sistema di pianificazione locale, che porti il robot da p_{n-1} a p_n, evitando le collisioni e definendo un cammino con buone caratteristiche dinamiche.

In termini più semplici, se il robot dispone di sensori adeguati, può modificare il cammino mentre lo esegue.

Tra i metodi in grado di fare ciò c'è quello dei campi potenziali che si basano sulle forze virtuali da essi generate.

Abbinando questo metodo a una mappa è possibile anche trasformarlo in un metodo di pianificazione globale, che produce il cammino mentre lo esegue.

5.6 I Campi Potenziali

5.6.1 Introduzione

Tra i possibili metodi di navigazione robotica, uno dei più comuni è quello campi potenziali.

Il robot è un punto dello spazio, ovvero una particella sotto l'influsso di un campo potenziale artificiale le cui variazioni rispecchiano la struttura dello spazio libero. In pratica lo scopo di questo metodo è quello di far attrarre il robot dal goal e di farlo respingere dagli ostacoli.

Analiticamente bisogna definire una funzione potenziale, che sarà la somma di due funzioni, una funzione potenziale attrattivo del goal e una funzione potenziale repulsivo degli ostacoli.

Gli ostacoli nel mondo del robot sono pesati in base alla loro distanza, e si parla di metodo locale, poiché la funzione potenziale dipende solo dalla distribuzione degli ostacoli nell'intorno del robot.

Utilizziamo il massimo gradiente del potenziale e quindi il robot cerca di portarsi dove il potenziale è minimo. Più basso è il potenziale più vicini siamo al goal, che avrà potenziale zero: uso dei minimi locali.

Questo algoritmo combinato con la ricerca sui grafi può essere usato come tecnica di pianificazione:

- Se è noto l'intero ambiente allora la pianificazione del cammino può essere eseguita dal robot stesso;
- Il metodo consente di valutare allo stesso tempo sia gli ostacolo sia le zone libere.

Il potenziale utilizzato è la somma di due potenziali:

- Uno attrattivo, che rappresenta il goal;

- Uno repulsivo, che rappresenta i vari ostacoli nello spazio.

In questo modo vi è un solo minimo globale, che rappresenta il punto di goal.

Il potenziale attrattivo è di tipo quadratico, funzione della distanza euclidea in linea d'aria tra tutti i punti e l'obiettivo, e genera una forza attrattiva che va ad attenuarsi quando si avvicina al goal, ma diventa eccessiva, se ci si allontana troppo da esso. Si può rimediare utilizzando a una certa distanza dal goal un potenziale di forma lineare.

Il potenziale repulsivo è calcolato per ogni ostacolo e consente invece di limitare gli influssi degli ostacoli a una certa distanza. L'unica attenzione è quella di non considerare ostacoli troppo distanti che potrebbero alterare la traiettoria del robot senza alcun beneficio.

I campi potenziali sono ampiamente usati, perché la determinazione dei cammini, una volta nota la funzione potenziale del mondo, è direttamente eseguibile a bordo del robot, e mediante una singola funzione è possibile trattare sia spazio libero sia ostacoli.

Per iniziare, consideriamo come unità di misura il pixel. La velocità di spostamento sarà dunque di un pixel per unità di tempo. Ci spostiamo su una mappa bidimensionale, che nel nostro caso ha la forma di un quadrato. Con queste considerazioni di base, possiamo definire un campo potenziale.

Immaginiamo una grossa matrice, dove ogni pixel, rappresenta un'informazione tramite un numero, nelle circostanze considerate in quel momento. Ricordiamo che il nostro obiettivo è trovare un percorso dal punto di partenza al punto di arrivo, quindi le informazioni contenute in ogni pixel saranno manipolate.

Ipotizzando che l'obiettivo sia il più in basso possibile, nella matrice e

che il nostro punto di partenza sia invece il più in alto possibile, dobbiamo muoverci vero il basso, ovvero considerare solo quei pixel che hanno potenziale minore di quello in cui siamo attualmente. Assegniamo ai pixel che rappresentano gli ostacoli dei valori più alti di quelli dei pixel più alti nella matrice, in modo tale da escluderli a priori nella ricerca del cammino.

A questo punto ci servono dei metodi per far camminare il robot.

5.6.2 Calcolo delle Distanze

Come primo passo possiamo calcolare la distanza tra ogni pixel e l'obiettivo. La distanza calcolata sarà proprio il valore da assegnare al pixel, ovvero il suo potenziale. In questo caso, tenendo a mente le condizioni di partenza, non di deve tener conto degli ostacoli, proprio grazie al valore assegnato loro in precedenza.

Come distanza consideriamo quella più breve da un pixel all'obiettivo, ovvero la linea retta. Una volta eseguito il calcolo di tutte le distanze bisogna applicare il seguente algoritmo: analizzare il valore contenuto in ogni pixel adiacente e scegliamo il pixel con il valore potenziale più basso. Ci spostiamo in quel pixel e ripetiamo l'algoritmo, fino a raggiungere l'obiettivo.

5.6.3 I Minimi Locali

Questo procedimento però ha il rischio di arrivare fino in un punto morto, senza trovare l'obiettivo. Questo punto morto o di stallo è rappresentato da un minimo locale. Immaginiamo un pixel circondato da pixel con valori potenziali tutti maggiori del valore potenziale in esso contenuto. In quel caso l'algoritmo penserà, erroneamente, di aver raggiunto l'obiettivo. Questo inconveniente può presentarsi quando ci troviamo in presenza di pixel accessibili, ma con valori potenziali di ostacoli, sia in presenza di ostacoli.

Una prima strategia potrebbe essere quella di abbandonare l'uso dei campi potenziali, muovendosi in quella particolare situazione a caso, e ricalcolando poi la traiettoria, per cercare di uscire dal minimo locale. Il pixel in questione sarà marcato come "cattivo" per evitare di ripassarci. Questa soluzione comporta un certo rischio perché il movimento del robot sulla griglia è casuale.

Un modo più efficiente di uscire dal guado è quello della ricerca in profondità.

Risolviamo questo problema ricordando che questa ricerca in profondità può essere migliorata applicando un algoritmo di tipo *best-first-search*, assai diffuso nella soluzione di problemi di ottimizzazione.

I pixel della griglia saranno inseriti in un albero, la cui radice è il punto di partenza. Questa radice avrò 8 possibili direzioni. Il robot si muoverà valutando il valore potenziale minore del pixel e si muoverà nel pixel con il valore potenziale più basso, ma con l'accortezza di considerare solo le direzioni che non sono già presenti nell'albero. Continuiamo con questo procedimento esaminando tutte le foglie dell'albero. Una volta terminata tutta l'ispezione, esaminiamo nuovamente le foglie con i valori potenziali più bassi. Continuiamo finché non arriviamo all'obiettivo oppure finché non abbiamo esplorato tutte le foglie, senza trovare alcun pixel con un valore più basso in cui poterci spostare. Questo accade se la destinazione è irraggiungibile a partire dal punto di partenza.

5.6.4 Wavelet Planner: Fronti d'Onda

Come osserviamo, possiamo trovarci ancora in una situazione di stallo.

Per evitare questa situazione e per evitare del tutto i minimi locali, da cui, come abbiamo visto, non possiamo spostarci.

Cambiamo il modo di calcolare le distanze tra il pixel che rappresenta l'obiettivo e gli altri pixel della griglia. Assegniamo al pixel obiettivo il valore zero e a partire da quel pixel, ci spostiamo in orizzontale e verticale, numerando progressivamente i pixel sul cammino.

Spostandoci poi dal nostro punto di partenza, ovunque esso sia, ci accorgiamo di avvicinarci all'obiettivo poiché il valore potenziale dei pixel decresce a mano a mano che ci facciamo più vicini. Notiamo anche che in questo modo, ogni pixel che porta all'obiettivo avrà sicuramente un vicino con un valore potenziale minore su cui sarà possibile spostarsi. Questo metodo è conosciuto come espansione a fronte d'onda: wavefront planner.

Entrando più nel dettaglio, questo algoritmo è una ricerca in ampiezza, breadth first search, e si applica molto facilmente alle rappresentazioni dello spazio bidimensionali, effettuate tramite una griglia, ovvero una matrice di pixel, nel nostro caso.

I vantaggi di questo algoritmo possono essere così riassunti:

1. Gli ostacoli, nella rappresentazione della griglia hanno distanza infinita e valore zero;

2. Lo spazio libero ha un costo relativamente basso;

3. Spazi indesiderati, ma comunque attraversabili hanno un costo medio basso;

4. Questo algoritmo è una buona via di mezzo tra il di percorsi lunghi attraverso terreni più accessibili e il passaggio di percorsi brevi in terreni meno accessibili.

All'interno di questa strategia è utile presentare l'algoritmo di trasformazione delle distanze.

5.6.5 Algoritmo di Trasformazione delle Distanze

Il robot si muove in una griglia che permette di gestire sia gli ostacoli, sia il cammino libero. Questa griglia può essere una matrice, la cui dimensione del pixel è data dalla semi-dimensione del robot se il mondo è poco ostruito.

L'algoritmo che costruisce il cammino, vede il robot al centro del quadretto, oppure in un suo vertice, se parliamo di una griglia di punti. La mossa può essere scelta dall'algoritmo A*.

Se la mappa è regolare, possiamo riempire la stanza, ovvero la griglia, di valori numerici che diano indicazioni della lontananza o meno del robot dagli ostacoli e dal punto di goal. L'idea è quella di usare un'onda con un fronte di propagazione circolare, che raggiungerà il punto di start, se non ci sono ostacoli.

A questo punto è possibile procedere con l'estrazione dello scheletro, ovvero la trasformazione di distanza. L'immagine che si ricava viene trasformata in forma binaria, l'onda si propaga da celle poste sul contorno verso l'interno e ciò che ne deriva è la composizione dello scheletro da cui si dovrebbe dedurre la forma più o meno precisa dell'oggetto.

L'algoritmo che compone lo scheletro è composto da due passi:

1. Analisi della griglia, ovvero della matrice con una passata da sinistra a destra e dall'alto verso il basso;

2. Analisi della griglia da destra verso sinistra e dal basso all'alto:

    - Alle celle esterne al perimetro viene assegnato il valore zero;

    - Alle celle interne viene assegnato un valore elevato;

    - Per ipotesi ogni cella ha otto celle confinanti.

Nel primo passaggio 1. , a ogni cella che non vale zero, si assegna un valore maggiore del più piccolo valore dei quattro vicini, che sono i tre inferiori e il quadretto a sinistra, incrementato di uno.

Nel secondo passaggio 2. si ripete l'operazione con i restanti quattro quadretti.

Al termine delle due passate, il risultato è una matrice che contiene in ogni cella un valore indicante il minor numero di passi necessari per raggiungere il perimetro di quel punto.

Volendo applicare questo algoritmo alla robotica, ritornando alla propagazione dell'onda di cui abbiamo parlato in precedenza, teniamo conto che da una cella di goal, la propagazione può avvenire in tutto lo spazio libero, ovvero fluire attorno agli ostacoli, con le seguenti informazioni:

- Ogni cella contiene la distanza dal goal;
- Le celle in cui si trovano gli ostacoli sono poste a un valore infinito;
- I due passi verranno ripetuti finché non ci saranno più cambiamenti di valore nelle celle.

Le uniche celle della griglia con valore zero sono quelle di goal; esisterà un solo minimo globale in tutto lo spazio libero.

Al robot, da qualsiasi cella si trovi, sarà sufficiente percorrere le celle con valori decrescenti fino a raggiungere il goal.

Una possibile struttura per l'algoritmo è data da:

- Matrice m*n, n righe, m colonne;
- Goal[x,y]: è vero se cell[x,y] è un punto di goal;
- Start[x,y]: è vero se cell[x,y] è un punto di start;

- Blocked[x,y]: è vero se cell[x,y] è un parzialmente o totalmente occupato.

Agli elementi di cell si assegna valore il minimo valore degli otto vicini incrementato della distanza: il numero delle iterazioni dipenderà dalla concavità o meno degli ostacoli.

Il cammino da start a goal si ottiene percorrendo le celle per valore decrescente. Non sempre questo cammino può essere trovato: può non esistere cammino da start a goal.

Il calcolo del cammino:

- Posiziona il robot nel pixel di partenza;
- Inizia la scansione dei pixel vicini: verifica se attorno esistono pixel che hanno un valore minore;
- Se non esistono: fallimento;
- Se esistono: aggiungi il pixel di partenza alla lista e ripeti i passi dell'algoritmo, posizionandoti nel pixel con valore minimo, finché non sei arrivato al pixel obiettivo;
- L'algoritmo deve restituire gli identificatori dei pixel, che sono stati messi nella lista dei pixel buoni.

5.6.6 Robot on the Edge: Attenti ai Bordi

C'è però un rischio anche in questa strategia. I minimi locali sono stati evitati, ma non per questo passiamo lontano dai bordi degli ostacoli e dalle pareti. Poiché in base alle nostre ipotesi conosciamo la posizione e la forma

degli oggetti presenti nella matrice possiamo elaborare una strategia che ci permetta di stare il più lontano possibile dai bordi degli ostacoli.

Invece di propagare un solo fronte d'onda dal pixel obiettivo, propaghiamo fronti d'onda da ogni pixel che si trova in prossimità dei bordi. Ognuno di questi pixel che si trova sui bordi riceve un nome univoco, e lo comunica a ogni vicino. Quando due nomi che sono sufficientemente distanti collidono allora abbiamo trovato il centro tra due ostacoli.

Il risultato è una sorta di diagramma di Voronoi tra gli ostacoli nella mappa, che può essere usato come una sorta di mappa per navigare nella griglia.

5.6.7 Conclusioni

I campi potenziali sono utili quando si conosce a priori lo spazio in cui può muoversi il robot, e quando è possibili ricondurre tale spazio in una griglia regolare, rettangolare o quadrata, suddivisibile in quadrati o pixel, ovvero suddivisibile in modo discreto.

Solo in tale modo garantiamo un'efficace esplorazione dello spazio da parte del robot.

Osserviamo che specie negli ultimi due casi trattati caso trattato, ovvero l'uso della strategia wavefront planner, specie con il metodo di evitare i bordi, è possibile un collegamento con il metodo di Voronoi, che opera su spazi non definiti e che utilizza l'algoritmo di A* per fare una stima del cammino ottimo da seguire.

5.7 I Diagrammi di Voronoi

5.7.1 Introduzione

Un algoritmo alternativo al metodo dei campi potenziali e l'algoitmo di Voronoi. L'**algoritmo di Voronoi** ricerca il luogo dei punti più vicino al punto di stop, ovvero al punto di arrivo del robot.

Tale regione è il cosidetto **diagramma di Voronoi**, che non rappresenta una mappatura degli ostacoli. Per avere un cammino possibile bisogna avere un numero sufficientemente elevato di punti di stop. I cammini possibili sono ottenuti passando attraverso le regioni di Voronoi che sono etichettate come libere.

Il duale dello spazio di Voronoi è la cosiddetta **triangolazione di Delaunay**.

Consideriamo che i lati delle regioni di Voronoi sono le zone più distanti dagli ostacoli. Connettiamo i punti di stop, i cui segmenti tagliano le facce delle rispettive regioni di Voronoi. In modo più semplice, la triangolazione di Delaunay si può ottenere dal diagramma di Voronoi, prendendo come punti di stop i punti al centro delle regioni di Voronoi.

A* può essere applicato alle regioni di Voronoi di spazio libero [4], che sono rappresentate in un grafo. L'algoritmo calcola la sequenza di punti generatori di regioni di Voronoi da attraversare. A* è solo uno dei metodi che permettono di calcolare il passaggio tra regioni contigue.

Questo algoritmo può essere applicato ancora ad altri tipi di rappresentazioni spaziali per il movimento del robot.

Vediamo in pratica come si può sfruttare questo algoritmo nella navigazione robotica, anche in fase di pulitura e raffinamento delle mappe generate da altri algoritmi.

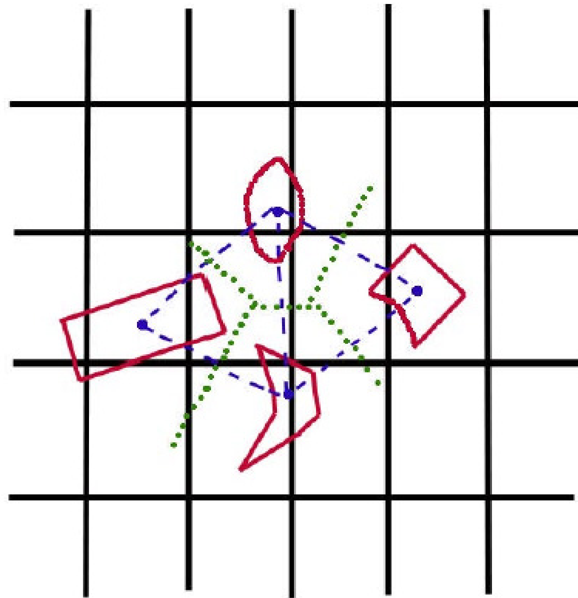

Figura 5.1: Diagrammi di Voronoi: Ostacoli

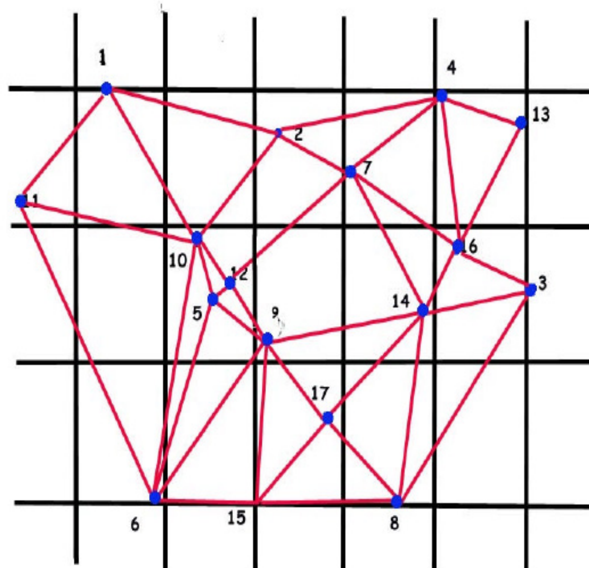

Figura 5.2: Triangolazione di Delaunay

5.7.2 Sketch Map

Voronoi permette di costruire per punti una mappa semplificata, ovvero uno sketch dell'ambiente in cui si muove il robot.

Pensiamo a una mappa dal punto umano.

Questo tipo di mappa mostra solo le parti più importanti, da un punto di vista umano, di quell'ambiente. Una generica mappa conterrà:

- **Muri**: in genere rappresentati da pixel neri, che non possono essere attraversati dal robot;
- **Corridoi**: gli spazi liberi tra i muri, in cui il robot può muoversi;
- **Stanze**: pensiamo le come dei poligoni con lati i muri;
- **Porte**: rappresentate in un colore differente dai muri: il robot può attraversarle.

Questo genere di mappa può andare bene per un uomo, perché già conosce queste informazioni. Un robot invece si muove in ambienti sconosciuti. In particolare:

- La scala non è nota: è uno svantaggio perché non permette al robot di conoscere e valutare le distane tra sé e gli ostacoli;
- La scala non è uniforme: le varie parti della mappa non sono rappresentate in modo omogeneo;
- Il mondo reale non corrisponde esattamente a quello della mappa: essa, in quanto tale, rappresenta solo le informazioni strettamente necessarie;
- La mappa può non avere informazioni sugli ostacoli.

Per questo motivo è utile utilizzare Voronoi.

5.7.3 Algoritmo di Voronoi

L'algoritmo può essere così strutturato:

- **Campionamento dei Pixel**: la mappa data è suddivisa in pixel. Si considerano piccoli quadrati di pixel, ad esempio quadrati di lato 5*5 pixel, e si fa una media dei pixel occupati, mettendo alla fine un solo pixel che rappresenta tutti quelli occupati in quel quadrato;
- **Spazio Duale**: è la triangolazione di Delaunay. Possiamo creare il diagramma di Voronoi con questa tecnica, che prevede di utilizzare un triangolo, che pensiamo come un insieme di tre punti (vertici). Il pixel che segniamo è quello che si trova all'interno del cerchio per i tre punti del triangolo, unico punto tra quei tre punti. Questo metodo è stato usato per misurare la Francia ai tempi del Re Sole;
- **Stanze**: Voronoi deve fare attenzione agli ostacoli che possono essere presenti nella stanza;
- 4. **Il Diagramma**: bisogna unire i centri delle circonferenze per trovare proprio i cammini liberi che possono essere percorsi dal robot.

Ma non è finita qui. Una volta ottenuti i cammini di Voronoi bisogna ottenere una mappa topologica.

5.7.4 Mappa Topologica

Una mappa topologica è un grafo, dove i nodi sono sono i posti e gli archi le adiacenze. Questo tipo di mappa può essere ottenuto con un ulteriore semplificazione della mappa di Voronoi.

In questa mappa distinguiamo:

- **Branch Points**: letteralmente i punti ramificazione, ovvero i punti con più di due vicini;

- **Dead Ends**: i vicoli ciechi, ovvero quei nodi che terminano contro i muri;

- **Doorways**: nodi in cui almeno uno dei vicini è un pixel differente si dal nero dei muri che dal bianco degli spazi vuoti, ovvero terminano in una porta.

Una coppia di nodi è collegata da un arco se c'è un cammino diretto tra quei nodi, ma, ricordando che le mappe non sono accurate, non è sempre garantita questa corrispondenza.

Un altro errore possibile è quello del rovesciamento, causato dall'uomo che ha disegnato la mappa, della destra con la sinistra, che può generare confusione nel robot.

Per una maggiore classificazione delle aree e dei percorsi è possibile usare colori specifici, che evidenzino i diversi tipi di informazioni. Per questo i diagrammi di Voronoi usano dei classificatori che si applicano sia alla generazione dei percorsi sia alle informazioni nelle aree della mappa.

Capitolo 6

SLAM

6.1 SLAM: Introduzione

SLAM è un acronimo inglese che sta per "s*imultaneous localization and map-building*".

Lo Slam tiene conto di due problemi chiave per un robot:

- **Localizzazione** - Dove sono?
- **Mapping** - Com'è l'ambiente in cui mi trovo?

Il suo scopo principale è quello di far rendere conto il robot dell'ambiente in cui si trova ad operare. Uno dei task principali è infatti la localizzazione, ovvero il capire la sua propria posizione nell'ambiente in cui si trova. Se il robot si trova in assenza di vincoli precisi, punti di riferimento fissi, deve essere in grado di costruirsi una mappa da solo. Questo è lo scopo e il compito dello SLAM [8].

Disegnare una mappa è spesso considerato uno dei task a sé stanti nell'operato del robot. Non a caso si parla di *exploration strategy*, strategia di

esplorazione. Il robot deve capire dove andare per disegnare una mappa la più precisa possibile. In generale, questo problema è l'intersezione di:

1. Guadagno di informazioni;
2. Costo di navigazione;
3. Qualità della localizzazione.

La soluzione al problema va ricercata all'interno delle aree sovrapponibili di Mapping, Localization, Motion Control.

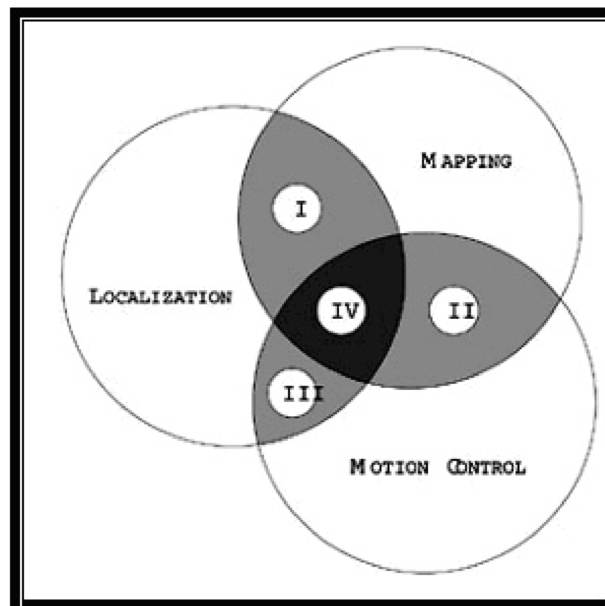

Figura 6.1: SLAM: Regioni di interesse

Nella figura sono evidenziate tre aree di sovrapposizione parziale e una di sovrapposizione totale tra le tre aree di ricerca.

La regione I rappresenta la intersezione tra il Mapping e la Localization, con l'uso di molti algoritmi di SLAM.

La regione II rappresenta l'intersezione tra il Mapping e il Motion Control, con la scelta di ogni strategia di esplorazione.

La regione III rappresenta l'intersezione tra Localization e Motion Contro, con la scelta di informazioni dai sensori e dalla navigazione attiva.

6.2 SLAM: Descrizione

Supponiamo di avere un robot mobile, dotato di sensori, in grado di effettuare delle rilevazioni sui punti di riferimento vicini al robot stesso.

I punti di riferimento possono essere sia naturali che artificiali, e comunque rilevabili da parte dell'algoritmo che governa i sensori.

Il robot parte da una locazione sconosciuta, senza alcuna conoscienza dei punti di riferimento nell'ambiente in cui si trova. non appena il veicolo inizia a muoversi, inizia anche l'osservazione dei punti di riferimento.

Dobbiamo basarci su un modello matematico, basato su matrici, dove possiamo individuare i seguenti risultati: il determinante dei ogni sottomatrice delle matrice di covarianza della mappa decresce in modo monotono con il susseguirsi di osservazioni favorevoli dei punti di riferimento; il numero delle osservazioni, nel suo crescere è direttamente correlato al numero dei punti di riferimento; la covarianza di ogni punto di riferimento stimato è determinata solamente dalla covarianza iniziale stimata dalla posizione del veicolo.

Questi tre risultati teorici dimostrano che [8] [9]:

- L'algoritmo di SLAM dipende dal modo in cui viene gestita la cross correlazione tra i punti di riferimento stimati. Questo aspetto non può essere minimizzato o sottovalutato, perché comporta un allontanamento dalla soluzione del problema.

- Più il robot si addentra nell'ambiente, compiendo esplorazioni, più gli errori delle stime di ogni coppia di punti di riferimento diventano correlati, senza mai recedere da questa osservazione.

- A lungo termine, gli errori nelle stime di ogni coppia di punti di riferimento diventano correlati in pieno. In pratica, significa che, data la posizione esatta di ciascun punto di riferimento è possibile determinare, con assoluta certezza, la posizione di ogni altro punto di riferimento.

- Più il robot continua nell'osservazione di punti di riferimento, più l'errore delle stime della sua posizione, tra punti di riferimento differenti, tende a convergere in modo monotono al punto che la mappa delle posizioni relative può essere determinata con assoluta certezza.

- Così come la mappa converge, secondo i le modalità esposte in precedenza, l'errore nella posizione assoluta dei punti di riferimento raggiunge un *lower bound*, ovvero un valore minimo, dipendente soltanto dall'errore presente nell'osservazione iniziale.

In questo modo è possibile trovare una soluzione al problema dello SLAM, e quindi è possibile, allo stesso, disegnare una mappa accurata e stimare la posizione corrente del robot, senza una conoscenza a priori dei punti di riferimento.

6.3 SLAM: il Filtro di Kalman

Il Filtro di Kalman è un filtro ricorsivo, che fa la stima di un sistema dinamico da una serie di misurazioni incomplete e affette da rumore. Pensiamo ad esempio a velocità e posizione di un oggetto in movimento. Ognuno dei singoli stati di osservazione fornisce delle informazioni, tutte affette da errori e disturbi.

Volendo essere più formali, il filtro di Kalman è un'applicazione dell'algebra lineare e del modello nascosto di Markov. Il sistema dinamico è modellato sulle *catene di Markov*, con n operatori lineari affetti da rumore gaussiano. Lo stato del sistema è rappresentato da vettori di numeri reali. A ogni incremento discreto di tempo, un operatore lineare viene applicato allo stato per generarne uno nuovo, affetto da rumore. A sua volta viene applicato un altro operatore lineare a questo nuovo stato, anch'esso affetto da rumore, e così si ottiene lo stato nascosto del sistema.

Il modo per ottenere la stima di un processo interno bisogna modellare il sistema secondo una serie nota di matrici relative al modello del filtro di Kalman, F_k, H_k, Q_k, R_k, e qualche volta B_k, per ogni istante di tempo k.

Il filtro di Kalman assume che il vero sistema, all'istante k, si è evoluto dall'istante $(k-1)$, secondo la seguente espressione:

$$x_k = F_k x_{k-1} + B_k u_k + w_k$$

dove:

- F_k è il modello di stato di transizione che è applicato allo stato prece-

dente x_{k-1};

- B_k è il segnale input di controllo che è applicato al vettore di controllo u_k;
- w_k è il processo rumore che si assume derivante da una distribuzione normale multivariante a valor medio zero, con covarianza Q_k.

Se: w_k $N(0, Q_k)$, all'istante di tempo k, una osservazione, o misura, z_k, del vero stato x_k è compiuta con l'assunto che: $z_k = H_k x_k + v_k$, dove H_k è il modello di osservazione che mappa il vero spazio di stato nello spazio osservato e Q_k è il rumore osservato, supposto bianco, gaussiano a valor medio zero e con covarianza R_k.

Se: w_k $N(0, R_k)$, lo stato iniziale e i vettori del rumore $x_0, w_1, ..., w_k, v_1 ... v_k$ sono supposti essere mutuamente indipendenti.

Per completezza, osserviamo che molti sistemi dinamici si rifanno a modelli più completi e complessi di questo.

6.3.1 Modelo Matematico del Filtro

Dopo aver illustrato il modello del sistema di Kalman, passiamo alla trattazione del Filtro di Kalman vero e proprio.

Il Filtro di Kalman è uno stimatore basato sul principio della ricorsione [10] [8] [9]. Questo significa che soltanto lo stato stimato all'istante precedente e la misurazione corrente sono necessarie all'algoritmo per stimare lo stato corrente. Dunque non è necessaria alcuna storia passata del sistema per fornire una stima del sistema stesso all'istante temporale presente.

Lo stato del filtro è rappresentato da due variabili:

- $x_{k|k}$: Stato Stimato all'istante di tempo k;
- $P_{k|k}$: Matrice di Covarianza dell'Errore, (una misura dell'accuratezza della stima dello stato).

Il filtro ha due diverse fasi: **Predizione** e **Aggiornamento**.

La *Predizione* usa la stima dal precedente istante temporale, per produrre una stima dello stato corrente. L'*Aggiornamento* si basa sulle informazioni delle misure all'istante corrente, per raffinare la sua predizione e arrivare a una nuova stima, possibilmente più accurata.

Predizione:

- $\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k$: Stato del Predittore;
- $P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k$: Covarianza Stimata del Predittore.

Aggiornamento:

- $\tilde{y}_k = z_k - H_k \hat{x}_{k|k-1}$: Innovazione o Misura Residua;
- $S_k = H_k P_{k|k-1} H_k^T + R_k$: Covarianza Residua;
- $K_k = P_{k|k-1} H_k^T S_k^{-1}$: Guadagno di Kalman;
- $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k$: Aggiornamento dello Stato Stimato;
- $P_{k|k} = (I - K_k H_k) P_{k|k-1}$: Aggiornamento della Covarianza Stimata.

6.3.2 Il Filtro di Kalman Esteso

Il **filtro di Kalman Esteso**, in inglese **EKF**, è un filtro di Kalman non lineare. Questa variante si basa su funzioni differenziabili che rappresentano lo stato di transizione e il modello di osservazione:

- $x_k = f(x_{k-1}, u_k, w_k)$
- $z_k = h(z_k, v_k)$

La funzione f può essere usata per calcolare lo stato del predittore dal precedente stato stimato e, allo stesso modo, la funzione h può essere usata per calcolare la misura predetta dallo stato del predittore. Tuttavia f e h non possono essere applicate alla covarianza, in modo diretto: per risolvere questo inconveniente si calcola una matrice di derivate parziali, lo Jacobiano.

A ogni istante di tempo lo Jacobiano è stimato dagli stati predetti correnti. Tali matrici possono essere usate nelle equazioni del filtro di Kalman. Tutto questo procedimento non è altro che una linearizzazione di funzioni non lineari attorno alla stima corrente. Il risultato sono le equazioni dell'EKF.

Predizione:

- $\hat{x}_{k|k-1} = f(x_{k-1}, u_k, 0)$
- $P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k$

Aggiornamento tramite Jacobiano:

- $F_k = \frac{\partial f}{\partial x}|_{\hat{x}_{k-1|k-1}, u_k}$

- $H_k = \frac{\partial h}{\partial x}|_{\hat{x}_{k|k-1}}$

Aggiornamento:

- $\tilde{y}_k = z_k + h(x_k, 0)$
- $S_k = H_k P_{k|k-1} H_k^T + R_k$
- $K_k = P_{k|k-1} H_k^T S_k^{-1}$
- $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k$
- $P_{k|k} = (I - K_k H_k) P_{k|k-1}$

Immaginando di avere un semplice robot che ha montata una telecamera su se stesso. Per ottenere un'immagine binaria si applica una soglia di luminosità all'immagine ottenuta tramite la telecamera. In questo modo eliminiamo gli effetti di riflessione che porterebbero a una minore chiarezza del risultato.

Dopo l'applicazione della soglia di luminosità all'immagine è applicata la segmentazione. Questo processo ridice l'immagine alle sue componenti costitutive, definendole in un grafo e trova le relazioni che esistono tra essi, i cosiddetti "blob". Notiamo che a causa della non perfetta nitidezza dell'immagine ottenuta, i contorni possono formare degli oggetti inesistenti, e compromettere il risultato finale dell'osservazione.

Il terzo passo è ottenere il perfetto posizionamento dell'immagine, ottenuta con l'algoritmo di posizionamento del centro di massa. In questa fase vengono anche addolciti i contorni dell'immagine e dei suoi oggetti.

Un ulteriore raffinamento del filtro di Kalman è rappresentato dall'IEKF.

IEKF è un acronimo inglese per **Interlaced Extended Kalman Filter** [9]. L'idea di base per questo filtro deriva dalla teoria dei giochi. In un ambito di stima i giocatori rappresentano l'algoritmo di stima, la strategia è la stima e la funzione obiettivo è la misura della covarianza dell'errore di stima.

In pratica IEKF è una applicazione in parallelo di m filtri di Kalman. Ogni filtro lavora in modo indipendente dall'altro e quello che fa è stimare solo un sottoinsieme delle variabili dello spazio di stato, considerando tutto il resto come parametri deterministici varianti nel tempo. L'errore che viene introdotto è parzialmente attenuato incrementando le matrici di covarianza del rumore.

Capitolo 7

Architettura Doro

7.1 Introduzione

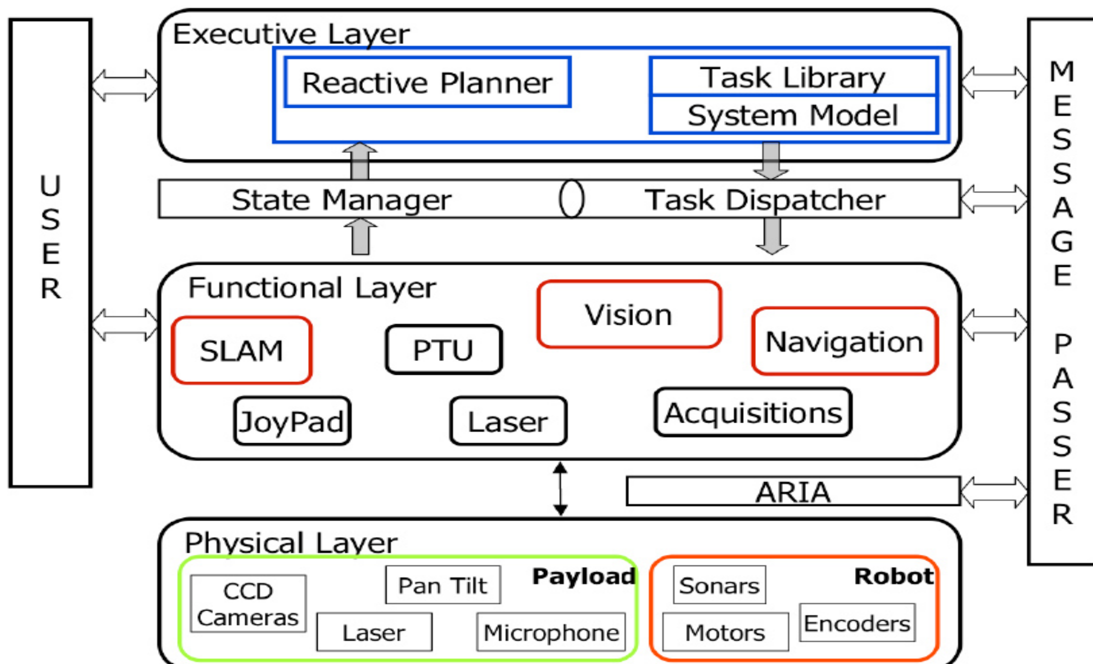

Figura 7.1: Architettura DO.RO. - Domestic Robot

DO.RO. - Domestic Robot - è un robot che è stato progettato per la ricerca e per il salvataggio delle persone in ambienti post-terremoto. In questo senso, DO.RO. svolge le proprie missioni impegnandosi nell'esplorazione di ambienti, al fine di trovare le vittime. L'operatore segue le attività del Robot valutando lo stato interno del sistema e considerando le informazioni percepite da quest'ultimo durante l'esplorazione. Pertanto, l'interazione tra l'operatore ed il sistema di controllo che regolarizza le diverse attività durante le missioni, è senza dubbio, uno degli aspetti più importanti che caratterizzano il comportamento di DO.RO. e l'esito della missione stessa [7].

Tra le necessità principali da soddisfare ci sono il **controllo dinamico dei cambiamenti** e l'**allocazione dinamica delle risorse**.

Questa esigenza è legata al fatto che spesso ci troviamo davanti ad un dominio incerto in termini di tempo e risorse.

Per questo scopo è stato introdotto un sistema di controllo **model-based**, che supervisiona ed integra le attività autonome del robot e gli interventi dell'operatore: seguendo questo approccio, sia i processi dell'operatore, che quelli del sistema autonomo, sono rappresentati esplicitamente attraverso un modello temporale dichiarativo il quale garantisce un'interpretazione globale del contesto esecutivo.

Un pianificatore reattivo può monitorare lo stato del sistema e generare il controllo svolgendo ciclicamente le attività: **sensing-planning-execution**.

Ad ogni ciclo il pianificatore deve generare le attività fino ad un orizzonte temporale fissato e monitorare la consistenza delle operazioni concorrenti controllando i guasti. Le attività di pianificazione a breve termine bilanciano la reattività ed il comportamento orientato all'obiettivo da raggiungere; in tutto questo, anche l'operatore può intervenire influenzando l'attività di pianificazione in una modalità di funzionamento ibrida.

7.2 Livello Esecutivo

La descrizione dell'architettura di controllo di DO.RO. si concentra fondamentalmente nella descrizione di tre livelli, ognuno dei quali lavora ad un livello di astrazione differente:

- Il livello fisico è costituito dal robot e da tutti i dispositivi che vengono controllati, nel livello funzionale, da un'insieme di moduli: il modulo mapping gestisce la costruzione dinamica della mappa e la localizzazione delle vittime; il modulo di navigation gestisce i movimenti del robot mentre il modulo di vision elabora le immagini per identificare le vittime, comunicandone la posizione al modulo mapping [7].

- Il livello esecutivo ha la responsabilità di generare un piano e controllarne l'esecuzione.

- La comunicazione tra il livello funzionale e quello esecutivo è regolata da un livello intermedio, costituito dal task dispatcher e dallo state manager.

Il sistema misura il tempo come una sequenza di tick: il periodo che intercorre tra una tick ed il successivo è quello necessario per l'esecuzione di un ciclo: **sensing-planning-execution**. Durante un tick, il sistema riceve informazioni riguardo lo stato di ciascun modulo funzionale dallo state manager (*sensing*), costruisce un piano considerando lo stato del sistema e le attività svolte nella missione fino ad allora (*planning*), e invia il piano al task dispatcher (*execution*).

Lo state manager si aggiorna riguardo lo stato di ogni modulo ogni 200 millisecondi. Il task dispatcher invia il segnale di attivazione di un task ai

moduli funzionali dopo averne ricevuto richiesta dal pianificatore. Il pianificatore, nel livello esecutivo, apprende le informazioni dallo state manager e sulla base di quest'ultime costruisce un piano d'azione.

Tale piano viene inviato quindi al task dispatcher il quale attiverà i vari segnali indirizzandoli ai vari moduli funzionali, affinché questi ultimi possano avviare il task opportuno e modificare così il proprio stato interno.

Al ciclo successivo il pianificatore controllerà le informazioni riguardanti lo stato di ogni modulo funzionale, tramite lo state manager. Grazie ai dati ricavati, il pianificatore sarà così in grado di capire se l'operazione è avvenuta con successo; in questo caso si procederà con la pianificazione. Nel caso in cui l'azione non sia avvenuta correttamente: verrà selezionata una procedura di recupero.

Anche il pianificatore, durante la sua attività, può fallire: in questo caso può essere selezionata una procedura che raggiunga velocemente una modalità sicura oppure che selezioni un nuovo task. Tutta questa struttura inoltre accoglie anche le richieste dell'operatore: il sistema di pianificazione cerca di collegare le attività scelte dall'operatore, con l'obiettivo della missione e con le attività dei moduli funzionali.

Il sistema è modellato come un insieme di componenti (*slam*, *nav*, *vis*, *ptu*) il cui stato cambia con il tempo.

Questi componenti modellano i moduli funzionali situati nel livello funzionale: *map* è associato al modulo mapping, *nav* è associato al modulo navigation, *vis* è associato al modulo vision e *ptu* è associato al modulo pan-tilt.

Ad ogni componente sono associate le azioni che possono essere compiute dal modulo corrispondente ed inoltre, ognuno di essi descrive la sua storia attraverso una sequenza di stati e di attività. In questo modo il sistema di

controllo descrive nel dettaglio tutte le azioni eseguibile dal sistema, le relazioni di causa ed effetto e le proprietà causali e temporali del sistema stesso; contiene inoltre tutta la storia vissuta dal sistema dall'inizio della missione. Tutte queste caratteristiche permettono, un comportamento flessibile e reattivo del sistema.

La **flessibilità** e la **reattività** sono caratteristiche importanti; il fatto che il robot venga a trovarsi all'interno di un dominio dinamico, presuppone una capacità di adattamento ai cambiamenti della realtà di interesse che lo circonda. Parlando della reattività, non possiamo non considerare che quest'ultima è stata ottenuta soprattutto grazie ad un pianificatore sviluppato sulla base del concurrent temporal situation calculus: il pianificatore è dotato di una libreria di TCGolog script che rappresenta una serie di frammenti di comportamenti (*task*). Ad ogni ciclo di esecuzione, pertanto, dopo che lo stato del robot è stato aggiornato, l'interprete Golog è chiamato ad estendere la sequenza di controllo scegliendo nuovi task dalla libreria, quando alcuni finiscono o falliscono.

7.3 Livello Funzionale

Il livello funzionale è costituito da 4 moduli: *navigation*, *mapping*, *vision* e *pan-tilt*. Ognuno di questi moduli contiene un insieme differente di azioni e ogni azioni può essere attivata o disattivata in relazione alle azioni di start e end comunicate dal task dispatcher. Un modulo funzionale quindi è un componente reattivo che cambia il suo stato interno rispetto alle informazioni ricevute dal task dispatcher.

Tuttavia un modulo può anche suggerire al pianificatore un azione da

eseguire in futuro: il modulo di mapping può assumere uno stato particolare per comunicare che il ciclo di generazione della mappa è terminato e che il livello superiore può scegliere un'azione per fermarne l'attività.

Nel livello funzionale inoltre esiste una costante comunicazione fra i vari moduli: ad esempio il modulo di navigation riceve dal modulo di mapping le coordinate dei punti necessari per decidere il percorso e per ottimizzare gli spostamenti.

Il **modulo navigation** gestisce sostanzialmente gli spostamento del robot e la sua posizione all'interno della mappa; tale modulo può trovarsi in tre macro-stati:

- **Stop**: il modulo non è attivo e quindi il robot è fermo;
- **Wander**: in questa fase deve essere attivo anche il modulo di mapping che genera la mappa e calcola una serie di zone da esplorare; il modulo di navigation provvede agli spostamenti del robot verso queste aree;
- **Goto**: il robot cerca di raggiungere un punto esatto che è stato calcolato dal modulo mapping.

Il **modulo mapping** è il responsabile della strategia di esplorazione: ha il compito di ottimizzare gli spostamenti e di calcolare i punti da cui effettuare i processi di visione.

Può trovarsi in tre stati:

- **Stop**: è inattivo;
- **Run**: calcola la mappa;
- **Scan**: effettua la scansione laser per aumentare l'accuratezza della mappa.

Vengono utilizzate due mappe: una *globale* e l'altra *locale*. Quella locale viene integrata in maniera incrementale in quella globale: questo è utile per vedere se ci fossero aree ancora inesplorate.

L'integrazione è ovviamente preceduta dall'allineamento del robot e dal controllo degli errori. Inoltre viene lasciata traccia della storia del percorso fatto dall'inizio della missione e di quello fatto nell'ultimo ciclo. Tali percorsi sono utili quando è necessario trarre conclusioni riguardo le zone già visitate in relazioni ad altri percorsi alternativi da fare per raggiungerle.

L'esplorazione finisce in base all'esito di un test che misura il livello di apprendimento raggiunto nell'ultima fase di esplorazione; tale misura viene calcolata per mezzo di una funzione che considera: il tempo percorso dall'inizio della missione, la distanza percorsa in quel periodo e quella percorsa nell'ultimo ciclo.

Il **modulo vision** gestisce le telecamere, le quali vengono attivate nel momento in cui viene raggiunta una posizione interessante. Può trovarsi in due stati:

- **Idle**: il modulo è inattivo;
- **Visprocess**: il modulo acquisisce una serie di foto, le immagini vengono segmentate e analizzate alla ricerca di colori che potrebbero appartenere alla pelle di una vittima.

Tale processo di riconoscimento è basato sulla costruzione di una skin map e su un processo di adattamento per far fronte alle diverse condizioni di illuminazione. Una volta trovata la vittima, deve essere inserita nella *global map* usando le informazioni relative all'orientamento e alla posizione del ro-

bot, nonché all'angolo del *pan-tilt*.

Il **modulo pan-tilt** può essere inutilizzato o può intervenire per servire il modulo di mapping o quello di vision.

Può trovarsi in tre stati:

- ptuIdle: il modulo è inattivo;
- ptuMoving: il modulo è in movimento;
- ptuPoint: il modulo punta in una direzione.

Capitolo 8

Iniziativa Mista

8.1 Interazione Robot-Operatore

L'interazione tra il robot e l'operatore, è sicuramente uno tra gli aspetti fondamentali che caratterizzano l'esito di una missione. I vantaggi e le novità di questa collaborazione possono essere apprezzati considerando:

- **Interazione robot-operatore**: grazie al modello dichiarativo delle attività del robot, il sistema è informato riguardo la situazione corrente e del suo stato, a diversi livelli di astrazione. Le interazioni tra le varie attività vengono rilevate e comunicate all'operatore.

- **Interazione operatore-robot**: l'operatore può approfittare delle funzionalità di base del robot come la mappa, il rilevamento delle vittime e della sua posizione e la conoscenza dei punti ottimali di osservazione. La conoscenza di queste informazioni consentono all'utente di focalizzare la sua attenzione sullo stato corrente dell'esplorazione e interagendo con un *Mixed-Initiative Planner*, l'operatore può influenzare l'attività del robot.

Le attività del robot e dell'operatore sono continuamente coordinate attraverso un processo di controllo, che verifica che lo stato del sistema sia allineato con il modello dichiarativo del sistema stesso e che integra le iniziative concorrenti.

Quest'interazione risulta essere di notevole importanza: se l'operatore dovesse controllare completamente l'attività del robot, dovrebbe focalizzare la sua attenzione su un numero considerevole di fattori, rischiando così di perdere di vista l'obiettivo della missione cioè il ritrovamento delle vittime.

D'altra parte, un sistema di controllo totalmente autonomo non è ancora proponibile in un contesto che presenta troppe irregolarità e imprevisti e dove sono richieste numerose capacità al sistema robotico. Pertanto la soluzione ideale è quella di integrare l'attività autonoma con quella teleoperata: in tal senso il processo di pianificazione coordina, integra e monitora gli interventi e le decisioni dell'operatore all'interno del piano, considerando l'obiettivo della missione, i vincoli e i processi funzionali concorrenti.

La modalità teleoperata e quella autonoma introducono degli svantaggi significativi: la scelta che riteniamo migliore è quella ad iniziativa mista. In questo modo il paradigma di progettazione ad alto livello dell'agente associato all'attività del pianificatore, consente una generazione del piano incrementale.

L'operatore conosce completamente l'attività del robot ma la sua attenzione è concentrata esclusivamente sulle decisioni locali e sui vincoli generali del problema. È possibile definire alcune modalità operative ad iniziativa mista [7]:

- **Planning-based interaction**: il sistema di pianificazione genera sequenze di azioni e l'operatore segue queste sequenze apportando piccole modifiche. L'operatore può controllare manualmente le attività funzionali schedulate dal pianificatore. Per esempio potrebbe decidere di

sospendere automaticamente la navigazione prendendo pertanto il controllo della attività legate al movimento del robot al fine di visitare un luogo interessante o allontanarsi da un'ambiente pericoloso. In questo tipo di interazione, l'operatore avrà un'influenza minima sul piano del pianificatore enfatizzandone così la stabilità.

- **Cooperation-based interaction**: in questa modalità l'operatore modifica la sequenza di azioni prodotta dal pianificatore togliendone o inserendone qualcuna. Pertanto tale tipo di intervento può comportare un disallineamento tra quello che l'operatore si aspetta e quello che realmente avviene. Il problema viene risolto grazie alla presenza di alcune procedure che riallineano il sistema reale con quello di controllo.

  Questa modalità garantisce la massima flessibilità per l'operatore e per il pianificatore: potendo dialogare tra loro e lavorare insieme apportano un contributo notevole all'esito della missione. Il sistema autonomo si interessa della generazione di un piano consistente, controlla il piano proposto dall'operatore e gestisce eventuali violazione dei vincoli del sistema. L'operatore interviene modificando il piano proposto dal sistema al fine di renderlo più efficiente.

- **Operator-based interaction**: in questa modalità le attività del sistema sono completamente nelle mani dell'operatore. In questo scenario il pianificatore seguirà il piano proposto dall'utente e sarà responsabile del controllo della consistenza: il sistema avvertirà l'operatore in caso di eventuali problemi di sicurezza suggerendo eventuali procedure di recupero.

È importante notare che in ognuno di questi approcci il sistema controlla continuamente le attività che devono essere svolte: sia quelle introdotte dal

pianificatore, che quelle proposte dall'utente. In entrambi i casi, nel momento in cui si dovessero presentare dei problemi legati alla sicurezza o alla consistenza, il pianificatore si impegnerà nella pianificazione o suggerirà delle alternative all'operatore.

8.2 Livello Fisico

Il ciclo esecutivo di DO.RO. è implementato mediante due programmi: ECLiPSe e C++, i quali possono essere pensati come due threads impegnati in un singolo processo di esecuzione. Il processo C++ è l'applicazione che guida il ciclo esecutivo comunicando pertanto con il pianificatore, implementato in TCGolog, attraverso l'execution monitor, implementato quindi in ECLiPSe Prolog

Entrambi i programmi, sono caratterizzati da un proprio stato e contengono dei metodi per lo scambio dei dati e per cedere il controllo all'altro thread. Mentre il processo C++ è responsabile della conoscenza e della modifica dello stato di DO.RO, il pianificatore si impegna nella costruzione di un piano orientato al raggiungimento dell'obiettivo e influenzato dallo stato del sistema. Pertanto durante lo scambio delle informazioni, il processo C++ informa il pianificatore riguardo lo stato di DO.RO. mentre il pianificatore fornisce al processo C++ la lista di istruzioni che dovranno essere eseguite dal Robot.

8.3 Pianificatore

In linea con quanto avviene nel situation calculus, il quale esprime i cambiamenti del mondo nella logica del primo ordine, nel pianificatore sono definiti:

- **Le azioni primitive** che possono essere svolte dal robot. Per ogni azione è presente la clausola poss(azione,situazione) che definisce le precondizioni dell'azione.
- **I fluenti** che rappresentano gli stati che possono essere assunti da moduli funzionali. Per ogni assioma è presente l'assioma di stato successore.
- Per la **situazione iniziale** è definita una clausola che descrive lo stato del sistema nel momento dell'avvio dopo una fase di sensing. iniziale.

Inoltre ritroviamo anche:

- **I programmi** che generano la sequenza di controllo e monitorano il sistema.
- **Le procedure** che modellano i task del robot.

In ogni task sono presenti le azioni che devono essere inviate ai vari moduli funzionali(al massimo un'azione per ogni modulo) e che devono essere eseguite nello stesso ciclo esecutivo. Ogni modulo funzionale è caratterizzato da un proprio insieme di azioni e può assumere determinati stati.

Le procedure più importanti sono quelle che modellano i task di esplorazione (*wander*), scansione laser (*scan*), processamento immagini (*vision*) e raggiungimento dei un punto (*Goto*); le altre sono relative all'inizializzazione, al riallineamento del sistema e alla *recovery*. Ad ognuna di queste procedure,

è stata associata una clausola $guard(Goto, S)$ che definisce le condizioni per l'attivazione della procedura *Goto*.

Analizziamo, ad esempio, la procedura *Goto*:

$proc(Goto, pi(t1, pi(t2, pi(x, [navigation_go_start(x, t1)] :$
$[navigation_go_end(x, t2)] :?(t2 - t1\sharp < 100);$

la guard è del tipo:

$guard(Goto, S) : -percept_idle(T, S),$
$navigation_stop(_, wand_T1, S), navigation_mode(_, on, S).$

Il processo del pianificatore si basa sull'esecuzione di un ciclo in cui si distinguono tre fasi: *sensing*, *planning* ed *execution*.

- **Sensing**: in questa fase il pianificatore aggiorna lo stato del sistema. Questa informazione gli perviene dal processo C++ attraverso una variabile di stato.

- **Planning**: in questa fase viene allargato l'orizzonte di pianificazione attraverso la scelta di uno o più task e l'inserimento di quest'ultimo nel piano. La scelta dei task è tale che ogni azione deve avere le precondizioni conformi allo stato del sistema ed inoltre il task deve essere consistente con il contesto di esecuzione.

- **Execution**: in questa fase viene eseguito il piano scelto. Pertanto dal piano viene estrapolata la prima istruzione che deve essere eseguita. A questo punto, se si tratta di un'azione di $startX$, si procede con la sua

esecuzione. In caso contrario, se l'azione è del tipo $endX$, prima viene controllato che il modulo funzionale associato ad X abbia comunicato che X può terminare. Se tale modulo precedentemente avesse avuto un cambiamento di stato tale da permettere l'esecuzione di $endX$, l'azione risulterebbe essere controllabile e si procederebbe con la sua esecuzione, in caso contrario l'esecuzione sarebbe rimandata (*delay*). La durata delle azioni non è un valore constante ma è rappresentata attraverso la specifica del limite inferiore e superiore entro il quale deve essere eseguita: il sistema decide ogni volta la durata dell'azione.

Quindi la procedura *Goto* può essere attivata quando lo stato del modulo vision è impostato su *idle* e lo stato della navigazione è impostato su *stop*(per esempio a seguito dell'azione *navigation_wand_end*). A queste condizioni, al piano viene aggiunta l'azione *navigation_go_start* e pertanto lo stato del modulo navigation diventa *goTo*; a $t1$ viene assegnato il valore relativo al tick in cui viene iniziata l'azione. Il sistema, come è stato visto precedentemente, percepisce il tempo come una sequenza di tick di durata fissa. Il valore 100 indica che l'azione *navigation_go_end* deve essere eseguita entro 100 tick dal momento in cui è stata eseguita *navigation_go_start*. Nel caso in cui dopo 100 tick, a partire da $t1$, non fosse ancora stata attivata l'azione *navigation_go_end*, il pianificatore procederà attivando una procedura di recupero. In caso contrario, verrà eseguita l'azione *navigation_go_end* che riporterà lo stato del modulo *navigation* a *stop*.

Quando al ciclo successivo, il pianificatore riceve l'informazione riguardante lo stato del sistema, riceve in realtà anche l'informazione riguardante l'esito dell'esecuzione dell'azione: nel caso in cui l'azione inviata precedentemente avesse terminato la sua esecuzione, il ciclo del pianificatore procederebbe con le fasi *sensing-planning-execution*, in caso contrario si limiterebbe alle fasi

sensing-execution.

8.4 Execution Monitor

L'execution monitor è formato da una serie di programmi che si interessano della gestione della comunicazione del pianificatore con il resto del sistema. Traduce le informazioni provenienti dallo state monitor in clausole comprensibile al pianificatore, ed il piano calcolato dal pianificatore in una struttura comprensibile dal processo C++. Inoltre l'execution monitor è in grado di effettuare ulteriori controlli sullo stato del robot e sulle fasi di esecuzione delle azioni.

8.5 Processo C++

Il processo C++ è responsabile dell'attivazione dei vari moduli funzionali; ricevuto il piano dal pianificatore, richiede al task dispatcher di inviare i segnali di attivazione ai moduli funzionali. Inoltre attraverso lo state manager viene a conoscenza dello stato interno del sistema e invia queste informazioni, insieme ad altre riguardanti le varie attività eseguite, al pianificatore. Il processo C++ è il primo programma che entra in azione quando viene avviato il sistema: richiede l'esecuzione di un programma prolog e gli invia lo stato del sistema affinchè quest'ultimo possa immediatamente procedere con la pianificazione. Durante l'esecuzione di ogni ciclo di esecuzione, il processo C++ e quello Prolog, presentano un solo punto di contatto in cui si scambino le informazioni descritte precedentemente. Anche il processo del C++ si basa su tre fasi:

- Lettura dallo state manager: il processo C++ ricava lo stato del sistema leggendolo dallo state manager;
- Ricezione piano: il processo C++ crea la struttura Status da inviare al prolog e a seguito della comunicazione con quest'ultimo, viene a conoscenza delle prossime azioni che devono essere eseguite dai vari moduli funzionali;
- Invio dei messaggi ai moduli funzionali.

8.6 Iniziativa Mista: Introduzione

La prima iniziativa mista introdotta nel sistema è stata quella finalizzata alla modifica del piano dovuta all'inserimento di un task in coda al piano stesso o alla cancellazione del task in corso.

L'operatore seguendo le attività svolte dal sistema potrebbe decidere di intervenire forzando l'esecuzione immediata di un task oppure attivando una procedura di recupero nel caso in cui venisse rilevato un comportamento anomalo.

Quando l'operatore inserisce un task, il pianificatore abbandona il piano attuale e ne costruisce uno nuovo.

Un task è costituito da un insieme di azioni; la caratteristica fondamentale di un task è quella di contenere al massimo un'azione per ogni unità funzionale.

Quindi il pianificatore in ogni ciclo esegue un task e pianifica il task, o i task, da eseguire successivamente. L'operatore, venuto a conoscenza del piano formulato dal sistema, potrebbe decidere di inserire un'azione all'interno di un task già presente nel piano oppure potrebbe voler inserire un'azione da eseguire in un tick diverso da quello dei task già presenti; quest'ultimo

caso coincide con l'inserimento di un task prima, dopo oppure tra i task già presenti.

Tuttavia nel momento in cui l'operatore decidesse di inserire un'azione nel piano, o decidesse di inserire un task, il sistema procederebbe con una serie di verifiche al fine di accertare che il piano finale risulti consistente.

Affinché un'azione possa essere eseguita, devono sussistere infatti le pre-condizioni necessarie all'attivazione dell'azione stessa ed inoltre devono essere soddisfatti ulteriori vincoli dettati dal sistema, che discuteremo in seguito. Nel caso in cui ci fossero dei vincoli o delle pre-condizioni insoddisfatte, il sistema procederebbe con l'esecuzione del piano da lui inizialmente formulato.

Quando l'operatore decide di intervenire sul piano, può farlo nei seguenti modi:

- Modificando le azioni presenti nel task proposta dal pianificatore;
- Cancellando un task proposta dal pianificatore ed inserendone un altro: questo attraverso la cancellazione di tutte le azioni presenti e l'inserimento di una nuova azione.
- Inserendo un task e decidendo se eseguire lo stesso prima o dopo quelli proposti dal pianificatore.
- Non intervenire sul piano proposto

8.7 Gestione del Futuro

Il fatto che l'operatore ha la possibilità di inserire un numero indefinito di task, ha comportato la necessità di ampliare l'orizzonte temporale pianificabile. L'aspetto riguardante l'estensione del futuro è stato uno dei fattori

largamente trattati durante l'introduzione dell'iniziativa mista: ha comportato infatti, l'estensione del sistema di controllo attraverso la revisione e l'inserimento di procedure più specifiche, mirate alla verifica della fattibilità del piano proposto.

La prima problematica affrontata durante l'attività di tirocinio, è stata quella riguardante la scelta di un canale di comunicazione tra l'operatore ed il livello esecutivo: l'operatore esegue l'intervento attraverso l'interfaccia. Il suo intervento causa l'aggiunta di un'ulteriore variabile alla struttura *Status* che viene inviato all'execution monitor all'inizio di ogni ciclo.

Anche l'interfaccia è stata arricchita con una struttura, che discuteremo in seguito, in grado di memorizzare il piano durante l'intervento dell'operatore. Il ruolo principale di quest'ultima è quello di mostrare il piano, che il pianificatore intenderebbe eseguire, all'operatore: tale piano conterrà il task che deve essere eseguito nel tick presente e quelli che verranno eseguiti successivamente.

Le azioni che sono contenute nel primo task, serviranno per l'attivazione dei messaggi da inviare ai moduli funzionali, al fine di eseguire il task stesso. I successivi task invece verranno opportunamente memorizzate e verranno mostrata all'operatore. Quest'ultimo, venuto a conoscenza del piano proposto dal pianificatore, potrebbe procedere apportando le modifiche per lui più opportune, ovviamente sempre attraverso l'interfaccia.

Le scelte che sono state fatte durante l'attività di progettazione sono state tutte mirate a garantire all'operatore la massima libertà di azione: l'operatore è libero infatti di inserire o cancellare, attraverso l'interfaccia,un numero non arbitrario di task o di azioni poichè si trova di fronte ad un orizzonte temporale esteso.

Proprio per queste ragioni, la struttura utilizzata nell'interfaccia per la memorizzazione del piano, sottoposto all'intervento dell'operatore, è una lista di liste:

- I task, all'interno del piano, vengono memorizzate in una lista in modo tale che non sia prestabilito il numero massimo di task che costituiscono il piano;
- Le azioni, all'interno di un task, vengono memorizzate in una lista.

Considerando che al massimo un task può contenere un numero di azioni pari a quello dei moduli funzionali presenti nel sistema, come visto precedentemente, la gestione delle azioni sarebbe potuta avvenire attraverso l'utilizzo di un array. Tuttavia non bisogna dimenticare che il numero di azioni all'interno di un task non è fisso, e pertanto si è optato, anche in questo caso, per una lista. Un altro fattore che sostiene tale scelta è quello riguardante l'evoluzione futura del sistema: l'inserimento di ulteriori moduli funzionali, non comporterà un eccessivo lavoro di aggiornamento sul sistema stesso.

A seguito di ogni modifica apportata sul piano, l'interfaccia presenterà il piano corrente all'operatore. Solo quando quest'ultimo avrà terminato la sua attività, le azioni che si trovano all'interno della lista di liste verranno gestite in modo da poter essere inviate al sistema di controllo: tutte le task verranno inserite in una lista con una struttura consona a quello che si aspetta il pianificatore. A questo punto quest'ultimo, procederà controllando le informazioni ricevute.

8.8 Verifica del Piano

Quando il pianificatore riceve il piano proposto dall'operatore, la prima cosa che deve fare è verificarne la consistenza. È stata introdotta una procedura in grado di valutare il piano inserito dall'operatore:

$choosePlan(P, S1, A, A1, OP)$

Tale procedura restituisce comunque un piano del quale è garantita l'eseguibilità: nel caso in cui il piano ricevuto dall'interfaccia non risultasse corretto dal punto di vista logico e dal punto di vista dei vincoli del sistema, verrà restituito il piano dettato dal pianificatore, altrimenti il piano restituito sarà proprio quello scelto dall'operatore.

$choosePlan(P, S1, A, A1, OP) : -$

$(P = [], A = [A11|A22], A22 = [T], OP = A;$

$(P = [], A = [A11|A22], OP = [A];$

$(checkfuturePlan(P, S1), sector(P),$

$writeln(OKPOSS), OP = [A1|P];$

$writeln(NOPOSS), OP = A)),$

$writeln(checkPlanOP =), writeln(OP)).$

Osservando tale procedura è possibile notare che ne richiama altre due, ognuna di rilevante importanza: $checkfuturePlan(P, S1)$ e $sector(P)$. Le procedure introdotte sono finalizzate a verificare che per ogni task inserito ci siano le pre-condizioni per l'eseguibilità, e inoltre si occupano di garantire che, in ogni task, non ci siano più azioni destinate alla stessa unità funzionale.

La prima procedura è riportata assume la seguente forma:

$checkfuturePlan([P1|P2], S1) : -$

$time(P1, T1), poss(P1, S1),$
$checkfuturePlan(P2, do(P1, S1)).$

La procedura $poss(a, s)$, come abbiamo visto precedentemente, verifica l'esistenza, nella situazione s, delle precondizioni necessarie per l'esecuzione dell'azione a. Inoltre la procedura $poss$ può essere invocata anche per i task; in questo caso verifica che per ogni azione del task risultino verificate tutte le pre-condizioni.

La procedura $checkfuturePlan$ pertanto, non fa altro che invocare la procedura poss per ogni task inserito.

Tuttavia, la procedura poss per verificare il primo task da eseguire, $P1$, ha a disposizione la situazione in cui avverrà tale task, ossia il parametro $S1$; invece per verificare il task successiva a $P1$, la procedura poss si deve calcolare la situazione $S2$. Ovviamente la situazione $S2$ in cui verrà eseguita il task successivo a $P1$, non è altro che $S2 = do(P1, S1)$: lo stato derivante dall'esecuzione di $P1$ nella situazione corrente $S1$. Per quanto riguarda la seconda procedura sector invece, è responsabile di garantire che all'interno di un task non ci siano più azioni dirette allo stesso modulo funzionale.

Quest'ultima ne utilizza varie di supporto: è stata introdotta una procedura $action_sector$ che raggruppa le azioni in base al modulo funzionale al quale sono dirette: data un'azione, restituisce il numero identificativo del modulo. Un'ulteriore procedura invocata da sector è $value_is_different$: viene invocata per verificare che all'interno di un task, non ci siano azioni che, a seguito dell'invocazione di $action_sector$, restituiscono lo stesso numero.

Sia durante l'esecuzione di sector che durante l'esecuzione di $checkfuturePlan$, vengono richiamate anche altre procedure specifiche al tipo di piano ricevuto dall'interfaccia: a seconda del numero di task contenuti nel piano interven-

gono procedure opportune al fine di effettuare adeguatamente le verifiche. L'introduzione di quest'ultime procedure è stata una conseguenza dell'estensione dell'orizzonte temporale pianificabile: il fatto di avere più task da valutare, ognuno dei quali è strettamente legata alle precedenti, ha comportato l'estensione delle procedure mirate alla valutazione del piano.

Durante tutto il processo di verifica del piano, il pianificatore conserva sempre il piano da lui inizialmente prodotto; quest'ultimo verrà utilizzato nel momento in cui il piano dell'operatore non superasse tutte le verifiche. In particolare tale piano non solo tiene memoria di task che devono essere eseguite in futuro ma memorizza anche tutte le azioni che si sono svolte a partire dall'inizio della missione, con i relativi tick di esecuzione.

Il pianificatore aggiorna continuamente queste informazioni, sia considerando le azioni realmente eseguite, sia quelle che dovranno essere eseguite in futuro. Pertanto il caso in cui il piano dell'operatore risulti fattibile, implica la necessità di memorizzare le azioni in esso contenute nel piano conservato dal pianificatore, Il pianificatore aggiorna continuamente queste informazioni, sia considerando le azioni realmente eseguite, sia quelle che dovranno essere eseguite in futuro. Pertanto il caso in cui il piano dell'operatore risulti fattibile, implica la necessità di memorizzare le azioni in esso contenute nel piano conservato dal pianificatore, in modo tale che, al tick successivo, si proceda con l'esecuzione dei giusti task, nell'opportuno ordine.

Quindi nel piano memorizzato dal pianificatore non compariranno più le azioni scelte da quest'ultimo ma quelle inserite dall'esterno ovviamente con i task già attivate nel passato.

Sia per questo motivo e sia per il fatto di poter avere una serie di task, anziché un solo che deve essere eseguita nel futuro, è stato necessario l'inserimento di una procedura $createFuture$ che, a partire dalla lista di task

inserite dall'utente, aggiorna il piano di cui il pianificatore tiene memoria, calcolando iterativamente la situazione che si avrebbe nel tick $St + 1$ come l'esecuzione dell'azione a t nella situazione St:

$createfuture(A1, S1, SD, SN) :- A1 = [A11|A22], SD = do(A11, S1),$ $(createfuture(A22, SD, SF, SN); SN = SD).$

Dall'esecuzione di tale procedura si otterrà quindi il piano aggiornato: $createFuture$ mette insieme le azioni già avvenute con quelle che dovranno essere eseguite. Al termine dell'esecuzione si ottiene un piano del tipo:

$S : [navigation_wand_end(_5106, _53073...1.0Inf),$
$percept_attention_end(_5307)]$
$do[navigation_wand_start(_49872..1.0Inf)]$
$do[percept_attention_start(2), map_start(2))]$
$do[abnormal_map_end([_4109, _4111, _4113], 1)]do\ s0$
avremmo che le azioni:
$abnormal_map_end([_4109, _4111, _4113], 1)$ è avvenuta nel tick 1;
$[percept_attention_start(2), map_start(2)]$ sono avvenute entrambe nel tick 2;
$navigation_wand_start(_49872...1.0Inf)]$ non è stata ancora eseguita; essendo la prima azione non eseguite incontrata nel piano, allora tale azione rappresenta quella che dovrà essere eseguita nel tick presente;
$[navigation_wand_end(_5106, _53073...1.0Inf)$ e
$percept_attention_end(_5307])$ non sono ancora state eseguite; verranno eseguite al tick successivo a quello in cui verrà eseguita

$navigation_wand_start(_49872...1.0Inf)$] rappresenta quindi, il piano futuro.

Osservando il piano S, i task che compaiono in testa, sono quelli che devono ancora essere eseguite. Questi ultimi ovviamente sono differenti da quelli già attivati. Ogni azione, da come si può notare, contiene un attributo che rappresenta l'istante in cui ne è avvenuta l'attivazione. Tale variabile inizialmente non assume un valore rigido ma piuttosto quello di un intervallo: solo quando l'azione viene attivata, tale variabile assume il valore dell'istante in cui è avvenuta l'esecuzione. Bisogna considerare però che se l'azione non viene eseguita entro l'intervallo di tempo indicato verranno attivate le procedure di recupero.

Osservando il valore della variabile che indica l'istante di attivazione di un'azione è quindi possibile ricavare le azioni, e quindi i task, che dovranno essere eseguite. All'inizio di ogni tick del pianificatore, verranno prelevate dal piano i task che devono essere attivati e si procederà con il loro invio all'interfaccia. Quando quest'ultima riceve il piano, mostra le azioni che stanno avvenendo nel tick presente, e tali azioni, come visto precedentemente, non sono soggette ad eventuali modifiche, sia quelle che saranno eseguite a partire dal tick successivo a quello considerato.

Un altro fenomeno che può capitare durante l'esecuzione del piano è quello relativo alle azioni che non risultano controllabili: ossia il modulo funzionato associato all'azione da eseguire non ha ancora inviato il messaggio di attivazione di tale azione. Come abbiamo visto precedentemente, questo è un comportamento associato alle azioni di end.

Tuttavia nel momento in cui si procede con l'esecuzione del task che presenta map_end accade che tale azione risulta non controllabile e certamente trascorrerà del tempo prima che lo diventi; fino a quel momento il sistema

procede non attivando nessuna azione. In questo caso si entra in uno stato di "attesa": il robot continua con la costruzione della mappa e la rilevazione delle vittime e non verranno pianificate altri task fino a quando sono sarà possibile eseguire quella contenete *map_end*.

Questa è una situazione tipica che caratterizza il sistema di controllo in esame. Generalmente il pianificatore procede selezionando inizialemente la task di $initANT$, in seguito manda in esecuzione ciclicamente le task: $wanderANT$, *search* e *visprocess*.

La task di inizializzazione $initANT$ è della forma: $proc(initANT, pi(t1, [map_start(t1), percept_attention_start(t1)]))$,

mentre la task

$wanderATN : proc(wanderATN, pi(t2, [navigation_wand_start(t2)] : pi(x, pi(d, pi(t3, [navigation_wand_end(x, t3), percept_attention_end(t3)])))))$.

Come si può vedere, mentre in $t1$ viene attivata l'azione *percept_attention_start*, dopo due tick, quindi in $t3$, è richiesta l'attivazione di *percept_attention_end*.

Precedentemente è stato visto che un'azione del tipo $endX$, affinché possa essere eseguita, non solo deve verificare la procedura poss ma deve anche risultare controllabile. Solitamente l'azione *percept_attention_end*, a seguito dell'attivazione di *percept_attention_start*, non risulta immediatamente controllabile: l'esecuzione della task contenente *percept_attention_end* viene quindi rimandata ed il robot continua così nel suo stato di *wander* e *visProcess*.

L'iniziativa mista introdotta, permette di gestire questa situazione di "attesa" in vari modi. L'operatore può semplicemente decidere di non in-

tervenire, può inserire delle azioni da eseguire durante l'attesa oppure può togliere l'azione che causa tale stato. Inoltre in questa situazione, più che nelle altre, l'operatore potrebbe trovar utile inserire delle azioni che possano essere attivate, nel giro di poco tempo e in qualsiasi situazione, al fine di forzarne l'esecuzione. Sono presenti infatti, delle azioni (*AbnormalAction*), una per ogni azione di end, sempre controllabili: sono azioni che hanno le stesse pre-condizioni e gli stessi effetti delle azioni originali ma verificano sempre il test di controllabilità, risultando così sempre eseguibili. Tali azioni intervengono durante le procedure di recupero del sistema e quando è necessario forzare l'esecuzione di una task, durante l'iniziativa mista già presente sul sistema; tuttavia nulla vieta all'operatore di utilizzarle anche durante la sua attività di pianificazione.

Capitolo 9

Visualizzatore

9.1 Introduzione

Il visualizzatore, che costituisce il nocciolo di questa tesi, è stato progettato e implementato con l'ausilio delle librerie grafiche, open source, *wxWidgets*. Queste, che sno utilizzate in un ambiente C++, con Microsoft .Net, grazie ala loro versatilità, hanno permesso l'implementazione del visualizzatore che permette l'analisi dei task relativi al piano, nei moduli di SLAM, Navigazione e Visione.

La struttura del visualizzatore è una griglia, il cui scheletro è indipendente dalle informazioni che il visualizzatore permette di osservare. Infatti, il Pianificatore genera una serie di file .log, che contengono i nomi dei task che costituiscono il piano e i loro colori, per la parte strettamente visiva. Questi file .log vengono passati in lettura al visualizzatore - dopo l'esecuzione di un piano - che può dunque mostrare all'operatore il piano effetivamente eseguito dal robot, durante la navigazione. Ogni task è evidenziato da un suo colore specifico; possiede una sua durata, un inizio e una fine, che possono essere ulteriormente visualizzati, attraverso finestre *pop-up*, cliccando sul task relativo

quando è visualizzato nella sua barra.

In questo modo, con l'interazione tra la Gui e il visualizzatore, l'operatore può effettivamente rendersi conto di quello che il robot sta eseguendo, in modo tale da poter effettuare delle modifiche e correzioni durante l'esplorazione. Questo è fondamentale se si opera con il robot in scene soccorso, dove è importante avere un feedback quasi istantaneo dell'operato del robot.

Figura 9.1: Visualizzatore

9.2 OnInit()

All'inizio del codice viene inizializzata l'applicazione con il seguente codice: **bool Fireball::OnInit()**.

9.2.1 MyFrame()

Il frame principale, **MyFrame()**, contenuto all'interno dell'applicazione viene lanciato:

```
myframe = new MyFrame(NULL);
myframe->Show( true );
return true;
```

9.3 Lettura da File .txt

Le due seguenti righe di codice permettono la lettura dei dati caricati da file di testo:

```
/* Lettura da File di Testo */
wxFileInputStream stream("fireball_taskname.txt");
wxFileInputStream stream2("fireball_color.txt");
wxFileInputStream stream3("fireball_string.txt");
wxFileInputStream stream4("fireball_serv.txt");
wxFileConfig fireball_taskname(stream);
wxFileConfig fireball_color(stream2);
wxFileConfig fireball_string(stream3);
wxFileConfig fireball_serv(stream4);
```

9.4 Costruttore MyFrame()

Il costruttore del frame principale, **MyFrame()**, ha presenti al suo interno i comandi necessari per mostrare un menù con delle informazioni riguardan-

ti il programma e la barra posta in basso all'applicazione.

```
/* MyFrame: Costruttore del Frame Principale: contiene tutti i
pannelli */


MyFrame::MyFrame(wxFrame* parent) :wxFrame(parent,-1,_T("Fireball"),
          wxPoint(0,0),
          wxSize(1024,768),
          wxDEFAULT_FRAME_STYLE){
```

9.5 I Due Pannelli

A questo punto vengono introdotti i due pannelli:

- **BigPanel()**;
- **StartPanel()**;

9.5.1 BigPanel()

```
    /* BigPanel: Il Pannello Principale */
BigPanel::BigPanel(wxWindow* parent) :wxPanel(parent,-1,
          wxPoint(0,0),
          wxSize(1024,768),
          wxDOUBLE_BORDER){
```

BigPanel() è il pannello più grande, contenuto nel frame MyFrame(), che a sua volta contiene tutte le altre strutture.

Questo pannello contiene anche i pulsanti Start, ZoomIn, ZoomOut, Fit, che permettono di controllare le barre dei task di Navigazione, Visione e Slam.

Questi pulsanti, sebbene dichiarati nella classe StartPanel(), sono gestiti dalla classe BigPanel(). I pulsanti sono posti nella parte superiore del frame, per una maggiore visibilità.

```
/* StartPanel */ class StartPanel:public wxPanel{ public:

    /* I pulsanti Start, ZoomIn, ZoomOut, Fit:
    la loro gestione è affidata alla classe BigPanel */
    wxButton *start,*zibtn,*zobtn,*zfbtn;
    StartPanel(wxWindow* parent);
};
```

Successivamente abbiamo le barre dei task: BarSlam, BarNav, BarVisio:

```
    /* Barre dei moduli di: SLAM, Navigazione, Visione */
    slam=new BigGrid(this,BarSlam);
    nav=new BigGrid(this,BarNav);
    visio=new BigGrid(this,BarVision);
```

I nomi dei tre moduli sono letti da file:

```
/* I nomi dei tre moduli */
    wxString SLAM = fireball_taskname.Read("SLAM", "");
    wxString Nav = fireball_taskname.Read("Nav", "");
    wxString Visio = fireball_taskname.Read("Visio", "");
```

All'inizio, istante di tempo corrente uguale a zero, richiamiamo la funzione AddTask(), che non aggiunge alcun task, all'inizio.

```
currenttime=0;
```

```
slam->AddTask(0,NONE);
nav->AddTask(0,NONE);
visio->AddTask(0,NONE);
```

Di seguito sono collocati i due pannelli.

I Semafori

All'interno del BigPanel sono presenti anche oggetti della classe wxSemaphore(), i cosiddetti semafori.

Il primo semaforo svolge il compito di far aprire una sola finestra di modifica alla volta.

È controllato dalle funzioni LockModify() e UnLockModify(). Nello specifico il semaforo viene posto a 1 per indicare che sarà soltanto uno il numero massimo di utenti che potranno utilizzare il semaforo.

Il secondo semaforo blocca l'accesso a tutte le risorse della barra dei task, consentendo a un solo thread alla volta di leggere e/o modificare i dati relativi a quel task.

È controllato dalle funzioni LockBar() e UnLockBar().

In generale i semafori vengono utilizzati per limitare l'accesso a risorse condivise. I task iniziano tutti dalla posizione corrente *cur = 0*.

9.5.2 StartPanel()

```
    /* StartPanel: Costruttore del Pannello di Start */
/* Contiene il pulsante di Start e i pulsanti per lo Zoom,
   il Fit dei Task */
StartPanel::StartPanel(wxWindow* parent)
```

```
:wxPanel(parent,-1,wxDefaultPosition,wxDefaultSize,wxDOUBLE_BORDER){
```

StartPanel() è il secondo pannello.
Esso contiene materialmente i pulsanti Start, ZoomIn, ZoomOut e Fit, ed è posto alla sommità al frame, per una maggiore visibilità.

Ricordiamo che la gestione dei pulsanti è demandata al pannello BigPanel(), in quanto StartPanel() funziona solo da contenitore per questi pulsanti.

```
/* StartBox: il box con il pulsante di Start, quelli di Zoom,
di Fit e di Lock */
wxStaticBox *startBox=new wxStaticBox(this,-1,"Quadro Pulsanti");
wxStaticBoxSizer* startsiz=new wxStaticBoxSizer(startBox,wxHORIZONTAL);
/* Il box con il pulsante di Start */
start=new wxButton(this,STARTBTN,"Start",
wxDefaultPosition,wxSize(120,70));
/* I bottoni di Zoom In, Zoom Out, Fit e Lock */
zibtn=new wxButton(this,ZOOMINBTN,"Zoom In",
wxDefaultPosition,wxSize(120,70));
zobtn=new wxButton(this,ZOOMOUTBTN,"Zoom Out",
wxDefaultPosition,wxSize(120,70));
zfbtn=new wxButton(this,ZOOMFITBTN,"Fit",
wxDefaultPosition,wxSize(120,70));
/* I bottoni sono sistemati al loro posto, nei loro rispettivi box */
startsiz->Add(30,15);
startsiz->Add(start,0,wxGROW|wxSHAPED|wxALIGN_CENTER);
startsiz->Add(30,15);
startsiz->Add(zibtn,0,wxGROW|wxSHAPED|wxALIGN_CENTER);
startsiz->Add(30,15);
```

```
startsiz->Add(zobtn,0,wxGROW|wxSHAPED|wxALIGN_CENTER);
startsiz->Add(30,15);
startsiz->Add(zfbtn,0,wxGROW|wxSHAPED|wxALIGN_CENTER);
SetAutoLayout( TRUE );
SetSizer( startsiz );
startsiz->Fit( this );
startsiz->SetSizeHints( this );
```

9.6 Le Funzioni

Incontriamo per prime due "funzioni di servizio" che sono all'interno del frame principale.

MyFrame() ha due "eventi di servizio", entrambi contenuti nel menù File sulla sinistra in alto.

```
/* Eventi Menu MyFrame */ void MyFrame::OnQuit(wxCommandEvent&
WXUNUSED(event)) {
    // Chiude il Frame
    Close(true);
}
```

Il primo evento, **On Quit()**, chiude il frame; il secondo, **OnAbout()**, fornisce informazioni sul programma.

9.6.1 OnBarDBClick()

OnBarDBClick(): questa funzione permette di leggere i dati e inviarli alla finestra popup, che viene lanciata cliccando col tasto sinistro del mouse sui task delle barre di Navigazione, Visione e Slam.

Anche in questo caso avviene la lettura da file di testo delle informazioni delle label dei moduli.

```
void BigPanel::OnBarDBClick(wxGridEvent& ev){
    wxString slamprint = fireball_taskname.Read("slamprint", "");
    wxString navprint = fireball_taskname.Read("navprint", "");
    wxString visioprint = fireball_taskname.Read("visioprint", "");
```

Tramite i parametri ”label”, ”data” e ”info”, fornisce i dati relativi ai task presenti nelle tre diverse barre e li invia alla finestra popup, che si lancia con un doppio click sul task di cui vogliamo conoscere i dati relativi:

```
    long info;
    wxString data,label;
    LockBar();
    switch(ev.GetId()){
    case BarSlam:  // Barra di SLAM
        info=slam->GetGridCursorCol();
        label.Printf(slamprint); // etichetta della finestra popup
        data.Printf("\n  Task: %s\n  Start: %s\n  End: %s\n  Length: %s",
                    slam->datagrid->GetCellValue(info,NAME),
        slam->datagrid->GetCellValue(info,HEAD), // Start
        slam->datagrid->GetCellValue(info,TALE), // End
        slam->datagrid->GetCellValue(info,DIMENSION)); // Length
```

Di ogni singolo task è possibile conoscere il modulo in cui è inserito, il nome NAME, l'inizio START, la fine END e la lunghezza LENGTH.

La funzione ripete le operazioni per ognuna delle tre barre: Navigazione, Visione, Slam.

È presente subito dopo la funzione UnLockBar(): questa, come abbiamo visto sopra, permette di bloccare il *flagsemaphore*, permettendo a un solo thread alla volta di accedere al task.

È presente infine la funzione che permette di lanciare la finestra *popup*.

```
/* Questo fa comparire la finestra al doppio click */
clkpopup=new LeftClickFrame(this,-1,label,data);
```

9.7 LeftClickFrame()

Dopo aver raccolto i dati, passiamo materialmente alla finestra popup, chiamata LeftClickFrame().

```
    /* LeftClickFrame::LeftClickFrame() */
/* Informazione sui Task */
LeftClickFrame::LeftClickFrame(wxWindow*parent,
                               wxWindowID id,
                               wxString& title,
                            wxString& mydata):wxFrame(parent,id,title,
                            wxDefaultPosition, wxSize(600,600)){
```

La finestra lanciata col tasto sinistro del moluse non è altro che un altro frame, interno al frame principale.

Il suo compito è quello di raccogliere e mostrare all'operatore i dati relativi al task che si clicca: il nome della barra dei task a cui appartiene il task, ovvero Navigazione, Visione, Slam; il nome del task; l'inizio del task, la fine del task; la lunghezza del task in tick, data dalla differenza di fine e inizio.

È presente un tasto di "ok" per chiudere la finestra, ed è l'unica funzione presente in questo frame.

```
    /* Eventi LeftClickFrame */
void LeftClickFrame::OnOk(wxCommandEvent& WXUNUSED(event)) {
    // Chiude il Frame
    Close(true);
    // Distrugge i Frame alla chiusura del Frame Principale
    Destroy();
}
```

9.8 ReadPlan()

```
    /* BigPanel::ReadPlan(wxString plan[]): legge il Piano */ void
BigPanel::ReadPlan(wxString plan[]){
```

Al passo successivo c'è la funzione che materialmente ha il compito di leggere il Piano: ReadPlan().

La funzione ReadPlan() ha il compito di interfacciarsi tra l'applicazione e la logica di controllo di sistema del robot, il pianificatore.

Essa riceve come parametro una stringa, di tipo wxString, plan[i], che contiene i dati sulla gestione dei task.

```
  long arrivetick;
    int i=1;
    // booleani per slam, nav, visio
    bool newslamplan=false;
    bool newnavplan=false;
    bool newvisioplan=false;
    // leggo il tick corrente e aggiorno le barre di avanzamento
    plan[0].ToLong(&arrivetick); // conversione da stringa a long
```

```
    // aggiungo alle barre il numero di tick necessario...
    AddTick(arrivetick-currenttime);
    // ... e modifico il tempo corrente
    currenttime=arrivetick;
    // leggo le azioni fino alla prima interruzione (stringa vuota)...
    while(plan[i]!=wxEmptyString){ // stringa vuota
        // ... e aggiungo il task letto
        // il controllo è eseguito nella funzione chiamata
```

ReadPlan legge quei dati; il primo elemento è proprio il numero di tick che indica a che istante è arrivato il programma; si calcola dunque la differenza tra il tick corrente e quello a cui il programma è arrivato e si fornisce questo dato alla funzione AddTick().

A seconda di quello che è specificato nelle stringhe, la funzione ReadPlan richiama una delle funzioni di Add; l'array letto deve essere terminato da una stringa vuota, *wxEmptyString*.

La seconda parte, separata da una stringa vuota, permette di tener conto anche dei dati relativi ai piani dei moduli in esecuzione. Ricordiamo che i moduli sono composti da più task e che è possibile ricevere un piano composto da più task.

Il primo argomento dell'array è il tick corrente, ovvero l'istante iniziale relativo alla creazione del piano. Come agisce l'interfaccia?

Fa avanzare la barra nel modo seguente.

L'avanzamento è pari alla differenza tra il valore del tick ricevuto e quello memorizzato come corrente:

```
AddTick(arrivetick-currenttime);
```

Il tick corrente viene aggiornato al valore del tick ricevuto:

```
currenttime=arrivetick;
```

Dopo il tick corrente, vengono effettivamente letti i piani, le cui stringhe indicano i task che devono essere eseguiti dopo il tick corrente. Se i task indicati non sono uguali a quelli attualmente in esecuzione, la funzione blocca il task corrente e aggiunge alla barra corrispondente il task ricevuto.

La lettura dei piani va avanti finché non compare una stringa vuota.

```
/* Le Nove Funzioni di Aggiunta dei Task */
       if(plan[i]=="SlamRunning"){      AddSlamRunning();}
       else{ if(plan[i]=="SlamStop"){   AddSlamStop();}
       else{ if(plan[i]=="SlamScan"){   AddSlamScan();}
       else{ if(plan[i]=="NavStop"){    AddNavStop();}
       else{ if(plan[i]=="NavExplore"){ AddNavExplore();}
       else{ if(plan[i]=="NavGoto"){    AddNavGoto();}
       else{ if(plan[i]=="NavWander"){  AddNavWander();}
       else{ if(plan[i]=="VisIdle"){    AddVisIdle();}
       else{ if(plan[i]=="VisProcess"){ AddVisProcess();}
```

Dopo la stringa vuota c'è un'altra serie di task, i task pianificati.

```
/* La serie dei Task Pianificati */
// Aggiungo le azioni pianificate
while(plan[i]!=wxEmptyString){
// le stringhe possono contenere sia il nome del task...
    if(plan[i].Contains("SlamRunning")){
        // ... sia la durata prevista
```

```
        long value;
        if(newslamplan==false) slam->DeletePlan();
        newslamplan=true;
        //verifico se e' specificata la durata
        if(plan[i].AfterFirst(' ').ToLong(&value)){
            slam->AddPlan(SLAMRUN,value);
        }
        else{ //altrimenti specifico la durata di default
            slam->AddPlan(SLAMRUN,STDDURATION);
        }
    }
```

Quando la funzione si accorge che, per un determinato modulo, è indicato un piano, cancella il piano corrente e lo sostituisce con quello ricevuto.

Vengono eliminati tutti i task pianificati, tranne quello che è in eventuale fase di modifica da parte dell'operatore. Quest'ultimo task del piano diventa il primo, mentre il nuovo piano viene appeso in coda a questo.

L'array deve essere terminato da un'altra stringa vuota.

Compare la variabile **STDDURATION**, specificata nel file .h, che vale *STDDURATION 30.*

9.8.1 LockModify()/UnLockModify()

```
    /* BigPanel::LockModify() e BigPanel::UnLockModify() */
/* Bloccano e sbloccano i semafori, permettendo una sola modifica
alla volta */ bool BigPanel::LockModify(){
    if(modify_semaphore->TryWait()!=wxSEMA_NO_ERROR ) return false;
    return true;
}
```

```
void BigPanel::UnLockModify(){
    modify_semaphore->Post();
}
```

LockModify() e **UnLockModify()** bloccano e sbloccano i semafori, permettendo una sola modifica alla volta.

9.8.2 LockBar()/UnlockBar()

```
/*BigPanel::LockBar() e BigPanel::UnlockBar() */ /* Bloccano e
sbloccano i semafori, impedendo l'intervento sui dati delle barre
dei task, e sull'interfaccia */ void BigPanel::LockBar(){
    flag_semaphore->Wait();
}

void BigPanel::UnlockBar(){
    flag_semaphore->Post();
}
```

Subito dopo sono presenti le funzioni **LockBar()** e **UnlockBar()**, che, come visto in precedenza, hanno il compito di permettere l'accesso a un task da parte di un solo thread alla volta.

9.8.3 Funzioni richiamate da ReadPlan()

Dopo queste funzioni di blocco, abbiamo le funzioni che vengono richiamate da parte della funzione **ReadPlan()**, descritta in precedenza, che servono per aggiungere i task.

```
/* Funzioni di Aggiunta Task */ /* BigPanel::AddSlamRunning() */
void BigPanel::AddSlamRunning(){
    if(slam->currentact!=SLAMRUN){
        if(slam->currentact!=INIT){
            slam->EndTask(currenttime);
        }
        slam->AddTask(currenttime,SLAMRUN);
    }
}
```

Esse sono, nell'ordine in cui sono presenti all'interno dei thread:

- AddSlamRunning();
- AddSlamScan();
- AddSlamStop();
- AddNavExplore();
- AddNavGoto();
- AddNavStop();
- AddNavWander();
- AddVisIdle();
- AddVisProcess().

Queste funzioni hanno anche il compito di controllare se il task che vogliamo aggiungere non sia già in esecuzione.

Se lo è già, richiamano la funzione AddTask() del modulo corrispondente - Navigazione, Visione, Slam - e passano come parametro, a tale funzione, il tick corrente e il tipo di task richiesto, secondo il seguente codice:

```
void BigPanel::AddNavExplore(){
    if  (nav->currentact!=NAVEXPL){
        if(nav->currentact!=INIT){
            nav->EndTask(currenttime);
        }
        nav->AddTask(currenttime,NAVEXPL);
    }
}
```

9.8.4 AddTick

Dopo queste funzioni, c'è la funzione AddTick che fornisce alle tre barre un numero di tick pari a quello specificato nel parametro, passandolo alla funzione Tick() di ogni barra.

```
/* BigPanel::AddTick */ void BigPanel::AddTick(long tick){
    nav->Tick(tick);
    visio->Tick(tick);
    slam->Tick(tick);
}
```

9.8.5 GetSlamPlan()/GetNavPlan()/GetVisioPlan()

```
/* Funzioni che Restituiscono i Task */ /* BigPanel::GetSlamPlan()
*/ wxString* BigPanel::GetSlamPlan(){
    wxString *outarray=new wxString[slam->planoffset+1];
```

```
    for(long i=1;i<=slam->planoffset;i++){
            LockBar();
        slam->GetPlanTaskString(i,outarray[i-1]);
        UnlockBar();
    }
    outarray[slam->planoffset]=wxEmptyString;
    return outarray;
}
```

GetSlamPlan(), **GetNavPlan()**, **GetVisioPlan()**, corrispondono a ognuno dei tre thread presenti nell'interfaccia. Il loro compito è ottenere dalla barra relativa - *Slam, Navigazione, Visione* - una stringa che contenga le seguenti informazioni: il nome e la durata dei task pianificati. Queste informazioni vengono poi passate al pianificatore che si occuperà recuperare i piani che sono stati nel caso modificati dall'operatore.

Richiama la funzione *GetPlanTaskString()*, descritta più avanti.

9.9 BigGrid

Ancora dopo è presente la funzione che definisce la griglia dei thread e dei task. È la funzione BigGrid(), il cuore del progetto. Questa funzione che deriva da wxGrid, contiene al suo interno un oggetto DataGrid, ovvero una griglia di dati che rappresentano i vari task. Ogni cella è un task, di qualsiasi tipo, correntemente in esecuzione.

```
 /* BigGrid: la Barra dei Task*/
BigGrid::BigGrid(wxWindow* parent, wxWindowID id)
:wxGrid(parent,id,wxDefaultPosition,wxSize(1024,100)){
```

```
    SetRowMinimalAcceptableHeight(0);
    SetColMinimalAcceptableWidth(1);
    SetRowMinimalHeight(0,1);
    SetDefaultCellBackgroundColour(GetParent()->
    GetBackgroundColour());
    datagrid = new wxGrid( this,
                      -1,
                      wxPoint( 0, 0 ),
                      wxSize( 1, 1 ) );
    datagrid->CreateGrid( 1, 6 );
    datagrid->SetRowSize( 0, 0 );
    datagrid->Show(FALSE);
    datagrid->SetColFormatNumber(HEAD);
    datagrid->SetColFormatNumber(TALE);
    datagrid->SetColFormatNumber(DIMENSION);
    datagrid->SetColFormatNumber(TYPE);
    zoomfactor=1;
    currentcell=-1;
    datagrid->DeleteRows(0,1);
    currentact=NONE;
    planoffset=0;
    newplan_semaphore=new wxSemaphore(1,1);
}
```

9.9.1 GoGrid()

La funzione che inizializza la griglia è la GoGrid(), che richiama tutti i parametri necessari per dare l'aspetto "fisico" della griglia.

```
    /* GoGrid: Inizializza la Griglia */

    void BigGrid::GoGrid(const wxString& name){

    //Crea una griglia di 1 riga e 0 colonne
    CreateGrid(1,0);
    //Dimensione delle righe
    SetRowSize(0,80);
    //Impedisce al n
    DisableDragColSize();
    DisableDragRowSize();
    //Larghezza Etichetta Colonne
    SetColLabelSize(0);
    //Il nome del singolo modulo
    SetRowLabelValue(0,name);
    //Larghezza Etichetta del singolo modulo
    SetRowLabelSize(80);
}
```

9.10 Lettura da File di Testo Nomi dei Task

Vengono poi impostati da lettura da file .txt i nomi dei task che sono eseguiti in ogni cella. Per fare questo si richiama la funzione Read() e il relativo file di testo in cui si specificano i nomi dei vari task. Queste letture

sono effettuate all'interno della funzione AddTask, un'altra funzione chiave del programma.

```
/* Lettura dei nomi dei task da file di testo */

/* SLAM */

wxString slamrunning = fireball_taskname.Read("slamrunning", "");
wxString slamstop = fireball_taskname.Read("slamstop", ""); wxString
slamscanning = fireball_taskname.Read("slamscanning", "");

/* Nav */

wxString navstopp = fireball_taskname.Read("navstopp", "");

wxString navexplore = fireball_taskname.Read("navexplore", "");

wxString navgoto = fireball_taskname.Read("navgoto", "");

wxString navwander = fireball_taskname.Read("navwander", "");

/* Visio */

wxString visioprocess = fireball_taskname.Read("visioprocess", "");

wxString visioidle = fireball_taskname.Read("visioidle", "");

/* Casella Nera */
```

```
wxString none = fireball_taskname.Read("none", "");
```

9.11 AddTask()

```
    /* BigGrid::AddTask(int currenttime,int type=NONE) */
/*Aggiunge un Task alla Barra Corrente, specificandone il Tick
Corrente e l'Id del Tipo di Task*/ void BigGrid::AddTask(int
currenttime,int type=NONE){
        if(currenttime!=0) ((BigPanel*)GetParent())->LockBar();
        currentcell++;
        InsertCols(currentcell);
        datagrid->InsertRows(currentcell);
        wxString msg;
        msg.Printf("%d",currenttime);
        datagrid->SetCellValue(currentcell,HEAD,msg);
        datagrid->SetCellValue(currentcell,DIMENSION,_T("0"));
        datagrid->SetCellValue(currentcell,TALE,_T("-1"));
        datagrid->SetCellValue(currentcell,FLAG,wxEmptyString);
        SetReadOnly( 0, currentcell );
        SetCellOverflow(0, currentcell, TRUE);
        SetColMinimalWidth(currentcell,0);
        SetColSize(currentcell,0);
```

I suoi parametri sono il tempo corrente e il tipo. Essa aggiunge il task specificato dal parametro, iniziando dal tempo che è fornito dal tick in currenttime. Aggiunge una nuova riga in datagrid e inserisce nella cella relativa, il nome, il tipo di task, l'istante di inizio e aumenta di uno l'indice di cur-

rentcell. Questi dati saranno disponibili cliccando col tasto sinistro sul task relativo, ovvero sulla cella, all'interno della funzione LeftClickFrame(). Nel caso in cui non ci sia alcun parametro viene scritto, per il task, NONE. Questo parametro è necessario per mantenere il sincronismo tra le varie barre, nel caso in cui i moduli non fossero avviati contemporaneamente.

Come per i nomi dei task, anche i colori che ne evidenziano la funzione nelle relative barre sono letti da testo.

```
/* Lettura dei colori da file di testo */
        /* Valgono per tutte e tre le barre */
        wxColour red = fireball_taskname.Read("red", "");
        wxColour lightgrey = fireball_taskname.Read("lightgrey", "");
        wxColour blue = fireball_taskname.Read("blue", "");
        wxColour cyan = fireball_taskname.Read("cyan", "");
        wxColour black = fireball_taskname.Read("black", "");
```

- **Red**: Slam Running; Nav Goto; Visio Process;
- **Light Grey**: Slam Stop; Nav Stop ;Visio Idle;
- **Blue**: Slam Scan; Nav Explore
- **Cyan**: Nav Wander;
- **Black**: None;

Sono presenti tutti i vari casi relativi a Slam, Navigazione e Visione. Inoltre sono presenti le chiamate a due funzioni LockBar() e UnlockBar() di BigPanel(), che garantiscono l'accesso esclusivo ai dati del task. Inoltre viene inizializzato il campo FLAG come stringa vuota.

```
switch(type){
            /* TASK DI SLAM */
        case SLAMRUN: // SlamRun
            SetCellBackgroundColour(0, currentcell, red);
            datagrid->SetCellValue(currentcell,NAME,_T(slamrunning));
            msg.Printf("%d",SLAMRUN);
            datagrid->SetCellValue(currentcell,TYPE,msg);
            msg.Printf("%d  Running",currenttime);
            SetCellValue(0,currentcell,msg);
            currentact=SLAMRUN;
            break;
        case SLAMSTOP: // SlamStop
            SetCellBackgroundColour(0, currentcell, lightgrey);
            datagrid->SetCellValue(currentcell,NAME,_T(slamstop));
            msg.Printf("%d",SLAMSTOP);
            datagrid->SetCellValue(currentcell,TYPE,msg);
            msg.Printf("%d  Stop",currenttime);
            SetCellValue(0,currentcell,msg);
            currentact=SLAMSTOP;
            break;
        case SLAMSCAN:// SlamScan
            SetCellBackgroundColour(0, currentcell, blue);
            datagrid->SetCellValue(currentcell,NAME,_T(slamscanning));
            msg.Printf("%d",SLAMSCAN);
            datagrid->SetCellValue(currentcell,TYPE,msg);
            msg.Printf("%d  Scan",currenttime);
            SetCellValue(0,currentcell,msg);
```

```
            currentact=SLAMSCAN;
            break;
```

9.12 AddPlan()

```
/* BigGrid::AddPlan(int type, int duration) */ /* Aggiunge un Piano
*/ void BigGrid::AddPlan(int type, int duration){
        wxString msg;
        msg.Printf("%d",duration);
        ((BigPanel*)GetParent())->LockBar();
        planoffset++;
        AppendCols(1);
        datagrid->AppendRows(1);
        datagrid->SetCellValue(currentcell+planoffset,DIMENSION,msg);
        datagrid->SetCellValue(currentcell+planoffset,FLAG,wxEmptyString);
        SetColMinimalWidth(currentcell+planoffset,0);
        SetReadOnly( 0, currentcell+planoffset );
        SetCellOverflow(0, currentcell+planoffset, TRUE);
        SetColSize(currentcell+planoffset,
        (int)duration*zoomfactor*COLSIZE);
```

Questa funzione appartiene anch'essa a BigGrid. Ha lo stesso scopo di AddTask(), ma i task aggiunti sono quelli pianificati.

```
    /* Lettura dei nomi dei Task Pianificati da file di testo */

/* PLAN SLAM */
```

```
wxString planslamrunning = fireball_taskname.Read("planslamrunning",
"");

wxString planslamstop = fireball_taskname.Read("planslamstop", "");

wxString
planslamscanning=fireball_taskname.Read("planslamscanning","");

/* PLAN Nav */

wxString plannavstop = fireball_taskname.Read("plannavstop", "");

wxString plannavexplore = fireball_taskname.Read("plannavexplore",
"");

wxString plannavgoto = fireball_taskname.Read("plannavgoto", "");

wxString plannavwander = fireball_taskname.Read("plannavwander",
"");

/* PLAN Visio */

wxString planvisioprocess =
fireball_taskname.Read("planvisioprocess", "");

wxString planvisioidle
```

```
=fireball_taskname.Read("planvisioidle","");
```

```
/* PLAN Casella Nera */

wxString plannone = fireball_taskname.Read("plannone", "");
```

Il tipo di task è specificato dal parametro type; la durata invece è specificata dal parametro duration. Se esso non viene impostato il suo valore di default è **STDDURATION**.

```
/* Lettura dei colori da file di testo */
        wxColour green = fireball_taskname.Read("green", "");
        wxColour orange = fireball_taskname.Read("orange", "");
        wxColour aquamarine = fireball_taskname.Read("aquamarine", "");
        wxColour pink = fireball_taskname.Read("pink", "");
        wxColour purple = fireball_taskname.Read("purple", "");
```

Al suo interno vengono richiamati tutti i casi relativi ai vari task delle tre barre di Navigazione, Slam, Visione. Questa funzione permette di aggiungere dunque task pianificati, evidenziandoli con colori differenti rispetto a quelli di AddTask().

```
cswitch(type){
            /* PLAN SLAM */
        case SLAMRUN: // PLAN SlamRun
            SetCellBackgroundColour(0, currentcell+planoffset, green);
            datagrid->SetCellValue(currentcell+planoffset,NAME,
            _T(planslamrunning));
            msg.Printf("%d",SLAMRUN);
```

```
        datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
        msg.Printf("PLAN Running %d ticks",duration);
        SetCellValue(0,currentcell+planoffset,msg);
        break;
    case SLAMSTOP: // PLAN SlamStop
        SetCellBackgroundColour(0, currentcell+planoffset, orange);
        datagrid->SetCellValue(currentcell+planoffset,NAME,
        _T(planslamstop));
        msg.Printf("%d",SLAMSTOP);
        datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
        msg.Printf("PLAN Stop %d ticks",duration);
        SetCellValue(0,currentcell+planoffset,msg);
        break;
    case SLAMSCAN: // PLAN SlamScan
        SetCellBackgroundColour(0, currentcell+planoffset,
        aquamarine);
        datagrid->SetCellValue(currentcell+planoffset,NAME,
        _T(planslamscanning));
        msg.Printf("%d",SLAMSCAN);
        datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
        msg.Printf("PLAN Scan %d ticks",duration);
        SetCellValue(0,currentcell+planoffset,msg);
        break;
```

9.12.1 GetPlanTaskString()

```
    /* BigGrid::GetPlanTaskString(long position,wxString &outstring) */
/* Costruisce una stringa con il nome e la durata del task
```

```
pianificato */ bool BigGrid::GetPlanTaskString(long
position,wxString &outstring){

    if(position<=planoffset){
        outstring.Printf("%s %s",datagrid->GetCellValue(
        position+currentcell,NAME),
            datagrid->GetCellValue(position+currentcell,DIMENSION));
        return true;
    }
    else{
        outstring.Printf("");
        return false;
    }
}
```

Ha il compito di costruire una stringa che contenga il nome e la durata del task pianificato. Contiene il parametro position: esso specifica l'offset relativo al task in esecuzione. Il parametro outstring è memorizzata la stringa ottenuta, uscente. La funzione restituisce true se nella posizione corrente si trova un task, altrimenti restituisce una stringa vuota e un valore false.

9.12.2 DeletePlan()

```
    /* BigGrid::DeletePlan(): Funzione inversa della precedente AddPlan() */
void BigGrid::DeletePlan(){

    long j=0;
    ((BigPanel*)GetParent())->LockBar();
    for(long i=0;i<planoffset;i++){
```

```
        if(datagrid->GetCellValue(currentcell+1+j,FLAG)==wxEmptyString){
            datagrid->DeleteRows(currentcell+1+j);
            DeleteCols(currentcell+1+j);
        }
        else{
            j++;
        }
    }
    planoffset=j;
    ((BigPanel*)GetParent())->UnlockBar();
}
```

Questa funzione è stata aggiunta per consentire l'eliminazione del piano corrente, l'accesso esclusivo ai dati della barra viene controllato dalla chiamata alla funzione LockBar() della classe BigPanel(). Non viene eliminato, controllando il valore del campo FLAG della 'datagrid', il task che si trova, *eventualmente*, in fase di modifica.

9.13 EndTask()

```
/* BigGrid::EndTask(int currenttime) */ void BigGrid::EndTask(int
currenttime){
    if(currenttime>0){
        if(currentact!=NONE){
            wxString msg;
            msg.Printf("%d",currenttime);
            datagrid->SetCellValue(currentcell,TALE,msg);
            msg.Printf("%s   %s",datagrid->GetCellValue(currentcell,HEAD),
```

```
            datagrid->GetCellValue(currentcell,NAME));
            SetCellValue(0,currentcell,msg);
            AddTask(currenttime);
        }
    }
}
```

Questa funzione calcola la fine del task, mettendolo in coda al task. Alla fine richiama la funzione *AddTask()* per il task successivo.

9.13.1 LockNewPlan()

Le funzioni LockNewPlan e AllockNewPlan servono a bloccare e a sbloccare il semaforo newplansemaphore. Questo permette un accesso esclusivo alle funzioni che accedono all'array newplan.

```
void BigGrid::LockNewPlan(){

    newplan_semaphore->Wait();
}

void BigGrid::UnlockNewPlan(){

    newplan_semaphore->Post();
}
```

9.14 OnRightClick()

```
/* Tasto Destro. Questo lancia la Finestra per la Modifica */ void
```

```
BigGrid::OnRightClick(wxGridEvent &ev){

    long evcol;
    long currentplan;
    evcol=ev.GetCol();
        if(evcol>currentcell){
            if(((BigPanel*)GetParent())->LockModify()){//se non è in corso
            //un’altra modifica apro una finestra modifica
                wxString name;
                wxString dimstr;
                ((BigPanel*)GetParent())->LockBar();
                datagrid->SetCellValue(evcol,FLAG,"X");//setto il flag
                per bloccare la cancellazione
                name=datagrid->GetCellValue(evcol,NAME);
                dimstr=datagrid->GetCellValue(evcol,DIMENSION);
                ((BigPanel*)GetParent())->UnlockBar();
                currentplan=evcol-currentcell;
                /* Lancio della Finestra di Modifica RightTaskDlg */
                ///* Questo fa comparire la finestra al doppio click */
    //clkpopup=new LeftClickFrame(this,-1,label,data);
                righttaskdlg=new RightTaskDlg(this,currentplan,dimstr,
                name,GetId());
                righttaskdlg->Show();
                }
        }
}
```

Questa funzione è richiamata quando si clicca con il pulsante destro del mouse su una cella della barra dei task. Se la cella selezionata rappresenta un task viene aperta una nuova finestra, che gira in un thread separato dall'applicazione, contenente i dati relativi al task selezionato. All'apertura della finestra il campo "FLAG" del task corrispondente viene marcato con "X", in modo da impedirne l'eliminazione durante la modifica. Viene impegnato il semaforo modifysemaphore in modo da non permettere più di una modifica alla volta.

9.14.1 Tick()

```
void BigGrid::Tick(long tick){

    if(tick>0){
    wxString msg;
    long value;
    if(datagrid->GetCellValue(currentcell,DIMENSION).ToLong(&value)){
        value+=tick;
        msg.Printf("%d",value);
        datagrid->SetCellValue(currentcell,DIMENSION,msg);
        if(zoomfactor>=0){
            SetColSize(currentcell,value*COLSIZE*zoomfactor);
        }
    }
    MakeCellVisible(0,currentcell);
    ForceRefresh();
    Update();
    }
```

```
}
```

Aumenta la dimensione del task corrente di un numero di tick uguale a quello specificato nel parametro (il cui valore di default è 1), aumentando la dimensione della cella puntata da ”currentcell” e modificando il valore ’DIMENSION’ nella riga della ”datagrid” corrispondente.

9.15 Start()/ZoomIn()/ZoomOut()/Fit()

Queste funzioni permettono di comandare, tramite i relativi pulsanti, la barra dei task. Sono contenute in StartPanel(), ma appartengono tutte a BigPanel().

9.16 Il Piano

```
wxString sst = fireball_taskname.Read("sst", "");
    wxString srn = fireball_taskname.Read("srn", "");
    wxString scn = fireball_taskname.Read("scn", "");
    wxString nst = fireball_taskname.Read("nst", "");
    wxString nex = fireball_taskname.Read("nex", "");
    wxString ngt = fireball_taskname.Read("ngt", "");
    wxString nwd = fireball_taskname.Read("nwd", "");
    wxString vid = fireball_taskname.Read("vid", "");
    wxString vpr = fireball_taskname.Read("vpr", "");
    wxString s1[8],s2[8],s3[9],s4[11],s5[8],s6[8],s7[11];
    wxString msg;
    msg.Printf("%d",0+cur);
```

```
    s1[0]=msg; s1[1]=sst; s1[2]=nst; s1[3]=vid; s1[4]=""; s1[5]=sst; s1[6]="";
    msg.Printf("%d",40+cur);
    s2[0]=msg; s2[1]=srn; s2[2]=nex; s2[3]=vid; s2[4]=""; s2[5]=vid; s2[6]="";
    msg.Printf("%d",75+cur);
    s3[0]=msg; s3[1]=srn; s3[2]=ngt; s3[3]=vid; s3[4]=""; s3[5]=srn; s3[6]=nst;
    s3[7]="ssn 70"; s3[8]="";
    msg.Printf("%d",105+cur);
    s4[0]=msg; s4[1]=srn; s4[2]=nwd; s4[3]=vpr; s4[4]=""; s4[5]="";
    msg.Printf("%d",135+cur);
    s5[0]=msg; s5[1]=scn; s5[2]=""; s5[3]=vpr; s5[4]="";
    msg.Printf("%d",170+cur);
    s6[0]=msg; s6[1]=nwd; s6[2]=vid; s6[3]=""; s6[4]="";
    msg.Printf("%d",210+cur);
    s7[0]=msg; s7[1]=""; s7[2]="NavWander 60";
    s7[3]="NavStop 40"; s7[4]="NavGoto 50"; s7[5]="";
    cur+=250;
    ReadPlan(s1);
    ReadPlan(s2);
    ReadPlan(s3);
    ReadPlan(s4);
    ReadPlan(s5);
    ReadPlan(s6);
    ReadPlan(s7);
}
```

9.17 Le Macro

Tutte le funzioni sopra elencate funzionano, nell'ambiente C++ *wxWidgets*, tramite l'esecuzione di macro. Esse permettono l'esecuzione del frame principale, che contiene a sua volta i pannelli con i pulsanti di comando delle barre dei task, le barre dei task stesse, e le varie funzioni esaminate.

Capitolo 10

Conclusioni

Lo svolgimento di questa tesi ha avuto come scopo la realizzazione di un collegamento tra pianificatore e visualizzatore, per poter facilitare l'azione dell'operatore sul robot da esso comandato.

L'operatore infatti, inserendo delle azioni opportune tramite il menù del pianificatore, provoca la produzione da parte del pianificatore stesso di alcuni file di dati. Questi file di dati vengono elaborati dal visualizzatore, che, a sua volta, mostra l'effetto delle azioni inserite dall'operatore e gli effetti della navigazione del robot all'interno dell'ambiente.

Infatti, come è stato documentato nel corso della tesi, le difficoltà pratiche maggiori dell'operatore nascono dal riscontro immediato dell'azione del robot nell'ambiente. Questo problema può dipendere sia da una serie di fattori ambientali, determinati dalle condizioni di utilizzo del robot, nella scena del disastro, sia dalla percezione limitata degli operatori del robot, che, agendo da una posizione lontana, in cui il robot non è visibile, non riescono a rendersi subito conto dei suoi movimenti.

La struttura del visualizzatore permette inoltre di poter ricevere e mostrare i dati ricevuti dal pianificatore in modo indipendente dalla sua struttura, che

funziona come un'intelaiatura per le informazioni da visualizzare.

Il visualizzatore utilizzato insieme alla Gui di controllo, è dunque un'ulteriore strumento per quanto riguarda la manovrabilità del robot da parte degli operatori, e permette loro di agire in tempo opportuno, rispetto a eventuali correzioni e modifiche della sua rotta di navigazione.

Per quanto riguarda eventuali miglioramenti dell'intera architettura DO.RO., possiamo ipotizzare una ancora maggiore interazione tra l'operatore e il robot, con lo sviluppo di interfacce sempre più complete e pratiche che facilitino il compito dei soccorritori.

Capitolo 11

Appendice

11.1 Fireball.h

```
/* HEADERS */

#include "wx/wx.h" #include "wx/thread.h" #include "wx/menu.h"
#include "wx/statusbr.h" #include "wx/scrolbar.h" #include
"wx/colordlg.h" #include "wx/fontdlg.h" #include "wx/grid.h"
#include "wx/dialog.h" #include "wx/generic/gridctrl.h"

/* Headers per la Lettura da File di Testo */ #include "wx/string.h"
#include "wx/wfstream.h" #include "wx/config.h" #include
"wx/confbase.h" #include "wx/fileconf.h" #include "wx/msw/regconf.h"

/* COSTANTI */ /* Costante di moltiplicazione per la dimensione
delle colonne dei Task */ #define COLSIZE 1 /* Durata Standard di un
Task */ #define STDDURATION 30 /* Modifica di un task: sua
eliminazione */ #define ERASE -1
```

```
/* INDIRIZZI FINESTRE */ /* BigPanel */ enum BIG_PANEL{
    BarSlam,
    BarNav,
    BarVision,
}; /* StartPanel */ enum START_PANEL{
    ZOOMINBTN=wxID_HIGHEST+1000,
    ZOOMOUTBTN,
    ZOOMFITBTN,
    STARTBTN,
}; /* LeftClickFrame */ enum DOUBLE_CLICK_FRAME{ OKDBCLK, }; /*
ModPanel: soltanto visibile */ enum MOD_PANEL{
    ADDSLAMRUNBTN,
    ADDSLAMSTOPBTN,
    ADDSLAMSCANBTN,
    ADDNAVSTOPBTN,
    ADDNAVEXPLOREBTN,
    ADDNAVGOTOBTN,
    ADDNAVWANDERBTN,
    ADDVISIOIDLE,
    ADDVISIOPROC,
    LOCKSCROLLCHK,
    IGNOREAUTOCHK,
    NEW_PLAN,
    TRIMBTN
}; /* RightTaskDlg */ enum MODIFY_TASK_DLG{
    MODOKBTN,
```

```
    MODCANCBTN,
    MODDELBTN,
}; /* PlanGrid */ enum {
    NAME=0,
    HEAD,
    TALE,
    DIMENSION,
    TYPE,
    FLAG,
}; /* Codici di Stato dei Task: non ancora implemetati */ enum
TASK_STATUS{
    INITIALIZED=0,
    RUNNING,
    STOPPED,
    OFF,
    PAUSED,
}; /* Funzioni AddTask e AddPlan: non ancora iplementate */ enum
TASK_LIST{
    INIT,
    SLAMRUN,
    SLAMSTOP,
    SLAMSCAN,
    NAVSTOP,
    NAVEXPL,
    NAVGOTO,
    NAVWAND,
    VISIDLE,
```

```
    VISPROC,
    NONE
}; /* BigGrid */ /* Indici per i valori di dimensioni e offset
nell'array newplan, in ogni BigGrid*/ enum{
    DIM=0,
    OFFSET
};

/* Menu Bar */ enum MENU_BAR {
    File_About,
    File_Quit,
};

/* LeftClickFrame */ class LeftClickFrame:public wxFrame{ public:
    wxButton *okbtn;
    LeftClickFrame(wxWindow*parent, wxWindowID id,
    wxString& title,wxString& mydata);
    void OnOk(wxCommandEvent& event);
    DECLARE_EVENT_TABLE()
};

/* RightTaskDlg */ class RightTaskDlg:public wxFrame{ public:
    RightTaskDlg(wxWindow* parent,long offse,
        wxString dim,const wxString name, int bar_id);
    wxButton* okbtn,*cancbtn,*delbtn;
    wxTextCtrl* dimctrl;
    void OnOk();
```

```
    void OnCanc();
    void OnErase();
    long offset;
    long callingplan;
    DECLARE_EVENT_TABLE()
};

/* StartPanel */ class StartPanel:public wxPanel{ public:

    /* I pulsanti Start, ZoomIn, ZoomOut, Fit:
    la loro gestione è affidata alla classe BigPanel */
    wxButton *start,*zibtn,*zobtn,*zfbtn;
    StartPanel(wxWindow* parent);
}; /* BigGrid */ class BigGrid:public wxGrid{ public:

    wxGrid *datagrid;
    float zoomfactor;
    int currentcell;
    int currentact;
    int scrollpos;
    int planoffset;
    /* GoGrid(): inizializza la griglia*/
    void GoGrid(const wxString& name);
    /* RightTaskDlg Pannello di Modifica */
    RightTaskDlg *righttaskdlg;
    /* AddTask: aggiunge il task specificato dal parametro. */
    virtual void AddTask(int currenttime,int type);
```

```
    virtual void EndTask(int currenttime);
    void FitPanel();
    void Tick(long tick=1);
    BigGrid(wxWindow* parent, wxWindowID id);
    /* Aggiunge un task pianificato */
    void AddPlan(int type=NONE, int duration=STDDURATION);
    float GetZoomFactor();
    float GetFitZoomFactor();
    void SetZoomFactor(float zoomfact);
    /* Elimina un task pianificato */
    void DeletePlan();
    void OnRightClick(wxGridEvent &ev);
    long new_plan[2];
    bool GetPlanTaskString(long position,wxString &outstring);
    /* I semafori limitano l'accesso a risorse cndivise:
    //i task in questo caso. */
    wxSemaphore *newplan_semaphore;
    void LockNewPlan();
    void UnlockNewPlan();
    DECLARE_EVENT_TABLE()

};

/* BigPanel */ class BigPanel:public wxPanel{ public: /* 3 oggeti di
tipo BigGrid */
    BigGrid *slam,*nav,*visio;
    int currenttime;
```

```
int cur;
wxSizer *topsizer, *upsizer, *middlesize, *bottomsizer, *btnsizer ;
LeftClickFrame* clkpopup; //finestra popup
BigPanel(wxWindow* parent);
void OnZoomIn(wxCommandEvent& WXUNUSED(ev));
void OnZoomOut(wxCommandEvent& WXUNUSED(ev));
void OnZoomFit(wxCommandEvent& WXUNUSED(ev));
void OnStart(wxCommandEvent& WXUNUSED(ev));
void OnBarDBClick(wxGridEvent& ev);
/* ReadPlan: Nove Funzioni di Modifica dei Task */
/* Vengono richiamate da ReadPlan, a seconda di quanto legge
//nell'array plan[i] */
void AddSlamRunning();
void AddSlamStop();
void AddSlamScan();
void AddNavStop();
void AddNavExplore();
void AddNavGoto();
void AddNavWander();
void AddVisIdle();
void AddVisProcess();
/* AddTick: riceve da ReadPlan il tick calcolato */
void AddTick(long tick=1);
/* ReadPlan: legge il piano; ha come argomento l'array plan[],
//con i nomi dei task */
void ReadPlan(wxString plan[]);
/* ReadPlan fine */
```

```
    void OnSlamNewPlan(wxCommandEvent& WXUNUSED(event) );
    void OnNavNewPlan(wxCommandEvent& WXUNUSED(event) );
    void OnVisioNewPlan(wxCommandEvent& WXUNUSED(event) );
    void OnSlamNoChange(wxCommandEvent& WXUNUSED(event) );
    void OnNavNoChange(wxCommandEvent& WXUNUSED(event) );
    void OnVisioNoChange(wxCommandEvent& WXUNUSED(event) );
    wxString* GetSlamPlan();
    wxString* GetNavPlan();
    wxString* GetVisioPlan();
    /*ModPanel *modpanel;*/
    StartPanel *startpanel;
    /* TrimBars(): richiamata da OnTrim */
    void TrimBars(long trim);
    /* Semafori */
    wxSemaphore *modify_semaphore;
    wxSemaphore *flag_semaphore;
    /* LockModify e UnLockModify: comandano modify_semaphore */
    bool LockModify();
    void UnLockModify();
    /* LockBar e UnLockBar: comandano flag_semaphore */
    void LockBar();
    void UnlockBar();

    DECLARE_EVENT_TABLE()
};

/* Il frame: MyFrame */ class MyFrame : public wxFrame { public:
```

```
    BigPanel *bigpanel;
    MyFrame(wxFrame* parent);

    /* Eventi Menu MyFrame */

    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

    DECLARE_EVENT_TABLE()
};

/* L'applicazione: Fireball */ class Fireball : public wxApp {
public:
    MyFrame *myframe;
    bool OnInit();

};

/* MACRO */ DEFINE_EVENT_TYPE(EVENT_OK);
DEFINE_EVENT_TYPE(NOTCHANGED);

/* Macro Applicazione: Fireball */ IMPLEMENT_APP(Fireball) /* Macro
MyFrame */ BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(File_Quit,  MyFrame::OnQuit)
    EVT_MENU(File_About, MyFrame::OnAbout)
END_EVENT_TABLE() /* Macro Big Panel */ BEGIN_EVENT_TABLE(BigPanel,
```

```
wxPanel) EVT_BUTTON(ZOOMINBTN, BigPanel::OnZoomIn)
EVT_BUTTON(ZOOMOUTBTN, BigPanel::OnZoomOut) EVT_BUTTON(ZOOMFITBTN,
BigPanel::OnZoomFit) EVT_BUTTON(STARTBTN,BigPanel::OnStart)
EVT_GRID_CELL_LEFT_DCLICK(BigPanel::OnBarDBClick) END_EVENT_TABLE()
/* Macro BigGrid */ BEGIN_EVENT_TABLE(BigGrid,wxGrid)
EVT_GRID_CELL_RIGHT_DCLICK(BigGrid::OnRightClick) END_EVENT_TABLE()
/* Macro ModPanel */
//BEGIN_EVENT_TABLE(ModPanel,wxPanel)
////EVT_BUTTON(TRIMBTN, ModPanel::OnTrim) // Questa Funziona,
//tolto il commento, NON Funziona: dà un errore "static_cast"
//END_EVENT_TABLE()
/* Macro RightTaskDlg */ BEGIN_EVENT_TABLE(RightTaskDlg,wxFrame)
//EVT_BUTTON(MODOKBTN,RightTaskDlg::OnOk) // Pure queste
//EVT_BUTTON(MODCANCBTN,RightTaskDlg::OnCanc)
//EVT_BUTTON(MODDELBTN,RightTaskDlg::OnErase)
END_EVENT_TABLE() /* Macro LeftClickFrame */
BEGIN_EVENT_TABLE(LeftClickFrame,wxFrame)
EVT_BUTTON(OKDBCLK,LeftClickFrame::OnOk) END_EVENT_TABLE()
```

11.2 Fireball.cpp

```
#include "fireball.h"

/* L'applicazione: Fireball */ bool Fireball::OnInit(){
```

```
    myframe = new MyFrame(NULL);
    myframe->Show( true );
    return true;
} /* Lettura da File di Testo */ wxFileInputStream
stream("fireball_taskname.txt"); wxFileInputStream
stream2("fireball_color.txt"); wxFileInputStream
stream3("fireball_string.txt"); wxFileInputStream
stream4("fireball_serv.txt"); wxFileConfig
fireball_taskname(stream); wxFileConfig fireball_color(stream2);
wxFileConfig fireball_string(stream3); wxFileConfig
fireball_serv(stream4);


/* MyFrame: Costruttore del Frame Principale: contiene tutti i
pannelli */ MyFrame::MyFrame(wxFrame* parent)
:wxFrame(parent,-1,_T("Fireball"),
         wxPoint(0,0),
         wxSize(1024,500),
         wxDEFAULT_FRAME_STYLE){

    bigpanel = new BigPanel(this);

     /* Menu Bar */
     // Icona del Frame
     //SetIcon(wxIcon(sample_xpm));
     // Menu
     wxMenu *menuFile = new wxMenu;
```

```
    menuFile->Append(File_About, _T("&About"));
    // Separatore
    menuFile->AppendSeparator();
    menuFile->Append(File_Quit, _T("&Quit"));
    // Menu Bar
    wxMenuBar *menuBar = new wxMenuBar;
    menuBar->Append(menuFile, _T("&File"));
    // Attacco la Menu Bar al Frame
    SetMenuBar(menuBar);
    // Status Bar
    CreateStatusBar(1);
    SetStatusText( "Fabrizio Mosconi ©" );
    // Scroll Bar: wxVERTICAL, wxORIZONTAL,
    //SetScrollbar(wxVERTICAL, 1, 16, 50, 1);
} /* I 3 Pannelli: BigPanel, StartPanel, ModPanel */

/* BigPanel: Il Pannello Principale */ BigPanel::BigPanel(wxWindow*
parent) :wxPanel(parent,-1,
         wxPoint(0,0),
         wxSize(1024,768),
         wxDOUBLE_BORDER){

    /* Barre dei moduli di: SLAM, Navigazione, Visione */
    slam=new BigGrid(this,BarSlam);
    nav=new BigGrid(this,BarNav);
    visio=new BigGrid(this,BarVision);
    /* PianoBox: il Box di Pianificazione dei Task */
```

```
wxStaticBox* PianoBox=new wxStaticBox(this,-1,"PIANO");
/* I nomi dei tre moduli */
wxString SLAM = fireball_serv.Read("SLAM", "");
wxString Nav = fireball_serv.Read("Nav", "");
wxString Visio = fireball_serv.Read("Visio", "");
slam->GoGrid(_T(SLAM));
nav->GoGrid(_T(Nav));
visio->GoGrid(_T(Visio));
currenttime=0;
slam->AddTask(0,NONE);
nav->AddTask(0,NONE);
visio->AddTask(0,NONE);
/* StartPanel e ModPanel */
startpanel=new StartPanel(this);
/* Collocazione e Aspetto dei Pannelli StartPanel e ModPanel */
/* Nell'ordine: StartPanel, le Barre per la
//Pianificazione dei Task, ModPanel */
wxSizer* mainsizer=new wxBoxSizer(wxVERTICAL);
topsizer=new wxStaticBoxSizer(PianoBox,wxVERTICAL);
mainsizer->Add(startpanel,0,wxGROW);
topsizer->Add(slam,0,wxGROW|wxSHAPED);
topsizer->Add(10,10);
topsizer->Add(nav,0,wxGROW|wxSHAPED);
topsizer->Add(10,10);
topsizer->Add(visio,0,wxGROW|wxSHAPED);
mainsizer->Add(topsizer,0,wxGROW);
slam->ForceRefresh();
```

```
    nav->ForceRefresh();
    visio->ForceRefresh();
    SetAutoLayout( TRUE );
    SetSizer( mainsizer );
    mainsizer->Fit( this );
    mainsizer->SetSizeHints( this );
    /* I Semafori: contatori che limitano l'accesso a risorse condivise */
    modify_semaphore=new wxSemaphore(1,1);
    flag_semaphore=new wxSemaphore(1,1);
    cur=0;
} /* StartPanel: Costruttore del Pannello di Start */ /* Contiene il
pulsante di Start e i pulsanti per lo Zoom,
   il Fit dei Task */
StartPanel::StartPanel(wxWindow* parent)
:wxPanel(parent,-1,wxDefaultPosition,wxDefaultSize,wxDOUBLE_BORDER){

    /* StartBox: il box con il pulsante di Start, quelli di Zoom, di Fit */
    wxStaticBox *startBox=new wxStaticBox(this,-1,"Quadro Pulsanti");
    wxStaticBoxSizer* startsiz=new wxStaticBoxSizer(startBox,wxHORIZONTAL);
    /* Il box con il pulsante di Start */
    start=new wxButton(this,STARTBTN,"Start",wxDefaultPosition,
    wxSize(120,70));
    /* I bottoni di Zoom In, Zoom Out, Fit e Lock */
    zibtn=new wxButton(this,ZOOMINBTN,"Zoom In",
    wxDefaultPosition,wxSize(120,70));
    zobtn=new wxButton(this,ZOOMOUTBTN,"Zoom Out",wxDefaultPosition,
    wxSize(120,70));
```

```
    zfbtn=new wxButton(this,ZOOMFITBTN,"Fit",wxDefaultPosition,
    wxSize(120,70));
    /* I bottoni sono sistemati al loro posto, nei loro rispettivi box */
    startsiz->Add(30,15);
    startsiz->Add(start,0,wxGROW|wxSHAPED|wxALIGN_CENTER);
    startsiz->Add(30,15);
    startsiz->Add(zibtn,0,wxGROW|wxSHAPED|wxALIGN_CENTER);
    startsiz->Add(30,15);
    startsiz->Add(zobtn,0,wxGROW|wxSHAPED|wxALIGN_CENTER);
    startsiz->Add(30,15);
    startsiz->Add(zfbtn,0,wxGROW|wxSHAPED|wxALIGN_CENTER);
    SetAutoLayout( TRUE );
    SetSizer( startsiz );
    startsiz->Fit( this );
    startsiz->SetSizeHints( this );
}

/* A PARTIRE DA QUESTO PUNTO SONO DEFINITE TUTTE LE FUNZIONI DEI
VARI PANNELLI */

/* Eventi Menu MyFrame */ void MyFrame::OnQuit(wxCommandEvent&
WXUNUSED(event)) {
    // Chiude il Frame
    Close(true);
}

void MyFrame::OnAbout(wxCommandEvent& WXUNUSED(event)) {
```

```
    wxMessageBox(_T("Fireball 1.0\n(c) 2005 Fabrizio Mosconi"),
                _T("Fireball 1.0"), wxOK| wxICON_INFORMATION, this);
} /* Fine Eventi Menu MyFrame */

/* Eventi BigPanel */ /* BigPanel::OnBarDBClick(wxGridEvent& ev) */
/* Fornisce i dettagli del task selezionato al doppio clic del tasto
sinistro e li invia alla finestra che si apre di conseguenza. Per
ogni task clickato vengono forniti: inizio e fine del task, HEAD e
TAIL, la lunghezza, DIMENSION. */

void BigPanel::OnBarDBClick(wxGridEvent& ev){
    wxString slamprint = fireball_serv.Read("slamprint", "");
    wxString navprint = fireball_serv.Read("navprint", "");
    wxString visioprint = fireball_serv.Read("visioprint", "");
    long info;
    wxString data,label;
    LockBar();
    switch(ev.GetId()){
    case BarSlam:  // Barra di SLAM
        info=slam->GetGridCursorCol();
        label.Printf(slamprint); // etichetta della finestra popup
        data.Printf("\n  Task: %s\n  Start: %s\n  End: %s\n  Length: %s",
                    slam->datagrid->GetCellValue(info,NAME),
        slam->datagrid->GetCellValue(info,HEAD), // Start
        slam->datagrid->GetCellValue(info,TALE), // End
        slam->datagrid->GetCellValue(info,DIMENSION)); // Length
    break;
```

```
    case BarNav: // Barra di Navigazione
        info=nav->GetGridCursorCol();
        label.Printf(navprint); // etichetta della finestra popup
        data.Printf("\n  Task: %s\n  Start: %s\n  End: %s\n  Length: %s",
                    nav->datagrid->GetCellValue(info,NAME),
        nav->datagrid->GetCellValue(info,HEAD), // Start
        nav->datagrid->GetCellValue(info,TALE), // End
        nav->datagrid->GetCellValue(info,DIMENSION)); // Length
    break;
    case BarVision: // Barra di Visione
        info=visio->GetGridCursorCol();
        label.Printf(visioprint); // etichetta della finestra popup
        data.Printf("\n  Task: %s\n  Start: %s\n  End: %s\n  Length: %s",
                    visio->datagrid->GetCellValue(info,NAME),
        visio->datagrid->GetCellValue(info,HEAD), // Start
        visio->datagrid->GetCellValue(info,TALE), // End
        visio->datagrid->GetCellValue(info,DIMENSION)); // Length
    break;
    }
    UnlockBar();
    /* Questo fa comparire la finestra al doppio click */
    clkpopup=new LeftClickFrame(this,-1,label,data);
} // vedi infatti il codice appena seguente

/* LeftClickFrame::LeftClickFrame() */ /* Informazione sui Task */
LeftClickFrame::LeftClickFrame(wxWindow*parent,
                               wxWindowID id,
```

```
                                 wxString& title,
                              wxString& mydata):wxFrame(parent,id,title,
                              wxDefaultPosition, wxSize(600,600)){

    SetBackgroundColour(*wxLIGHT_GREY);
    wxStaticText* text=new wxStaticText(this,-1,mydata);
    text->SetBackgroundColour(*wxLIGHT_GREY);
    okbtn=new wxButton(this,OKDBCLK,"Ok");
    wxBoxSizer *vz=new wxBoxSizer(wxVERTICAL);
    wxBoxSizer *zz=new wxBoxSizer(wxHORIZONTAL);
    zz->Add(text,1);
    zz->Add(60,70);
    vz->Add(zz,0);
    vz->Add(okbtn,0,wxCENTRE);
    vz->Add(5,5);
    SetSizer(vz);
    vz->SetSizeHints(this);
    vz->Fit(this);
    Show();
} /* Eventi LeftClickFrame */ void
LeftClickFrame::OnOk(wxCommandEvent& WXUNUSED(event)) {
    // Chiude il Frame
    Close(true);
    // Distrugge i Frame alla chiusura del Frame Principale
    Destroy();
}
```

```
/*Fine Eventi per LeftClickFrame */

/* BigPanel::ReadPlan(wxString plan[]): legge il Piano */ void
BigPanel::ReadPlan(wxString plan[]){

    long arrivetick;
    int i=1;
    // booleani per slam, nav, visio
    bool newslamplan=false;
    bool newnavplan=false;
    bool newvisioplan=false;
    // leggo il tick corrente e aggiorno le barre di avanzamento
    plan[0].ToLong(&arrivetick); // conversione da stringa a long
    // aggiungo alle barre il numero di tick necessario...
    AddTick(arrivetick-currenttime);
    // ... e modifico il tempo corrente
    currenttime=arrivetick;
    // leggo le azioni fino alla prima interruzione (stringa vuota)...
    while(plan[i]!=wxEmptyString){ // stringa vuota
        // ... e aggiungo il task letto
        // il controllo è eseguito nella funzione chiamata

        /* Le Nove Funzioni di Aggiunta dei Task */
        if(plan[i]=="SlamRunning"){      AddSlamRunning();}
        else{ if(plan[i]=="SlamStop"){   AddSlamStop();}
        else{ if(plan[i]=="SlamScan"){   AddSlamScan();}
        else{ if(plan[i]=="NavStop"){    AddNavStop();}
```

```
    else{ if(plan[i]=="NavExplore"){ AddNavExplore();}
    else{ if(plan[i]=="NavGoto"){    AddNavGoto();}
    else{ if(plan[i]=="NavWander"){  AddNavWander();}
    else{ if(plan[i]=="VisIdle"){    AddVisIdle();}
    else{ if(plan[i]=="VisProcess"){ AddVisProcess();}
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    i++;
}
i++;//salto l'elemento che contiene l'interruzione
/* La serie dei Task Pianificati */
// Aggiungo le azioni pianificate
while(plan[i]!=wxEmptyString){
// le stringhe possono contenere sia il nome del task...
    if(plan[i].Contains("SlamRunning")){
        // ... sia la durata prevista
        long value;
        if(newslamplan==false) slam->DeletePlan();
        newslamplan=true;
        //verifico se e' specificata la durata
        if(plan[i].AfterFirst(' ').ToLong(&value)){
```

```
        slam->AddPlan(SLAMRUN,value);
    }
    else{ //altrimenti specifico la durata di default
        slam->AddPlan(SLAMRUN,STDDURATION);
    }
}
    // SlamScan
    else{ if(plan[i].Contains("SlamScan")){
    long value;
    if(newslamplan==false) slam->DeletePlan();
    newslamplan=true;
    if(plan[i].AfterFirst(' ').ToLong(&value)){
        slam->AddPlan(SLAMSCAN,value);
    }
    else{ slam->AddPlan(SLAMSCAN,STDDURATION); }
}
    // SlamStop
    else{ if(plan[i].Contains("SlamStop")){
    long value;
    if(newslamplan==false) slam->DeletePlan();
    newslamplan=true;
    if(plan[i].AfterFirst(' ').ToLong(&value)){
        slam->AddPlan(SLAMSTOP,value);
    }
    else{ slam->AddPlan(SLAMSTOP,STDDURATION); }
}
    // NavStop
```

```
    else{ if(plan[i].Contains("NavStop")){
    long value;
    if(newnavplan==false) nav->DeletePlan();
    newnavplan=true;
    if(plan[i].AfterFirst(' ').ToLong(&value)){
        nav->AddPlan(NAVSTOP,value);
    }
    else{ nav->AddPlan(NAVSTOP,STDDURATION); }
}
    // NavExplore
    else{ if(plan[i].Contains("NavExplore")){
    long value;
    if(newnavplan==false) nav->DeletePlan();
    newnavplan=true;
    if(plan[i].AfterFirst(' ').ToLong(&value)){
        nav->AddPlan(NAVEXPL,value);
    }
    else{ nav->AddPlan(NAVEXPL,STDDURATION); }
}
    // NavGoto
    else{ if(plan[i].Contains("NavGoto")){
    long value;
    if(newnavplan==false) nav->DeletePlan();
    newnavplan=true;
    if(plan[i].AfterFirst(' ').ToLong(&value)){
        nav->AddPlan(NAVGOTO,value);
    }
```

```
    else{ nav->AddPlan(NAVGOTO,STDDURATION); }
}
    // NavWander
    else{ if(plan[i].Contains("NavWander")){
    long value;
    if(newnavplan==false) nav->DeletePlan();
    newnavplan=true;
    if(plan[i].AfterFirst(' ').ToLong(&value)){
        nav->AddPlan(NAVWAND,value);
    }
    else{ nav->AddPlan(NAVWAND,STDDURATION); }
}
    // VisIdle
    else{ if(plan[i].Contains("VisIdle")){
    long value;
    if(newvisioplan==false) visio->DeletePlan();
    newvisioplan=true;
    if(plan[i].AfterFirst(' ').ToLong(&value)){
        visio->AddPlan(VISIDLE,value);
    }
    else{ visio->AddPlan(VISIDLE,STDDURATION); }
}
    // VisProcess
    else{ if(plan[i].Contains("VisProcess")){
    long value;
    if(newvisioplan==false) visio->DeletePlan();
    newvisioplan=true;
```

```
            if(plan[i].AfterFirst(' ').ToLong(&value)){
                visio->AddPlan(VISPROC,value);
            }
            else{ visio->AddPlan(VISPROC,STDDURATION); }
        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
        i++;// elemento successivo
    }
} /* BigPanel::LockModify() e BigPanel::UnLockModify() */ /*
Bloccano e sbloccano i semafori, permettendo una sola modifica alla
volta */ bool BigPanel::LockModify(){
    if(modify_semaphore->TryWait()!=wxSEMA_NO_ERROR ) return false;
    return true;
}

void BigPanel::UnLockModify(){
    modify_semaphore->Post();
}

/*BigPanel::LockBar() e BigPanel::UnlockBar() */ /* Bloccano e
```

```
sbloccano i semafori, impedendo l'intervento sui dati delle barre
dei task, e sull'interfaccia */ void BigPanel::LockBar(){
    flag_semaphore->Wait();
}

void BigPanel::UnlockBar(){
    flag_semaphore->Post();
} /* Funzioni di Aggiunta Task */ /* BigPanel::AddSlamRunning() */
void BigPanel::AddSlamRunning(){
    if(slam->currentact!=SLAMRUN){
        if(slam->currentact!=INIT){
            slam->EndTask(currenttime);
        }
        slam->AddTask(currenttime,SLAMRUN);
    }
} /* BigPanel::AddSlamScan() */ void BigPanel::AddSlamScan(){
    if(slam->currentact!=SLAMSCAN){
        if(slam->currentact!=INIT){
            slam->EndTask(currenttime);
        }
        slam->AddTask(currenttime,SLAMSCAN);
    }
} /* BigPanel::AddSlamStop() */ void BigPanel::AddSlamStop(){
    if(slam->currentact!=SLAMSTOP){
        if(slam->currentact!=INIT){
            slam->EndTask(currenttime);
        }
```

```
        slam->AddTask(currenttime,SLAMSTOP);
    }
} /* BigPanel::AddNavExplore() */ void BigPanel::AddNavExplore(){
    if  (nav->currentact!=NAVEXPL){
        if(nav->currentact!=INIT){
            nav->EndTask(currenttime);
        }
        nav->AddTask(currenttime,NAVEXPL);
    }
} /* BigPanel::AddNavGoto() */ void BigPanel::AddNavGoto(){
    if(nav->currentact!=NAVGOTO){
        if(nav->currentact!=INIT){
            nav->EndTask(currenttime);
        }
        nav->AddTask(currenttime,NAVGOTO);
    }
} /* BigPanel::AddNavStop()*/ void BigPanel::AddNavStop(){
    if(nav->currentact!=NAVSTOP){
        if(nav->currentact!=INIT){
            nav->EndTask(currenttime);
        }
        nav->AddTask(currenttime,NAVSTOP);
    }
} /* BigPanel::AddNavWander() */ void BigPanel::AddNavWander(){
    if(nav->currentact!=NAVWAND){
        if(nav->currentact!=INIT){
            nav->EndTask(currenttime);
```

```
        }
        nav->AddTask(currenttime,NAVWAND);
    }
} /* BigPanel::AddVisIdle() */ void BigPanel::AddVisIdle(){
    if(visio->currentact!=VISIDLE){
        if(visio->currentact!=INIT){
            visio->EndTask(currenttime);
        }
        visio->AddTask(currenttime,VISIDLE);
    }
} /* BigPanel::AddVisProcess() */ void BigPanel::AddVisProcess(){
    if(visio->currentact!=VISPROC){
        if(visio->currentact!=INIT){
            visio->EndTask(currenttime);
        }
        visio->AddTask(currenttime,VISPROC);
    }
}

/* BigPanel::AddTick */ void BigPanel::AddTick(long tick){
    nav->Tick(tick);
    visio->Tick(tick);
    slam->Tick(tick);
}

/* Funzioni che Restituiscono i Task */ /* BigPanel::GetSlamPlan()
*/ wxString* BigPanel::GetSlamPlan(){
```

```
    wxString *outarray=new wxString[slam->planoffset+1];
    for(long i=1;i<=slam->planoffset;i++){
            LockBar();
        slam->GetPlanTaskString(i,outarray[i-1]);
        UnlockBar();
    }
    outarray[slam->planoffset]=wxEmptyString;
    return outarray;
} /* BigPanel::GetNavPlan() */ wxString* BigPanel::GetNavPlan(){
    wxString* outarray=new wxString[nav->planoffset+1];
    for(long i=1;i<=nav->planoffset;i++){
        LockBar();
        nav->GetPlanTaskString(i,outarray[i-1]);
        UnlockBar();
    }
    outarray[nav->planoffset]=wxEmptyString;
    return outarray;
} /* BigPanel::GetVisioPlan() */ wxString* BigPanel::GetVisioPlan(){
    wxString* outarray=new wxString[visio->planoffset+1];
    for(long i=1;i<=visio->planoffset;i++){
        LockBar();
        visio->GetPlanTaskString(i,outarray[i-1]);
        UnlockBar();
    }
    outarray[visio->planoffset]=wxEmptyString;
    return outarray;
}
```

```
/* BigGrid: la Barra dei Task*/ BigGrid::BigGrid(wxWindow* parent,
wxWindowID id)
:wxGrid(parent,id,wxDefaultPosition,wxSize(1024,100)){

    SetRowMinimalAcceptableHeight(0);
    SetColMinimalAcceptableWidth(1);
    SetRowMinimalHeight(0,1);
    SetDefaultCellBackgroundColour(GetParent()->GetBackgroundColour());
    datagrid = new wxGrid( this,
                       -1,
                       wxPoint( 0, 0 ),
                       wxSize( 1, 1 ) );
    datagrid->CreateGrid( 1, 6 );
    datagrid->SetRowSize( 0, 0 );
    datagrid->Show(FALSE);
    datagrid->SetColFormatNumber(HEAD);
    datagrid->SetColFormatNumber(TALE);
    datagrid->SetColFormatNumber(DIMENSION);
    datagrid->SetColFormatNumber(TYPE);
    zoomfactor=1;
    currentcell=-1;
    datagrid->DeleteRows(0,1);
    currentact=NONE;
    planoffset=0;
    newplan_semaphore=new wxSemaphore(1,1);
}
```

```
/* GoGrid: Inizializza la Griglia */ void BigGrid::GoGrid(const
wxString& name){

    //Crea una griglia di 1 riga e 0 colonne
    CreateGrid(1,0);
    //Dimensione delle righe
    SetRowSize(0,80);
    //Impedisce al n
    DisableDragColSize();
    DisableDragRowSize();
    //Larghezza Etichetta Colonne
    SetColLabelSize(0);
    //Il nome del singolo modulo
    SetRowLabelValue(0,name);
    //Larghezza Etichetta del singolo modulo
    SetRowLabelSize(80);
} /* Lettura dei nomi dei task da file di testo */ /* SLAM */
wxString slamrunning = fireball_taskname.Read("slamrunning", "");
wxString slamstop = fireball_taskname.Read("slamstop", ""); wxString
slamscanning = fireball_taskname.Read("slamscanning", ""); /* Nav */
wxString navstopp = fireball_taskname.Read("navstopp", ""); wxString
navexplore = fireball_taskname.Read("navexplore", ""); wxString
navgoto = fireball_taskname.Read("navgoto", ""); wxString navwander
= fireball_taskname.Read("navwander", ""); /* Visio */ wxString
visioprocess = fireball_taskname.Read("visioprocess", ""); wxString
visioidle = fireball_taskname.Read("visioidle", ""); /* Casella Nera
```

```
*/ wxString none = fireball_taskname.Read("none", "");

/* BigGrid::AddTask(int currenttime,int type=NONE) */ /*Aggiunge un
Task alla Barra Corrente, specificandone il Tick Corrente e l'Id del
Tipo di Task*/ void BigGrid::AddTask(int currenttime,int type=NONE){
        if(currenttime!=0) ((BigPanel*)GetParent())->LockBar();
        currentcell++;
        InsertCols(currentcell);
        datagrid->InsertRows(currentcell);
        wxString msg;
        msg.Printf("%d",currenttime);
        datagrid->SetCellValue(currentcell,HEAD,msg);
        datagrid->SetCellValue(currentcell,DIMENSION,_T("0"));
        datagrid->SetCellValue(currentcell,TALE,_T("-1"));
        datagrid->SetCellValue(currentcell,FLAG,wxEmptyString);
        SetReadOnly( 0, currentcell );
        SetCellOverflow(0, currentcell, TRUE);
        SetColMinimalWidth(currentcell,0);
        SetColSize(currentcell,0);
        /* Lettura dei colori da file di testo */
        /* Valgono per tutte e tre le barre */
        wxColour red = fireball_color.Read("red", "");
        wxColour lightgrey = fireball_color.Read("lightgrey", "");
        wxColour blue = fireball_color.Read("blue", "");
        wxColour cyan = fireball_color.Read("cyan", "");
        wxColour black = fireball_color.Read("black", "");
        switch(type){
```

```
    /* TASK DI SLAM */
case SLAMRUN: // SlamRun
    SetCellBackgroundColour(0, currentcell, red);
    datagrid->SetCellValue(currentcell,NAME,_T(slamrunning));
    msg.Printf("%d",SLAMRUN);
    datagrid->SetCellValue(currentcell,TYPE,msg);
    msg.Printf("%d  Running",currenttime);
    SetCellValue(0,currentcell,msg);
    currentact=SLAMRUN;
    break;
case SLAMSTOP: // SlamStop
    SetCellBackgroundColour(0, currentcell, lightgrey);
    datagrid->SetCellValue(currentcell,NAME,_T(slamstop));
    msg.Printf("%d",SLAMSTOP);
    datagrid->SetCellValue(currentcell,TYPE,msg);
    msg.Printf("%d  Stop",currenttime);
    SetCellValue(0,currentcell,msg);
    currentact=SLAMSTOP;
    break;
case SLAMSCAN:// SlamScan
    SetCellBackgroundColour(0, currentcell, blue);
    datagrid->SetCellValue(currentcell,NAME,_T(slamscanning));
    msg.Printf("%d",SLAMSCAN);
    datagrid->SetCellValue(currentcell,TYPE,msg);
    msg.Printf("%d  Scan",currenttime);
    SetCellValue(0,currentcell,msg);
    currentact=SLAMSCAN;
```

```
    break;
    /* TASK DI NAVIGAZIONE */
case NAVSTOP: // NavStop
    SetCellBackgroundColour(0, currentcell, lightgrey);
    datagrid->SetCellValue(currentcell,NAME,_T(navstopp));
    msg.Printf("%d",NAVSTOP);
    datagrid->SetCellValue(currentcell,TYPE,msg);
    msg.Printf("%d  Stop",currenttime);
    SetCellValue(0,currentcell,msg);
    currentact=NAVSTOP;
    break;
case NAVEXPL: // NavExplore
    SetCellBackgroundColour(0, currentcell, blue);
    datagrid->SetCellValue(currentcell,NAME,_T(navexplore));
    msg.Printf("%d",NAVEXPL);
    datagrid->SetCellValue(currentcell,TYPE,msg);
    msg.Printf("%d  Explore",currenttime);
    SetCellValue(0,currentcell,msg);
    currentact=NAVEXPL;
    break;
case NAVGOTO: // NavGoto
    SetCellBackgroundColour(0, currentcell, red);
    datagrid->SetCellValue(currentcell,NAME,_T(navgoto));
    msg.Printf("%d",NAVGOTO);
    datagrid->SetCellValue(currentcell,TYPE,msg);
    msg.Printf("%d  Goto",currenttime);
    SetCellValue(0,currentcell,msg);
```

```
        currentact=NAVGOTO;
        break;
    case NAVWAND: // NavWand
        SetCellBackgroundColour(0, currentcell, cyan);
        datagrid->SetCellValue(currentcell,NAME,_T(navwander));
        msg.Printf("%d",NAVWAND);
        datagrid->SetCellValue(currentcell,TYPE,msg);
        msg.Printf("%d  Wander",currenttime);
        SetCellValue(0,currentcell,msg);
        currentact=NAVWAND;
        break;
        /* TASK DI VISIONE */
    case VISIDLE: // VisIdle
        SetCellBackgroundColour(0, currentcell, lightgrey);
        datagrid->SetCellValue(currentcell,NAME,_T(visioidle));
        msg.Printf("%d",VISIDLE);
        datagrid->SetCellValue(currentcell,TYPE,msg);
        msg.Printf("%d  Idle",currenttime);
        SetCellValue(0,currentcell,msg);
        currentact=VISIDLE;
        break;
    case VISPROC: //VisProcess
        SetCellBackgroundColour(0, currentcell, red);
        datagrid->SetCellValue(currentcell,NAME,_T(visioprocess));
        msg.Printf("%d",VISPROC);
        datagrid->SetCellValue(currentcell,TYPE,msg);
        msg.Printf("%d  Process",currenttime);
```

```
            SetCellValue(0,currentcell,msg);
            currentact=VISPROC;
            break;
            /* CASELLA NERA DI DEFAULT */
        default: // Default: Casella Nera quando non c'è alcun Task
            SetCellBackgroundColour(0, currentcell, black);
            datagrid->SetCellValue(currentcell,NAME,_T(none));
            msg.Printf("%d",NONE);
            datagrid->SetCellValue(currentcell,TYPE,msg);
            msg.Printf("%d  None",currenttime);
            SetCellValue(0,currentcell,msg);
            currentact=NONE;
            break;
        }
        if(currenttime!=0) ((BigPanel*)GetParent())->UnlockBar();
        ForceRefresh();
}

/* Lettura dei nomi dei Task Pianificati da file di testo */ /* PLAN
SLAM */ wxString planslamrunning =
fireball_taskname.Read("planslamrunning", ""); wxString planslamstop
= fireball_taskname.Read("planslamstop", ""); wxString
planslamscanning = fireball_taskname.Read("planslamscanning", "");
/* PLAN Nav */ wxString plannavstop =
fireball_taskname.Read("plannavstop", ""); wxString plannavexplore =
fireball_taskname.Read("plannavexplore", ""); wxString plannavgoto =
fireball_taskname.Read("plannavgoto", ""); wxString plannavwander =
```

```
fireball_taskname.Read("plannavwander", ""); /* PLAN Visio */
wxString planvisioprocess =
fireball_taskname.Read("planvisioprocess", ""); wxString
planvisioidle = fireball_taskname.Read("planvisioidle", ""); /* PLAN
Casella Nera */ wxString plannone =
fireball_taskname.Read("plannone", ""); /* BigGrid::AddPlan(int
type, int duration) */ /* Aggiunge un Piano */ void
BigGrid::AddPlan(int type, int duration){
        wxString msg;
        msg.Printf("%d",duration);
        ((BigPanel*)GetParent())->LockBar();
        planoffset++;
        AppendCols(1);
        datagrid->AppendRows(1);
        datagrid->SetCellValue(currentcell+planoffset,DIMENSION,msg);
        datagrid->SetCellValue(currentcell+planoffset,FLAG,
        wxEmptyString);
        SetColMinimalWidth(currentcell+planoffset,0);
        SetReadOnly( 0, currentcell+planoffset );
        SetCellOverflow(0, currentcell+planoffset, TRUE);
        SetColSize(currentcell+planoffset,
        (int)duration*zoomfactor*COLSIZE);
        /* Lettura dei colori da file di testo */
        wxColour green = fireball_color.Read("green", "");
        wxColour orange = fireball_color.Read("orange", "");
        wxColour aquamarine = fireball_color.Read("aquamarine", "");
        wxColour pink = fireball_color.Read("pink", "");
```

```
wxColour purple = fireball_color.Read("purple", "");
switch(type){
    /* PLAN SLAM */
case SLAMRUN: // PLAN SlamRun
    SetCellBackgroundColour(0, currentcell+planoffset, green);
    datagrid->SetCellValue(currentcell+planoffset,NAME,
    _T(planslamrunning));
    msg.Printf("%d",SLAMRUN);
    datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
    msg.Printf("PLAN Running %d ticks",duration);
    SetCellValue(0,currentcell+planoffset,msg);
    break;
case SLAMSTOP: // PLAN SlamStop
    SetCellBackgroundColour(0, currentcell+planoffset, orange);
    datagrid->SetCellValue(currentcell+planoffset,NAME,
    _T(planslamstop));
    msg.Printf("%d",SLAMSTOP);
    datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
    msg.Printf("PLAN Stop %d ticks",duration);
    SetCellValue(0,currentcell+planoffset,msg);
    break;
case SLAMSCAN: // PLAN SlamScan
    SetCellBackgroundColour(0, currentcell+planoffset, aquamarine);
    datagrid->SetCellValue(currentcell+planoffset,NAME,
    _T(planslamscanning));
    msg.Printf("%d",SLAMSCAN);
    datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
```

```
    msg.Printf("PLAN Scan %d ticks",duration);
    SetCellValue(0,currentcell+planoffset,msg);
    break;
    /* PLAN NAVIGAZIONE */
case NAVSTOP: // PLAN NavStop
    SetCellBackgroundColour(0, currentcell+planoffset, orange);
    datagrid->SetCellValue(currentcell+planoffset,NAME
    ,_T(plannavstop));
    msg.Printf("%d",NAVSTOP);
    datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
    msg.Printf("PLAN Stop %d ticks",duration);
    SetCellValue(0,currentcell+planoffset,msg);
    break;
case NAVEXPL: // PLAN NavExplorer
    SetCellBackgroundColour(0, currentcell+planoffset, aquamarine);
    datagrid->SetCellValue(currentcell+planoffset,NAME,
    _T(plannavexplore));
    msg.Printf("%d",NAVEXPL);
    datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
    msg.Printf("PLAN Explore %d ticks",duration);
    SetCellValue(0,currentcell+planoffset,msg);
    break;
case NAVGOTO: // PLAN NavGoto
    SetCellBackgroundColour(0, currentcell+planoffset, green);
    datagrid->SetCellValue(currentcell+planoffset,NAME,
    _T(plannavgoto));
    msg.Printf("%d",NAVGOTO);
```

```
        datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
        msg.Printf("PLAN Goto %d ticks",duration);
        SetCellValue(0,currentcell+planoffset,msg);
        break;
    case NAVWAND: // PLAN NavWand
        SetCellBackgroundColour(0, currentcell+planoffset, pink);
        datagrid->SetCellValue(currentcell+planoffset,NAME,
        _T(plannavwander));
        msg.Printf("%d",NAVWAND);
        datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
        msg.Printf("PLAN Wander %d ticks",duration);
        SetCellValue(0,currentcell+planoffset,msg);
        break;
        /* PLAN VISIONE */
    case VISIDLE: // PLAN VisIdle
        SetCellBackgroundColour(0, currentcell+planoffset, orange);
        datagrid->SetCellValue(currentcell+planoffset,NAME,
        _T(planvisioidle));
        msg.Printf("%d",VISIDLE);
        datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
        msg.Printf("PLAN Idle %d ticks",duration);
        SetCellValue(0,currentcell+planoffset,msg);
        break;
    case VISPROC: // PLAN VisProcess
        SetCellBackgroundColour(0, currentcell+planoffset, green);
        datagrid->SetCellValue(currentcell+planoffset,NAME,
        _T(planvisioprocess));
```

```
            msg.Printf("%d",VISPROC);
            datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
            msg.Printf("PLAN Process %d ticks",duration);
            SetCellValue(0,currentcell+planoffset,msg);
            break;
        default: // Default: Casella Nera quando non c'è alcun Task
            SetCellBackgroundColour(0, currentcell+planoffset, purple);
            datagrid->SetCellValue(currentcell+planoffset,NAME,
            _T(plannone));
            msg.Printf("%d",NONE);
            datagrid->SetCellValue(currentcell+planoffset,TYPE,msg);
            msg.Printf("PLAN None %d ticks",duration);
            SetCellValue(0,currentcell+planoffset,msg);
            break;
        }
        ((BigPanel*)GetParent())->UnlockBar();
        ForceRefresh();
}

/* BigGrid::GetPlanTaskString(long position,wxString &outstring) */
/* Costruisce una stringa con il nome e la durata del task
pianificato */ bool BigGrid::GetPlanTaskString(long
position,wxString &outstring){

    if(position<=planoffset){
        outstring.Printf("%s %s",datagrid->
        GetCellValue(position+currentcell,NAME),
```

```
            datagrid->GetCellValue(position+currentcell,DIMENSION));
        return true;
    }
    else{
        outstring.Printf("");
        return false;
    }
}

/* BigGrid::DeletePlan(): Funzione inversa della precedente
AddPlan() */ void BigGrid::DeletePlan(){

    long j=0;
    ((BigPanel*)GetParent())->LockBar();
    for(long i=0;i<planoffset;i++){
        if(datagrid->GetCellValue(currentcell+1+j,
        FLAG)==wxEmptyString){
            datagrid->DeleteRows(currentcell+1+j);
            DeleteCols(currentcell+1+j);
        }
        else{
            j++;
        }
    }
    planoffset=j;
    ((BigPanel*)GetParent())->UnlockBar();
}
```

```
/* BigGrid::EndTask(int currenttime) */ void BigGrid::EndTask(int
currenttime){
    if(currenttime>0){
        if(currentact!=NONE){
            wxString msg;
            msg.Printf("%d",currenttime);
            datagrid->SetCellValue(currentcell,TALE,msg);
            msg.Printf("%s   %s",datagrid->
            GetCellValue(currentcell,HEAD),
            datagrid->GetCellValue(currentcell,NAME));
            SetCellValue(0,currentcell,msg);
            AddTask(currenttime);
        }
    }
}

void BigGrid::LockNewPlan(){

    newplan_semaphore->Wait();
}

void BigGrid::UnlockNewPlan(){

    newplan_semaphore->Post();
} /* Tasto Destro. Questo lancia la Finestra per la Modifica */ void
BigGrid::OnRightClick(wxGridEvent &ev){
```

```
    long evcol;
    long currentplan;
    evcol=ev.GetCol();
        if(evcol>currentcell){
            if(((BigPanel*)GetParent())->LockModify())
            {//se non è in corso
            //un'altra modifica apro una finestra modifica
                wxString name;
                wxString dimstr;
                ((BigPanel*)GetParent())->LockBar();
                datagrid->SetCellValue(evcol,FLAG,"X");
                //setto il flag per bloccare la cancellazione
                name=datagrid->GetCellValue(evcol,NAME);
                dimstr=datagrid->GetCellValue(evcol,DIMENSION);
                ((BigPanel*)GetParent())->UnlockBar();
                currentplan=evcol-currentcell;
                /* Lancio della Finestra di Modifica RightTaskDlg */
                ///* Questo fa comparire la finestra al doppio click */
    //clkpopup=new LeftClickFrame(this,-1,label,data);
                righttaskdlg=new RightTaskDlg(this,
                currentplan,dimstr,name,GetId());
                righttaskdlg->Show();
                }
        }
}
```

```
void BigGrid::Tick(long tick){

    if(tick>0){
    wxString msg;
    long value;
    if(datagrid->GetCellValue(currentcell,DIMENSION).ToLong(&value)){
        value+=tick;
        msg.Printf("%d",value);
        datagrid->SetCellValue(currentcell,DIMENSION,msg);
        if(zoomfactor>=0){
            SetColSize(currentcell,value*COLSIZE*zoomfactor);
        }
    }
    MakeCellVisible(0,currentcell);
    ForceRefresh();
    Update();
    }
} /* Eventi BigPanel per il pannello StartPanel */ /*
BigPanel::ZoomIn() */ /* Pulsante di ZoomIn per i task */ void
BigPanel::OnZoomIn(wxCommandEvent& WXUNUSED(ev)){

    float newzoom;
    newzoom=slam->GetZoomFactor();
    newzoom*=2;
    slam->SetZoomFactor(newzoom);
    nav->SetZoomFactor(newzoom);
    visio->SetZoomFactor(newzoom);
```

```
} /* BigGrid::GetZoomFactor() */ float BigGrid::GetZoomFactor(){
    return zoomfactor;
} /* BigGrid::GetFitZoomFactor() */ float
BigGrid::GetFitZoomFactor(){
    int width,height;
    float temp;
    int i=0;
    long value,tot;
    GetSize(&width,&height);
    i=GetNumberCols();
    tot=0;
    for(int m=0;m<i;m++){
        datagrid->GetCellValue(m,DIMENSION).ToLong(&value);
        tot+=value;
    }
    temp=(((float)width-70.0)/(float)tot)/(float)COLSIZE;
    return temp;
} /* BigGrid::SetZoomFactor() */ void BigGrid::SetZoomFactor(float
zoomfact){

    int i=GetNumberCols();
    long value;
    for(int j=0;j<i;j++){
        datagrid->GetCellValue(j,DIMENSION).ToLong(&value);
        SetColSize(j,(int)floor(value*COLSIZE*zoomfact));
    }
    zoomfactor=zoomfact;
```

```
    ForceRefresh();
}

/* BigPanel::OnZoomOut() */ /* Pulsante di ZoomOut per i task */
void BigPanel::OnZoomOut(wxCommandEvent& WXUNUSED(ev)){

    float newzoom;
    newzoom=slam->GetZoomFactor();
    newzoom/=2;
    slam->SetZoomFactor(newzoom);
    nav->SetZoomFactor(newzoom);
    visio->SetZoomFactor(newzoom);
} /* BigPanel::OnStart() */ /* Pulsante di Start per i task */ void
BigPanel::OnStart(wxCommandEvent& WXUNUSED(ev)){

    wxString var1 = fireball_string.Read("var1", "");
    wxString var2 = fireball_string.Read("var2", "");
    wxString var3 = fireball_string.Read("var3", "");
    wxString var4 = fireball_string.Read("var4", "");
    wxString var5 = fireball_string.Read("var5", "");
    wxString var6 = fireball_string.Read("var6", "");
    wxString var7 = fireball_string.Read("var7", "");
    wxString var8 = fireball_string.Read("var8", "");
    wxString var9 = fireball_string.Read("var9", "");
    wxString plssc = fireball_string.Read("plssc", "");
    wxString plnwd = fireball_string.Read("plnwd", "");
    wxString plnst = fireball_string.Read("plnst", "");
```

```
wxString plngt = fireball_string.Read("plngt", "");
/*wxString qua = fireball_string.Read("qua", "");
long value;*/
/* I Piani sotto forma di stringhe */
wxString s1[7],s2[7],s3[9],s4[6],s5[5],s6[5],s7[6];
wxString msg;
msg.Printf("%d",0+cur);
//s1[7]
s1[0]=msg;
s1[1]=var1;
s1[2]=var4;
s1[3]=var8;
s1[4]="";
s1[5]=var1;
s1[6]="";
msg.Printf("%d", /*qua.ToLong(&value)*/40+cur);
//s2[7]
s2[0]=msg;
s2[1]=var2;
s2[2]=var5;
s2[3]=var8;
s2[4]="";
s2[5]=var8;
s2[6]="";
msg.Printf("%d",75+cur);
//s3[9]
s3[0]=msg;
```

```
s3[1]=var2;
s3[2]=var6;
s3[3]=var8;
s3[4]="";
s3[5]=var2;
s3[6]=var4;
s3[7]=plssc;
s3[8]="";
msg.Printf("%d",105+cur);
//s4[6]
s4[0]=msg;
s4[1]=var2;
s4[2]=var7;
s4[3]=var9;
s4[4]="";
s4[5]="";
msg.Printf("%d",135+cur);
//s5[5]
s5[0]=msg;
s5[1]=var3;
s5[2]="";
s5[3]=var9;
s5[4]="";
msg.Printf("%d",170+cur);
//s6[5]
s6[0]=msg;
s6[1]=var7;
```

```
    s6[2]=var8;
    s6[3]="";
    s6[4]="";
    msg.Printf("%d",210+cur);
    //s7[6]
    s7[0]=msg;
    s7[1]="";
    s7[2]=plnwd;
    s7[3]=plnst;
    s7[4]=plngt;
    s7[5]="";
    cur+=250; //incremento del valore corrente
    ReadPlan(s1);
    ReadPlan(s2);
    ReadPlan(s3);
    ReadPlan(s4);
    ReadPlan(s5);
    ReadPlan(s6);
    ReadPlan(s7);
} /* BigPanel::OnZoomIn() */ /* Pulsante di OnZoomFit per i task */
void BigPanel::OnZoomFit(wxCommandEvent& WXUNUSED(ev)){

    float oldzoom,newzoom;
    oldzoom=slam->GetFitZoomFactor();
    newzoom=nav->GetFitZoomFactor();
    if(newzoom<oldzoom) oldzoom=newzoom;
    newzoom=visio->GetFitZoomFactor();
```

```
    if (newzoom<oldzoom) oldzoom=newzoom;
    slam->SetZoomFactor(oldzoom);
    nav->SetZoomFactor(oldzoom);
    visio->SetZoomFactor(oldzoom);
} /* Fine Eventi per i pulsanti del pannello StartPanel */

/* Qua c'è la finestra di modifica: tasto destro */ /* E' lanciata
da OnRightClick() */ RightTaskDlg::RightTaskDlg(wxWindow*
parent,long offse,wxString dim,const wxString name, int bar_id):
        wxFrame(parent,-1,"Modifica dimensione"){
            wxPanel *panel=new wxPanel(this,-1);
    offset=offse;
    callingplan=bar_id;
    /* Parte Grafica */
    wxSizer *topsizer=new wxBoxSizer(wxVERTICAL);
    okbtn=new wxButton(panel,MODOKBTN,"Modify");
    cancbtn=new wxButton(panel,MODCANCBTN,"Cancel");
    delbtn=new wxButton(panel,MODDELBTN,"Delete");
    wxStaticText* static_1=new wxStaticText(panel,-1,"Dimensione:  ");
    wxStaticText* static_2=new wxStaticText(panel,-1," ticks");
    dimctrl=new wxTextCtrl(panel,-1,dim, wxDefaultPosition,wxSize(50,18));
    wxSizer* textsizer=new wxBoxSizer(wxHORIZONTAL);
    wxString msg;
    msg="Task: "+name;
    wxStaticText *staticname=new wxStaticText(panel,-1,msg);
    switch(bar_id){
    case BarSlam:
```

```
    msg.Printf("Modulo: Slam");
    break;
case BarNav:
    msg.Printf("Modulo: Nav");
    break;
case BarVision:
    msg.Printf("Modulo: Visio");
    break;
}
wxStaticText *staticmodule=new wxStaticText(panel,-1,msg);
wxSizer* btnsiz=new wxBoxSizer(wxHORIZONTAL);
btnsiz->Add(okbtn,0);
btnsiz->Add(10,10);
btnsiz->Add(delbtn,0);
btnsiz->Add(10,10);
btnsiz->Add(cancbtn,0);
textsizer->Add(10,10);
textsizer->Add(static_1,0);
textsizer->Add(dimctrl,0);
textsizer->Add(static_2,0);
textsizer->Add(20,20);
topsizer->Add(10,10);
topsizer->Add(staticmodule,0,wxCENTRE);
topsizer->Add(8,8);
topsizer->Add(staticname,0,wxCENTRE);
topsizer->Add(8,8);
topsizer->Add(textsizer,0,wxCENTRE);
```

```
    topsizer->Add(btnsiz,0,wxCENTRE|wxALL,20);
    SetAutoLayout(TRUE);
    panel->SetSizer(topsizer);
    topsizer->Fit(panel);
    topsizer->SetSizeHints(this);
} /* Eventi RightTaskDialog */ void RightTaskDlg::OnOk(){

    long value;
    ((BigGrid*)GetParent())->LockNewPlan();
    if(dimctrl->GetValue().ToLong(&value)){
        if(value>0){

            ((BigGrid*)GetParent())->new_plan[OFFSET]=offset;
            ((BigGrid*)GetParent())->new_plan[DIM]=value;
            ((BigGrid*)GetParent())->UnlockNewPlan();
            wxCommandEvent newpevent(EVENT_OK,callingplan);
            newpevent.SetInt(-1);
            wxPostEvent((BigPanel*)(GetParent()->GetParent()),newpevent);
            Close();
        }
        else{
            ((BigGrid*)GetParent())->UnlockNewPlan();
        }
    }
    else{
        ((BigGrid*)GetParent())->UnlockNewPlan();
    }
```

```
}

void RightTaskDlg::OnErase(){

    ((BigGrid*)GetParent())->LockNewPlan();
    ((BigGrid*)GetParent())->new_plan[OFFSET]=offset;
    ((BigGrid*)GetParent())->new_plan[DIM]=ERASE;
    ((BigGrid*)GetParent())->UnlockNewPlan();
    wxCommandEvent newpevent(EVENT_OK,callingplan);
    newpevent.SetInt(-1);
    wxPostEvent((BigPanel*)(GetParent()->GetParent()),newpevent);
    Close();
}

void RightTaskDlg::OnCanc(){

    ((BigGrid*)GetParent())->LockNewPlan();
    ((BigGrid*)GetParent())->new_plan[OFFSET]=offset;
    ((BigGrid*)GetParent())->new_plan[DIM]=ERASE;
    ((BigGrid*)GetParent())->UnlockNewPlan();
        wxCommandEvent newpevent(NOTCHANGED,callingplan);
        newpevent.SetInt(-1);
        wxPostEvent((BigPanel*)(GetParent()->GetParent()),newpevent);
        Close();
}
```

Capitolo 12

Ringraziamenti

”*Il sempre sospirar nulla rileva*”, Francesco Petrarca

”*Mai lasciare in frigorifero roba che va a male.*”, Spike Spiegel

Ringrazio Dio per avermi messo alla prova e per avermi aiutato a superare la prova, e anche per avermi dato una buona testa e delle buone gambe, e Maria, ”*d'ogni fedel nocchier, sicura guida*”.

Ringrazio Andrea Orlandini e la prof. Limongelli, per l'incarico assegnatomi, la fiducia accordatami e soprattutto per la pazienza dimostrata, e ce n'è voluta tanta! Buon lavoro!

Ringrazio i miei genitori. Mamma per avermi trasmesso l'amore per la lettura e la scrittura, e papà, per avermi dato un po' della sua grande passione scientifica.

Ringrazio mio fratello Federico, a cui dedico un ”*Dattebayo!*” (KWGoD!). Sei diventato l'incarnazione del perfetto *otaku*! Spero che *Naruto* e *Bleach* finiscano presto, perhé sono stanco di usare il mio computer per scaricare tutte quelle boiate! E' ora di dedicarsi a letture decisamente più serie (*One Piece*).

Ringrazio Palma, la mia *white queen*, per l'amore e la pazienza donati, e per avermi insegnato che non si cammina e non si vive sempre da soli.

Ringrazio Alessandro, un vero amico, anche se per gli amici non servono aggettivi. Love & Peace! (Immaginalo mentre incrocio le dita, nella classica posa da idiota)

Ringrazio don Martino e a don Carlo per le loro parole di saggezza e conforto.

Ringrazio Sir Robert Baden Powell, il fondatore degli Scout: "*estote parati!*".

Ringrazio l'Inter, perché dopo tanto tempo ha vinto uno scudetto sul campo, alla faccia di juventini, milanisti, romanisti e di tutti i maledetti gufi. Citando i Queen - un grazie anche a loro, per la loro musica - in quel di Wembley '86, "*They're talking from here!*". Non sono state sempre rose e fiori, ma "*We are the champions of the world!*".

Un saluto, infine, a tutti i colleghi e amici del Vecchio Ordinamento. Coraggio, brutti bastardi, la strada è stata ed è lunga, ma se ce l'ho fatta io, potete farcela anche voi!

Bibliografia

[1] Robin R. Murphy, *Rescue Robotics for Homeland Security*, 2004.

[2] Robin R. Murphy, *Human-Robot Interaction in Rescue Robotics*, 2004.

[3] *Pianificazione Prolog* Corso di Intelligenza Artificiale, Università Ferrara.

[4] G. Gini, *Navigazione e Mappe per Robot Mobili a Ruote*, Politecnico Milano, Anno Accademico 2002/2003.

[5] G. Gini, *Modello Cognitivo e Pianificazione*, Politecnico Milano, Anno Accademico 2003/2004.

[6] M. Cialdea, *Dispense, Dia Roma3*, Università Roma 3, Dipartimento di Informatica e Automazione, Anno Accademico 2000/2001.

[7] Valentina Ventriglia *Sistema di Controllo per Robot con Iniziativa Mista Operatore-Sistema di Controllo Autonomo*, Università Roma 3, Dipartimento di Informatica e Automazione, Anno Accademico 2004/2005.

[8] M. W. M. Gamini Dissanayake, Paul Newman, Steven Clark, M. Csorba, *A Solution to the Simultaneous Localization and Map Building (SLAM) Problem*, 2001.

[9] Stefano Panzieri, Federica Pascucci, Roberto Setola, *Simultaneous Localization and Map Building Algorithm for Real-Time Applications*, 2003.

[10] Greg Welch, Gary Bishop, *An Introduction to the Kalman Filter*, 2006.

12.1 Codice

[11] wxWidgets Homepage,
http://www.wxwidgets.org/.

[12] wxWidgets Discussion Forum,
http://wxforum.shadonet.com/.

[13] wxWidgets version 2.6.2 Docs, Samples, Manual.

[14] Aria Demo 2.4-1, ActivMedia Robotics, LLC, Docs, Samples, Manual.