



SAPIENZA
UNIVERSITÀ DI ROMA

Progetto Robocup: sviluppo del comportamento di un robot difensore intercettatore

Facoltà di Ingegneria dell'informazione, informatica e statistica
Corso di Ingegneria informatica

Giacomo Jin
Matricola 1937721

Relatore
Prof. Thomas Alessandro Ciarfuglia

Anno Accademico 2023/2024

Tesi non ancora discussa

Progetto Robocup: sviluppo del comportamento di un robot difensore intercettatore

Sapienza Università di Roma

© 2024 Giacomo Jin. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: jin.1937721@studenti.uniroma1.it

Sommario

La RoboCup è una competizione annuale di robotica il cui obiettivo è promuovere la ricerca in robotica e intelligenza artificiale (IA) offrendo una sfida impegnativa e di grande interesse al pubblico. 2 squadre di robottini, 7 giocatori a squadra, che si muovono in modo autonomo si spostano in un campo di calcio grande 75 metri quadrati. Il compito nostro da programmatore è di codificare un framework abbastanza solido e robusto affinchè questi robot riescano autonomamente a giocare a calcio come se fossero esseri umani. Programmare i robot in tal modo presenta varie sfide tecniche: i robot devono essere in grado di muoversi in modo efficiente e stabile sul campo di gioco, devono essere in grado di eseguire azioni come calciare la palla e coordinarsi con i compagni di squadra, devono essere in grado di gestire situazioni impreviste e continuare a funzionare in modo affidabile, e soprattutto devono essere in grado di percepire l'ambiente circostante, riconoscere gli oggetti e prendere decisioni in tempo reale.

L'architettura creata dal team di robocup B-Human ha giocato un ruolo cruciale nell'affrontare diverse sfide nella RoboCup. La sua natura modulare ha permesso al team di evolvere e migliorare il sistema senza la necessità di riscrivere l'intero codice. Oltre a ciò, il team B-Human condivide il suo codice in modo open source dopo ogni anno di RoboCup, permettendo così ad altri team di trarne vantaggio. Tutto ciò ha permesso al team B-Human di vincere in totale il titolo di campioni del mondo per nove volte.

Anche la nostra università ha una squadra di ricerca coinvolta nella Robocup, ossia SPQR. La squadra SPQR è la squadra RoboCup del Dipartimento di Ingegneria Informatica, Automatica presso l'Università Sapienza di Roma, e ha partecipato a numerosi eventi, tra cui la Maker Faire a Roma, la RoboCup e la RomeCup. L'architettura del codice del team SPQR attualmente si basa principalmente sul codice rilasciato da B-Human nel 2021, per questo spiegheremo in seguito l'architettura B-Human per poi proseguire con la spiegazione del compito assegnatomi nella partecipazione dello sviluppo del codice. Più in specifico il task che mi è stato assegnato è lo sviluppo di un behavior per il robot calciatore di un intercettatore per un difensore che deve fermare il pallone che attraversa il campo nelle vicinanze del robot. Per fare ciò ho dovuto sviluppare il codice di una "carta" che descrive i vari stati in cui passa il robot nel momento in cui decide di intercettare una palla, e inserire poi tale carta in un "deck" dalla quale il "dealer" pescherà questa carta, ossia il momento in cui decide di attivare la fase di intercettazione del robot difensore.

Contents

1	Introduzione	1
1.1	Struttura della tesi	1
2	Intelligenza artificiale	2
2.1	storia e scoperte	2
2.2	machine learning	3
2.3	robotica	3
3	Robocup	4
3.1	storia	4
3.2	B-human	6
3.3	Robot NAO	6
3.4	Software B-Human sui robot NAO	8
3.5	Applicazione SimRobot	9
4	Architettura B-Human	10
4.1	Sistemi di coordinate usate in B-Human	10
4.2	Trasformazioni spaziali	12
4.3	Misure Spazio e Tempo	13
4.4	Threads	14
4.5	Moduli e Rappresentazioni	15
4.5.1	BlackBoard	15
4.5.2	Definizione di Modulo	17
4.5.3	Configurazione providers e threads	19
4.5.4	Rappresentazioni di Default	19
4.5.5	Parametrizzare i Moduli	19
5	Arearie di Sviluppo	21
5.1	Perception	21
5.2	Modeling	22
5.3	Behavior Control	23
5.3.1	CABSL	24
5.3.2	Sistema di Skills e Cards	28
5.4	Motion Control	31
6	Task assegnata: robot difensore intercettatore	33
6.1	Problema	33
6.2	Approccio e risoluzione	34
6.2.1	Orientamento nella directory	34
6.2.2	Rappresentazione Intercept-ball	34
6.2.3	Carta defgotoball	36

7 Risultati	42
7.1 Casistiche verificate di intercettazione	43
8 Conclusione	45
Bibliography	46

Chapter 1

Introduzione

1.1 Struttura della tesi

La tesi seguirà la seguente struttura:

- Nel capitolo 1 introduciamo la struttura della tesi.
- Nel capitolo 2 è presente una piccola introduzione all' intelligenza artificiale e le sue varie branchie di studio e sviluppo.
- Nel capitolo 3 vi è una introduzione generale della Robocup, ossia la parte di cui parleremo di piu' in questa tesi.
- Nel capitolo 4 studiamo tutta l'architettura B-Human e cerchiamo di capirla a fondo poichè essa sarà la base per risolvere il compito assegnatoci e dovremo infatti programmare dentro questa architettura.
- Nel capitolo 5 spieghiamo tutte le aree di sviluppo possibile dentro il codice B-Human, codice su cui il Team SPQR si basa fortemente per programmare i propri robot.
- Nel capitolo 6 iniziamo a spiegare il compito assegnatomi dal Team SPQR Robocup e sul come ho risolto tale compito insieme al mio collega Emilio Leo.
- Nel capitolo 7 mostriamo i positivi risultati ottenuti.
- Nel capitolo 8 vi è una conclusione finale per tale tesi.

Tale organizzazione renderà possibile lo studio approfondito degli argomenti che affronteremo.

Chapter 2

Intelligenza artificiale

2.1 storia e scoperte

L'Intelligenza Artificiale (IA) è un ramo della scienza informatica che si concentra sulla creazione di macchine capaci di pensare e apprendere come gli esseri umani. Questo campo è in continua evoluzione e ha avuto un impatto significativo su vari settori, tra cui la medicina, l'istruzione, il trasporto e l'industria. Le sue radici risalgono agli anni '50, dove il contributo fondamentale fu quello di Alan Turing, considerato uno dei padri dell'informatica moderna. Nel 1950, Turing propose quello che sarebbe divenuto noto come test di Turing, secondo il quale una macchina poteva essere considerata intelligente se il suo comportamento, osservato da un essere umano, fosse considerato indistinguibile da quello di una persona. Nel 1956, al Darmouth College, nel New Hampshire, si tenne un convegno al quale presero parte i maggiori esponenti dell'informatica: in quell'occasione si raccolsero i principali contributi sul tema, ponendo anche l'attenzione sugli sviluppi futuri. Da allora, l'IA ha visto un'evoluzione costante, con le prime teorie di reti neurali, di IA forte e debole, e le prime applicazioni industriali degli anni '80. L'Intelligenza Artificiale può essere suddivisa in due tipi principali: IA debole e IA forte. IA debole (o IA specifica): Questo tipo di IA è progettato per eseguire un task specifico, come il riconoscimento vocale o l'analisi dei dati. Nonostante il termine "debole", queste IA sono estremamente efficaci nel loro campo specifico. Ad esempio, un sistema di IA debole potrebbe essere in grado di battere un campione del mondo a scacchi, ma non sarebbe in grado di comprendere un libro o di svolgere altre attività al di fuori del suo campo specifico. IA forte (o IA generale): Questo tipo di IA è un sistema con capacità cognitive complete. Quando presentato con un task sconosciuto, un sistema di IA forte è in grado di trovare una soluzione senza intervento umano. In teoria, un'IA forte potrebbe eseguire qualsiasi task che un essere umano può fare. Basandosi sul paradigma dell'Intelligenza Artificiale Debole, a partire dagli anni Ottanta sono state sviluppate le prime applicazioni di Intelligenza Artificiale in ambito industriale. In particolare, la prima intelligenza artificiale applicata in ambito commerciale fu R1, sviluppata nel 1982 dall'azienda Digital Equipment per configurare gli ordini di nuovi computer: quattro anni dopo, l'azienda era in grado di risparmiare 40 milioni di dollari all'anno. Il progetto Robocup di cui parleremo in questa tesi farà parte del tipo di IA debole. Nella storia dell'intelligenza artificiale e della robotica, nel maggio 1997, IBM Deep Blue ha sconfitto il campione mondiale umano negli scacchi. Quaranta anni di sfida nella comunità dell'IA si sono conclusi con successo. Il 4 luglio 1997, la missione MARS Pathfinder della NASA ha effettuato un atterraggio di successo e il primo sistema di robotica autonomo, Sojourner, è stato dispiegato

sulla superficie di Marte. Oggi, l'IA è al centro delle scelte tecnologiche di imprese e governi, e fa parte della vita quotidiana di tutti noi.

2.2 machine learning

L'Apprendimento Automatico (Machine Learning, ML) è parte fondamentale dell' I.A. che si concentra sull'addestramento dei modelli di IA per fare previsioni o prendere decisioni senza essere esplicitamente programmati per eseguire il task. Ecco come funziona in termini generali. Raccolta dei dati: Il processo inizia con un insieme di dati, noto come set di dati. Questi dati possono provenire da molte fonti diverse e possono includere informazioni come immagini, testo o numeri. Preparazione dei dati: I dati vengono poi preparati per l'apprendimento automatico. Questo può includere la pulizia dei dati (rimozione di errori o dati inutili) e la trasformazione dei dati in un formato che il modello di apprendimento automatico può utilizzare. Addestramento del modello: Un modello di apprendimento automatico viene poi addestrato utilizzando l'insieme di dati preparato. Durante questo processo, il modello "impara" dai dati. Test del modello: Dopo che il modello è stato addestrato, viene testato per vedere quanto bene ha "imparato" dai dati. Questo viene fatto utilizzando un set di dati di test che il modello non ha mai visto prima. Utilizzo del modello: Una volta che il modello ha dimostrato di essere in grado di fare previsioni accurate, può essere utilizzato per fare previsioni su nuovi dati. L'apprendimento automatico è alla base di molte delle tecnologie che usiamo oggi, dai motori di ricerca ai sistemi di raccomandazione e molto altro ancora. È un campo in continua evoluzione con molte opportunità e sfide.

2.3 robotica

La robotica e l'intelligenza artificiale (IA) sono due campi strettamente correlati che hanno visto un notevole sviluppo negli ultimi decenni.

La robotica riguarda la progettazione, costruzione e utilizzo di robot per eseguire compiti che normalmente richiederebbero l'intervento umano. Questo include l'uso di robot in settori come l'industria, la medicina, l'esplorazione spaziale e molti altri. I robot possono variare notevolmente nella loro forma e funzione, da semplici manipolatori industriali a complessi robot antropomorfi come ASIMO.

Negli anni '70, la ricerca sulla robotica ha iniziato a concentrarsi sull'intelligenza artificiale. I primi modelli di IA erano basati su regole e algoritmi predefiniti, ma in seguito l'introduzione del machine learning e delle reti neurali ha permesso alle macchine di imparare dai dati e migliorare le proprie prestazioni nel tempo.

Oggi, la combinazione di robotica e IA sta portando a sviluppi rivoluzionari in numerosi settori, dalla produzione all'assistenza sanitaria, e promette di avere un impatto significativo sulla società nel suo complesso.

Chapter 3

Robocup

3.1 storia

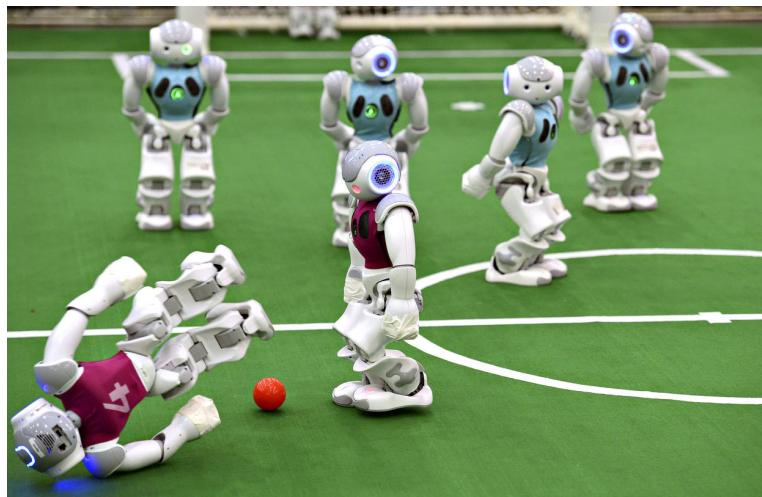


Figure 3.1. Una partita di robocup

La RoboCup è una competizione internazionale di robotica che si svolge annualmente. È stata fondata nel 1996 da un gruppo di professori universitari, tra cui Hiroaki Kitano, Manuela M. Veloso e Minoru Asada.

La RoboCup offre un task di ricerca integrato che copre ampie aree dell'IA e della robotica. Queste aree includono: fusione di sensori in tempo reale, comportamento reattivo, acquisizione di strategie, apprendimento, pianificazione in tempo reale, sistemi multiagente, riconoscimento del contesto, visione, decisioni strategiche, controllo del motore, controllo intelligente dei robot e molti altri.

L'obiettivo ufficiale del progetto è: “Entro la metà del 21° secolo, una squadra di robot giocatori di calcio completamente autonomi dovrà vincere una partita di calcio, rispettando le regole ufficiali della FIFA, contro il vincitore della Coppa del Mondo più recente”.

Ogni anno dal 1997, ricercatori di diversi paesi si riuniscono per giocare la coppa del mondo.

L'idea dei robot che giocano a calcio è stata menzionata per la prima volta dal professor Alan Mackworth (Università della British Columbia, Canada) in un articolo intitolato "On Seeing Robots" presentato al VI-92, 1992. e successivamente

pubblicato in un libro Computer Vision: System, Theory, and Applications, pagine 1-13, World Scientific Press, Singapore, 1993. Una serie di articoli sul progetto di calcio robotico Dynamo è stata pubblicata dal suo gruppo.

Indipendentemente, un gruppo di ricercatori giapponesi ha organizzato un Workshop sulle Grandi Sfide nell'Intelligenza Artificiale nell'ottobre 1992 a Tokyo, discutendo possibili problemi di grande sfida. Inoltre, sono state redatte delle regole, così come lo sviluppo di prototipi di robot da calcio e sistemi di simulazione. A seguito di questi studi, i ricercatori hanno concluso che il progetto era fattibile e desiderabile. Nel giugno 1993, un gruppo di ricercatori, tra cui Minoru Asada, Yasuo Kuniyoshi e Hiroaki Kitano, ha deciso di lanciare una competizione di robot, chiamata provvisoriamente Robot J-League (J-League è il nome della nuova lega di calcio professionistica giapponese appena istituita). Tuttavia, entro un mese, hanno ricevuto reazioni travolgenti da ricercatori al di fuori del Giappone, richiedendo che l'iniziativa fosse estesa come un progetto congiunto internazionale. Di conseguenza, hanno rinominato il progetto come Iniziativa della Coppa del Mondo di Robot, "RoboCup" per abbreviare.

Parallelamente a questa discussione, diversi ricercatori stavano già utilizzando il gioco del calcio come dominio per la loro ricerca. Ad esempio, Itsuki Noda, presso l'ElectroTechnical Laboratory (ETL), un centro di ricerca governativo in Giappone, stava conducendo ricerche su multi-agenti utilizzando il calcio, e ha iniziato lo sviluppo di un simulatore dedicato per i giochi di calcio. Questo simulatore è diventato in seguito il server ufficiale di calcio di RoboCup. Indipendentemente, il laboratorio del professor Minoru Asada presso l'Università di Osaka, e la professoressa Manuela Veloso e il suo studente Peter Stone presso la Carnegie Mellon University stavano lavorando su robot che giocano a calcio. Senza la partecipazione di questi pionieri del campo, RoboCup non avrebbe potuto decollare.

Nel settembre 1993, è stato fatto il primo annuncio pubblico dell'iniziativa, e sono state redatte specifiche regolamentazioni. Di conseguenza, si sono tenute discussioni su organizzazioni e questioni tecniche in numerose conferenze e workshop, tra cui AAAI-94, JSAI Symposium, e in vari incontri della società di robotica.

Nel frattempo, la squadra di Noda all'ETL ha annunciato la versione 0 del Soccer Server (versione LISP), il primo simulatore di sistema aperto per il dominio del calcio che consente la ricerca sui sistemi multi-agente, seguito dalla versione 1.0 del Soccer Server (versione C++) che è stata distribuita tramite il web. La prima dimostrazione pubblica di questo simulatore è stata fatta all'IJCAI-95.

Durante la Conferenza Internazionale Congiunta sull'Intelligenza Artificiale (IJCAI-95) tenutasi a Montreal, Canada, agosto 1995, è stato annunciato di organizzare i Primi Giochi e Conferenze della Coppa del Mondo di Robot in concomitanza con l'IJCAI-97 Nagoya. Allo stesso tempo, è stata presa la decisione di organizzare il Pre-RoboCup-96, al fine di identificare potenziali problemi associati all'organizzazione del RoboCup su larga scala. È stata presa la decisione di fornire due anni di tempo per la preparazione e lo sviluppo, in modo che il gruppo iniziale di ricercatori potesse iniziare lo sviluppo di robot e squadre di simulazione, oltre a dare tempo per i loro programmi di finanziamento.

Il Pre-RoboCup-96 si è tenuto durante la Conferenza Internazionale su Intelligenza Robotica e Sistemi (IROS-96), Osaka, dal 4 all'8 novembre 1996, con otto squadre che hanno gareggiato in una lega di simulazione e dimostrazione di robot reali per la lega di medie dimensioni. Pur limitata in scala, questa competizione è stata la prima competizione che utilizza i giochi di calcio per promuovere la ricerca e l'educazione.

I primi giochi ufficiali e la conferenza di RoboCup si sono tenuti nel 1997 con grande successo. Oltre 40 squadre hanno partecipato (reali e simulazione combinate), e oltre 5.000 spettatori hanno assistito.

3.2 B-human

B-Human è una squadra di calcio robotica che partecipa alla RoboCup Standard Platform League. È un progetto universitario congiunto del Dipartimento di Informatica dell'Università di Brema e del dipartimento di ricerca sui sistemi cibernetico-fisici del DFKI.

La squadra è stata fondata nel 2006 come squadra nella Humanoid League di RoboCup, ma ha iniziato a partecipare alla Standard Platform League nel 2009. Da allora, B-Human ha vinto dodici competizioni europee ed è diventata campione del mondo di RoboCup nove volte. Il numero di membri del team varia di anno in anno e si trova sempre tra 8 e 30. Oltre ad alcuni ricercatori e ex studenti ancora attivi, la maggior parte dei membri del team sono studenti dell'Università di Brema.

Un aspetto notevole del team B-Human è il loro impegno per la condivisione della conoscenza. Rilasciano la maggior parte del loro codice su base annuale su GitHub. Questo codice non solo permette ad altre squadre di apprendere dalle loro innovazioni, ma fornisce anche un'opportunità per gli studenti e i ricercatori di tutto il mondo di studiare e costruire sulle loro idee.

Il loro sito web descrive gli aspetti più tecnici del loro codice, cioè come installarlo, la sua architettura e come usarlo. Inoltre, ogni rilascio del codice viene fornito con un documento che presenta le implementazioni e le innovazioni per quell'anno specifico o RoboCup. Baseremo la progettazione del software di un robot difensore intercettatore sulla piattaforma e codice rilasciata da B-human in linguaggio c++. Nella Standard Platform League (SPL), lega in cui compete la squadra B-Human, tutte le squadre utilizzano la stessa piattaforma robotica, che non possono modificare, e sviluppano un software intelligente che fa giocare autonomamente a calcio i robot. Il robot standard attuale è il robot umanoide NAO. I robot giocano in squadre di sette l'uno contro l'altro su un campo di circa 75 metri quadrati. Durante una partita, tutti i robot devono agire completamente autonomamente ma possono comunicare con i loro compagni di squadra. Non è consentito alcun intervento da parte dei membri umani della squadra.

3.3 Robot NAO

La Lega della Piattaforma Standard utilizza attualmente i robot NAO prodotti da Aldebaran. I robot NAO hanno una forma umanoide, alti 60 cm e percepiscono il loro ambiente in molti modi, ad esempio le telecamere, montate una sopra l'altra nella testa del robot, una guarda in avanti, l'altra guarda verso il pavimento davanti ai piedi del robot. Inoltre, ci sono molti altri sensori: un'unità di misura inerziale (accelerometro e giroscopio), resistori sensibili alla forza nei piedi del robot, sensori di contatto sulla testa, le mani e la parte anteriore dei piedi, microfoni e un sonar a corto raggio nel petto del robot (che non viene utilizzato da B-Human).

I robot NAO hanno 26 giunti controllati da 25 attuatori indipendenti. Questo perché un motore nell'anca controlla due giunti contemporaneamente. Per ogni attuatore, c'è anche un corrispondente sensore di posizione del giunto. Queste sono le sue caratteristiche tecniche attuali: ATOM Z530 1.6GHz CPU 1 GB RAM; 4 to 8 GB flash memory dedicate.

Il robot NAO può passare attraverso vari stati. Ecco una descrizione di questi stati.

Stato Inattivo: Quando il robot non viene utilizzato, indipendentemente dal fatto che sia acceso o spento, dovrebbe essere conservato in una posizione specifica per garantire la sua sicurezza. **Accensione:** Il robot può essere acceso con una breve

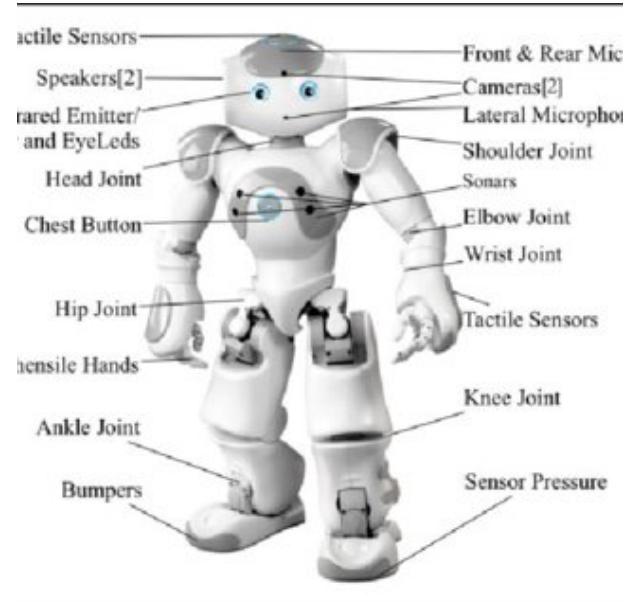


Figure 3.2. modello di robot NAO

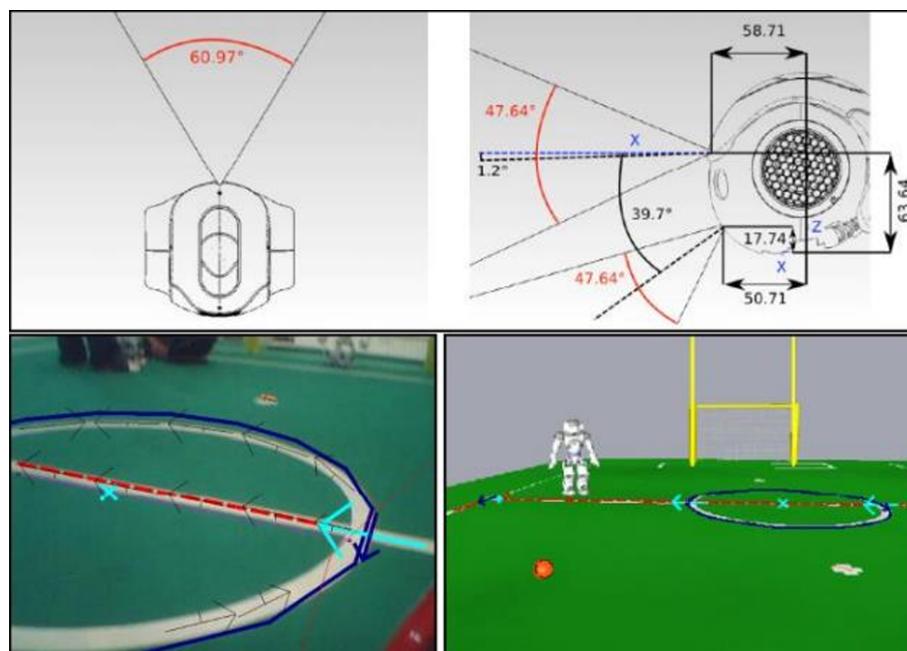


Figure 3.3. camera NAO

pressione sul pulsante sul petto. Durante l'avvio, è molto importante non muovere il robot perché il suo software sta calibrando l'unità di misura inerziale (IMU) durante questo tempo.

Stato Unstiff: Da ogni stato, è possibile passare allo stato Unstiff toccando tutti e tre i pulsanti sulla testa per almeno un secondo.

Stati Initial, Penalize, Playing, Ready e Set: Puoi passare da Unstiff a Initial a Penalize a Playing e ritorno a Penalize con una pressione del pulsante sul petto. Gli stati Ready e Set possono essere raggiunti solo in alcune demo o tramite il GameController.

Lo Stato Calibration serve per calibrare correttamente il robot, poiché il software B-Human si aspetta che tutte le parti del robot NAO siano in posizioni specifiche, è molto importante che i robot siano calibrati correttamente affinchè esso riesca a camminare in modo stabile secondo il codice. Per eseguire la calibrazione, bisogna posizionare il robot sulla linea laterale di un campo da calcio, come se il robot dovesse tornare da un calcio di rigore (in linea con la croce del rigore), guardando la croce del rigore. In seguito premere una volta il pulsante sul petto del robot per portare il robot dallo stato inattivo allo stato iniziale (in cui il robot si trova in posizione eretta). Si deve poi premere e tener premuto il pulsante sulla parte anteriore della testa e il pulsante sul petto. Dopo un secondo, il robot passerà alla modalità di calibrazione, indicata da un pulsante sul petto che brilla di un colore blu-viola. A quel punto si devono rilasciare entrambi i pulsanti.

Finita la partita, il robot va in Stato Finished.

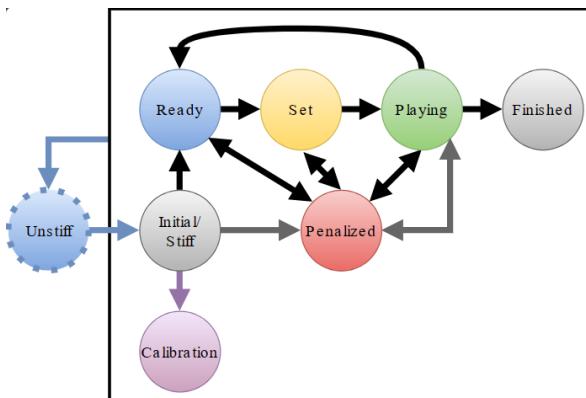


Figure 3.4. schema grafico dei vari stati in cui può passare il robot

3.4 Software B-Human sui robot NAO

Il software B-Human attualmente consiste di circa 150000 righe di codice e si basa sul framework sviluppato dal GermanTeam nel 2007, modificato per adattarsi al NAO. L'unica lingua di programmazione che utilizziamo per il software di controllo del robot e la simulazione è C++. Attualmente è possibile sviluppare e testare il software del robot su tutti e tre i principali sistemi operativi: Windows, macOS e Linux. Solo il programma binario finale viene distribuito ai robot. Alcune capacità dei nostri robot sono state realizzate applicando tecniche di machine learning. Tuttavia, i robot stessi non imparano, specialmente non durante una partita. I robot registrano dati, come immagini della telecamera, che vengono utilizzati per l'apprendimento effettivo su un computer esterno più potente. Il risultato di tale processo di apprendimento

è la configurazione di una rete neurale che può essere eseguita successivamente dai robot. Per l'addestramento delle nostre reti neurali, attualmente utilizziamo per lo più il framework TensorFlow. Per eseguirli in modo efficiente sui robot, abbiamo implementato il nostro motore di inferenza CompiledNN, che è ottimizzato specificamente per l'architettura della CPU del robot NAO.

3.5 Applicazione SimRobot

L'applicazione SimRobot è il tool fondamentale per lo sviluppo e il debugging del codice.

Nell'ambiente di simulazione, è fattibile disputare intere partite di calcio robotico 7 contro 7 e far girare un'istanza separata del codice B-Human completo per ciascun robot. SimRobot ha la capacità di simulare quasi tutti i sensori e tutti gli attuatori del robot NAO attuale. L'Open Dynamics Engine, molto diffusa, applica le leggi della fisica a tutto ciò che avviene all'interno della simulazione e gestisce anche le collisioni tra oggetti.

SimRobot, inoltre, può collegarsi a robot reali e mostrare i dati dei sensori e le informazioni di debugging inviate da un robot, oltre a inviare comandi per il debugging a un robot. È anche possibile riprodurre e analizzare all'interno di SimRobot i dati registrati da un robot durante una partita.

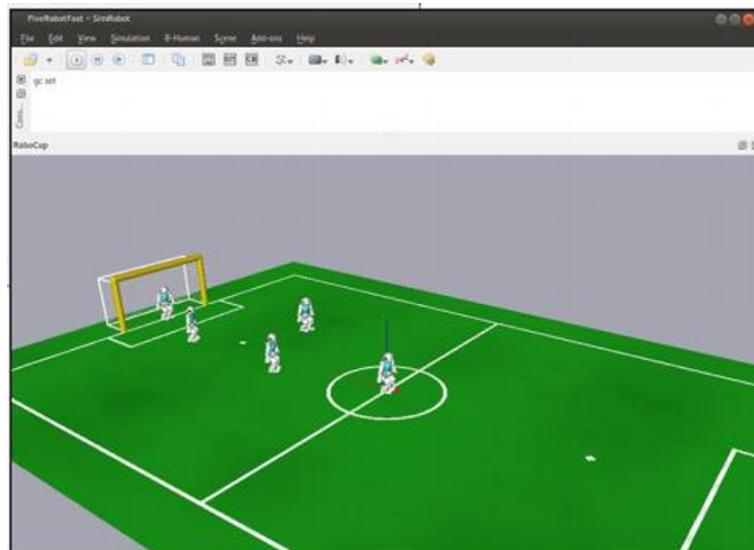


Figure 3.5. finestra dell'applicazione SimRobot

Chapter 4

Architettura B-Human

4.1 Sistemi di coordinate usate in B-Human

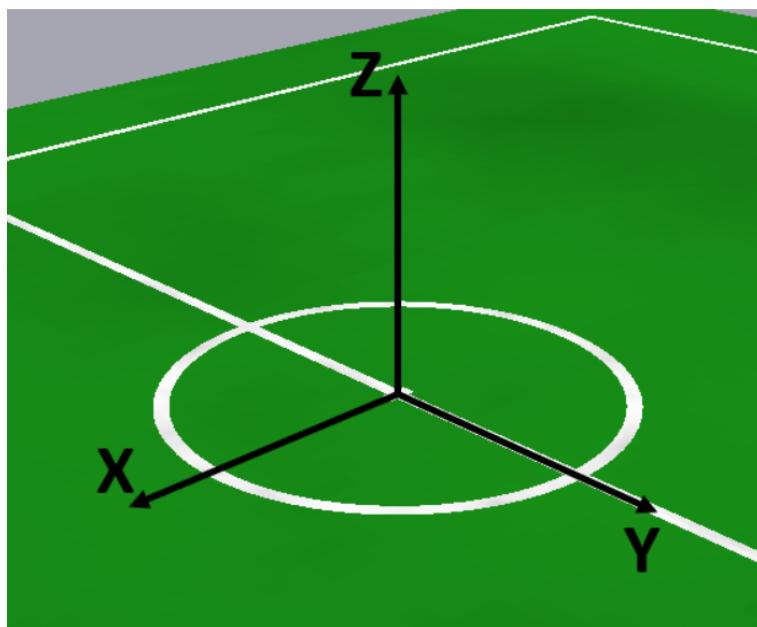


Figure 4.1. sistema di riferimento sul campo

L'origine del **sistema di coordinate del campo** è al centro del cerchio centrale. L'asse x punta verso la porta avversaria, l'asse y lungo la linea di metà campo, e l'asse z verso l'alto. Il sistema di coordinate cambia a seconda della porta su cui la squadra sta giocando. Nel simulatore, ogni squadra ha il proprio sistema di coordinate del campo.

Il **sistema di coordinate del robot** ha origine tra i piedi, con l'asse z non inclinato rispetto al campo e l'asse x che punta in avanti. Le variabili nel codice di comportamento e percezione usano spesso i suffissi Relative e OnField.

Il **sistema di coordinate del torso** del robot ha origine tra le articolazioni dell'anca, con l'asse x che punta in avanti. In alcune parti del codice, soprattutto nel movimento/sensing, questo sistema di coordinate è anche chiamato "sistema di coordinate del robot". È importante notare che questo sistema di coordinate non

coincede con il sistema di coordinate del torso di Aldebaran (la sua origine è 85mm sopra la nostra).

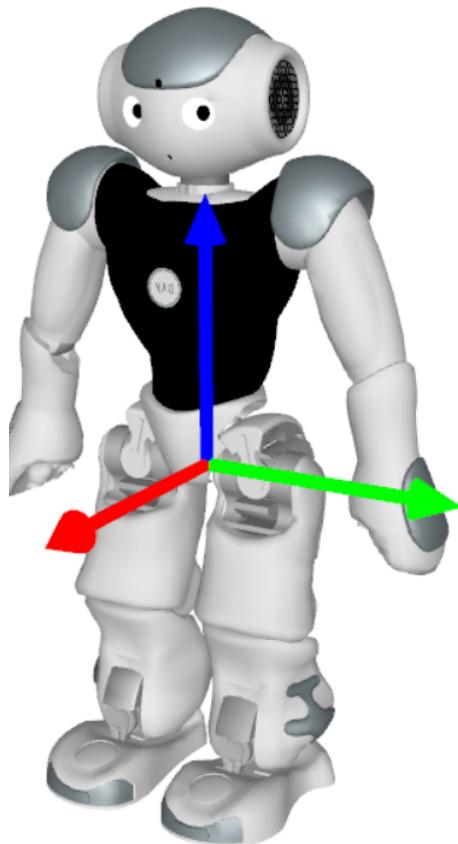


Figure 4.2. sistema di riferimento del torso del robot

Il **sistema di coordinate dei pixel** ha origine nell'angolo in alto a sinistra, con l'asse x che punta a destra e l'asse y in basso. A causa del formato YUYV, la larghezza dell'immagine della telecamera è la metà di CameraInfo::width, ma i metodi getY,U,V,YUV funzionano con coordinate relative alle dimensioni di CameraInfo. Nell'accesso all'immagine simile a una matrice/array, la coordinata y è sempre il primo indice (a causa dell'ordine di memorizzazione per righe dell'immagine).

Esiste un **sistema di coordinate dell'immagine corretto** che compensa l'effetto dell'otturatore rotante e considera gli angoli delle articolazioni e la rotazione del busto utilizzati per calcolare la posa della telecamera.



Figure 4.3. sistema di coordinate dei pixel di un immagine

4.2 Trasformazioni spaziali

Se si ha un punto o una posa in un determinato sistema di coordinate, potrebbe essere necessario convertirlo in un altro sistema di coordinate usando delle funzioni trasformazioni. La maggior parte di queste conversioni si basano sulla posa relativa dell'origine di un sistema rispetto all'origine dell'altro. Queste trasformazioni possono essere applicate in entrambe le direzioni utilizzando il metodo `.inverse()`. Tuttavia, le trasformazioni tra le coordinate della camera e quelle dei pixel seguono un approccio diverso. Lo schema sottostante mostra tutti i sistemi di coordinate usati e le trasformazioni (chiamate rappresentazioni, approfondiremo più avanti) tra di esse.

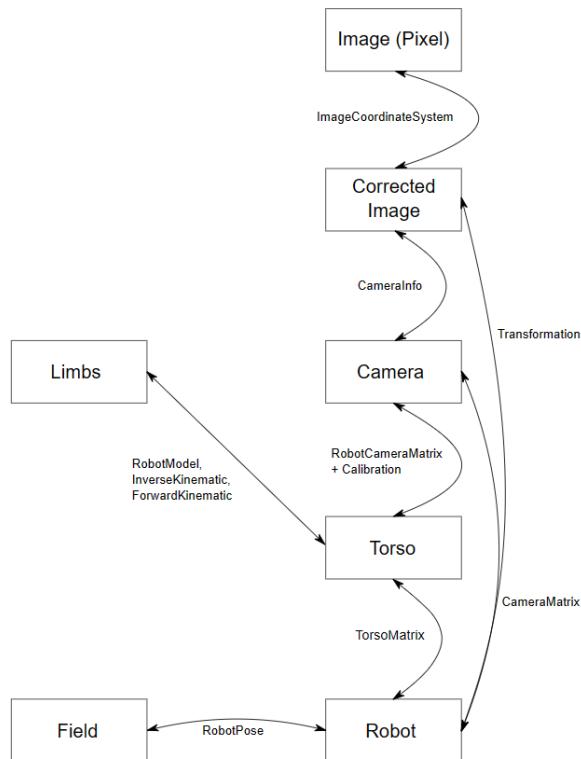


Figure 4.4. i riquadri sono i sistemi di coordinate, mentre nei collegamenti ci sono le rappresentazioni

Ad esempio se voglio passare da un punto dato nel sistema di coordinate del robot ("relative") al sistema di coordinate del campo ("OnField") scrivereò:

`pointOnField = theRobotPose * pointRelative`

Per fare il passaggio inverso scriverò:

`pointRelative = theRobotPose.inverse() * pointOnField`

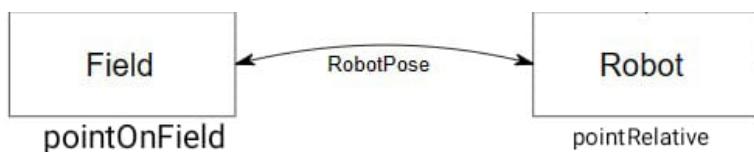


Figure 4.5. passaggio tra pointOnField e pointRelative

Representation	is/contains
RobotPose	2D pose of the robot relative to the field
TorsoMatrix	3D pose of the torso relative to the robot (on the ground)
RobotCameraMatrix	3D pose of the camera relative to the torso
CameraMatrix	3D pose of the camera relative to the robot (on the ground)
RobotModel	3D poses/positions of limbs relative to the torso

Figure 4.6. Lista di Rappresentazioni che possono essere viste come matrici trasformazioni

I metodi `fromCorrected` e `toCorrected` della rappresentazione `ImageCoordinateSystem` vengono utilizzati per le trasformazioni tra le coordinate dell’immagini corrette e le coordinate dei pixel. Per una trasformazione diretta tra le coordinate del robot e le coordinate dell’immagine (corretta), si utilizza il namespace `Transformation` (`Tools/Math`). Le funzioni chiave sono `imageToRobot` e `robotToImage`.

`ImageToRobot` prende un punto nelle coordinate dell’immagine (corretta) e lo proietta come un punto 2D rispetto al robot nel piano del terreno (se il punto nell’immagine non è sul terreno, può essere utilizzato `imageToRobotHorizontalPlane`).

`RobotToImage` fa l’operazione inversa, cioè prende un punto nelle coordinate del robot (sia 2D nel piano del terreno che 3D con altezza arbitraria) e lo proietta nelle coordinate dell’immagine (corretta).

Le varianti `robotWithCameraRotation` trasformano da/a un sistema di coordinate del robot che è ruotato attorno all’asse z in modo che l’asse x punti in avanti dal punto di vista della telecamera (anziché dal punto di vista del busto).

Gli oggetti memorizzati in coordinate relative al robot devono essere aggiornati dall’odometria. Quando si trasformano i dati tra i sistemi di coordinate, si dovrebbero considerare anche le velocità e le matrici di covarianza.

4.3 Misure Spazio e Tempo

Nel sistema B-Human, le distanze sono misurate in millimetri, gli angoli in radianti (o gradi), e il tempo in millisecondi. I timestamp iniziano a 100 secondi (100000 ms) prima dell’avvio del software, questo permette di distinguere i timestamp dal timestamp 0, che viene spesso utilizzato per indicare che non è ancora disponibile un timestamp per un certo evento.

Gran parte del codice fa uso del cosiddetto “tempo del frame”, ovvero l’istante in cui i dati attualmente in elaborazione sono stati acquisiti dai sensori (immagine della telecamera o dati sensoriali ricevuti da LoLA). `FrameInfo` è una rappresentazione disponibile in tutti i thread del framework (come `theFrameInfo`). Questa ha un campo “time” che contiene il timestamp in millisecondi e un metodo “`getTimeSince`” che consente di calcolare la durata (con segno) tra un altro timestamp e questo.

Esistono due funzioni globali che consentono di misurare direttamente il tempo: `Time::getCurrentSystemTime` e `Time::getRealSystemTime`. Nel robot, entrambe le funzioni fanno la stessa cosa, ovvero restituiscono il tempo corrente in millisecondi. Nel simulatore, invece, esiste una modalità in cui il tempo avanza solo ad ogni passaggio di simulazione. Di conseguenza, `Time::getCurrentSystemTime` restituisce un tempo proporzionale alla velocità di esecuzione della simulazione, mentre `Time::getRealSystemTime` misura sempre il tempo reale, indipendentemente dalla velocità di simulazione. Esistono anche altre due funzioni: `Time::getTimeSince` e `getRealTimeSince`.

4.4 Threads

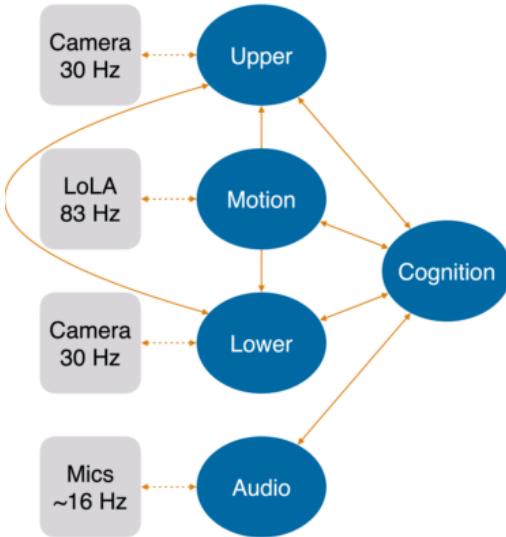


Figure 4.7. Threads

La maggior parte dei software di controllo robotico sfrutta sia i processi concorrenti che i threads. Noi, tuttavia, utilizziamo solo i thread per avere un contesto di shared memory unico. Il NAO fornisce immagini da ciascuna telecamera a una frequenza di 30 Hz e accetta nuovi comandi per gli angoli dei giunti a 83 Hz. Per gestire le immagini della telecamere usiamo due threads, ciascuno dei quali elabora le immagini di una delle due telecamere, e un terzo thread che raccoglie i risultati dell'elaborazione delle immagini ed esegue il controllo del modello del mondo e del comportamento, per sfruttare al meglio l'hardware multicore del robot. Questo ci permette di elaborare entrambe le immagini in parallelo, portando a un tempo di calcolo disponibile significativamente maggiore per immagine. Questi threads funzionano a 30 Hz e gestiscono un'immagine della telecamera ciascuno. Entrambi attivano un terzo thread, che elabora i risultati come descritto sopra. Questo funziona a 60 Hz in parallelo all'elaborazione delle immagini. Inoltre, c'è un thread che funziona alla frequenza di frame del movimento del NAO, cioè a 83 Hz e uno che registra i dati dai microfoni del NAO e li elabora. Un altro thread esegue la comunicazione TCP con un PC host per scopi di debugging.

I thread utilizzati sul NAO risultano sei : thread Upper, Lower, Cognition, Motion, Audio e Debug effettivamente utilizzati nel sistema B-Human. I thread di percezione Upper e Lower ricevono immagini della telecamera da Video per Linux. Inoltre, ricevono dati dal thread Cognition sul modello del mondo e dati sensoriali dal thread Motion. Elaborano le immagini e inviano i risultati del rilevamento al thread Cognition. Questo thread utilizza effettivamente queste informazioni insieme ai dati sensoriali dal thread Motion per il controllo del modello del mondo e del comportamento e invia comandi di movimento di alto livello al thread Motion. Quest'ultimo esegue effettivamente questi comandi generando gli angoli target per i 25 giunti del NAO. Invia questi angoli target attraverso il NaoProvider all'interfaccia LoLA del NAO, e riceve letture dei sensori come gli angoli di giunto effettivi, misurazioni di accelerazione del corpo e giroscopio, ecc.

Inoltre, Motion segnala il movimento del robot, ad esempio fornendo i risultati

del dead reckoning. Il thread Audio esegue il rilevamento del fischio se lo stato di gioco corrente lo richiede e riporta i risultati a Cognition. Il thread Debug comunica con il PC host. Distribuisce i dati ricevuti da esso agli altri thread e raccoglie i dati forniti da loro e li inoltra alla macchina host. È inattivo durante le partite di calcio.

LoLA è un’interfaccia utilizzata nell’architettura B-Human per il robot NAO. Quando il software B-Human viene avviato, contatta prima LoLA per recuperare il numero di serie del corpo del robot, che viene poi mappato su un nome di corpo utilizzando il file Config/Robots/robots.cfg .

Il modulo NaoProvider nel programma di controllo principale del robot scambia dati con LoLA seguendo il protocollo definito da Aldebaran per le squadre del RoboCup.

Inoltre, dopo la terminazione del programma di controllo principale del robot, viene stabilita un’altra connessione con LoLA per far sedere il robot se necessario, spegnere tutti i giunti e visualizzare lo stato di uscita del programma principale negli occhi del robot.

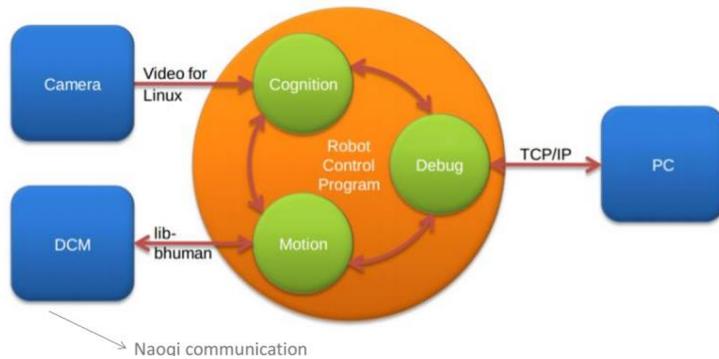


Figure 4.8. B-Human Framework

4.5 Moduli e Rappresentazioni

Un programma di controllo del robot (robot control program) è composto da **vari moduli** che svolgono **task specifici**. Questi moduli necessitano di input specifici e producono output specifici(chiamati **rappresentazioni**). Il framework introdotto da GermanTeam1 semplifica l’interfacciamento dei moduli, si utilizza infatti uno **Scheduler** per determinare automaticamente la giusta sequenza di esecuzione dei vari moduli, che dipende dai input e output dei moduli.

Il framework comprende una lavagna(BlackBoard), la definizione del modulo e un componente di visualizzazione.

4.5.1 BlackBoard

La lavagna è il deposito centrale per le informazioni, ovvero le rappresentazioni. Ogni thread è associato alla sua propria istanza della lavagna. Le rappresentazioni vengono condivise tra i thread attraverso la comunicazione tra thread se il modulo in un thread richiede una rappresentazione che è fornita in un altro thread esterno. La lavagna mappa i nomi delle rappresentazioni alle istanze di queste rappresentazioni contate e contiene solo le rappresentazioni richieste o fornite da almeno un modulo nel thread associato.

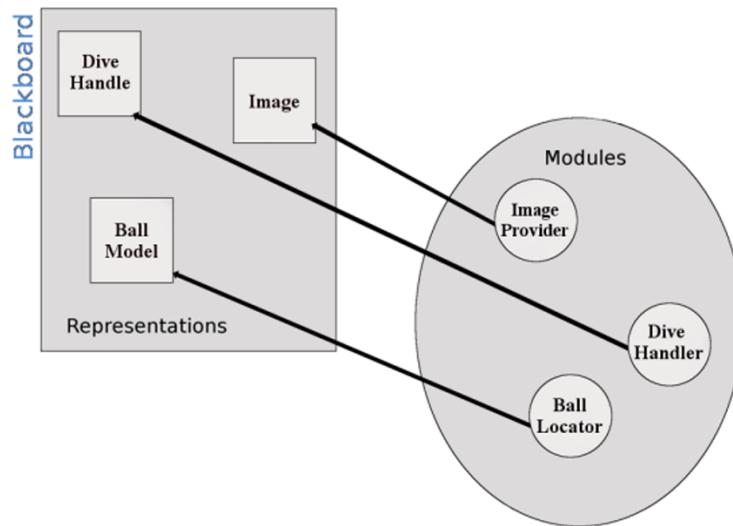


Figure 4.9. lavagna, moduli e rappresentazioni

```

#pragma once
#include "Tools/Streams/Enum.h"
#include "Tools/Math/Eigen.h"
#include "Tools/Streams/AutoStreamable.h"
class MyRepresentation : public Streamable
{
private :
    void serialize(In *in, Out *out)
    {
        STREAM_REGISTER_BEGIN;
        STREAM(robotPose);
        STREAM(ballPose);
        STREAM_REGISTER_FINISH;
    }
public:
    Vector2f ballPose;
    Vector2f robotPose;

    MyRepresentation(){;}
};

```

Figure 4.10. Template di Rappresentazione in c++

4.5.2 Definizione di Modulo

La Definizione di Modulo è divisa in 3 parti: la interfaccia del modulo, la sua implementazione e una dichiarazione che consente di istanziare il modulo.

L’interfaccia del modulo stabilisce il nome del modulo (ad esempio, SimpleBall-Locator), le rappresentazioni richieste per svolgere il suo task, le rappresentazioni fornite dal modulo e i parametri del modulo, i cui valori possono essere definiti sul posto o caricati da un file. L’interfaccia crea essenzialmente una classe base per il modulo effettivo seguendo lo schema di denominazione <NomeModulo>Base. L’implementazione effettiva del modulo è una classe derivata da quella classe base.

I seguenti statements sono disponibili in una definizione dell’interfaccia del modulo:

REQUIRES e **USES** che definiscono gli inputs del modulo;

PROVIDES che definiscono gli outputs del modulo.

Si deve definire un metodo update per ogni rappresentazione fornita.

- REQUIRES dichiara che questo modulo ha accesso in sola lettura alla rappresentazione indicata nella lavagna (es. REQUIRES(BallPercept); BallPercept è la rappresentazione della palla vista dal robot). **I moduli si aspettano che tutte le loro rappresentazioni richieste con REQUIRES siano state aggiornate prima che venga chiamato uno qualsiasi dei loro metodi provider.** Se una rappresentazione richiesta è fornita da più thread, deve essere utilizzato un alias per specificare quale rappresentazione è utilizzata dal modulo. Un alias è una rappresentazione che ha come prefisso il nome del thread prima del nome effettivo, ad esempio <thread><rappresentazione>. La rappresentazione deve essere derivata dalla rappresentazione effettiva senza aggiungere membri ad essa.
- USES dichiara che questo modulo accede a una rappresentazione, che **non deve necessariamente essere aggiornata prima che il modulo venga eseguito**. Questo dovrebbe essere utilizzato solo per risolvere conflitti nella configurazione(ad esempio Deadlock causato da 2 moduli, si guardi la figure 4.11.). Si noti che USES non viene considerato quando si scambiano dati tra thread.

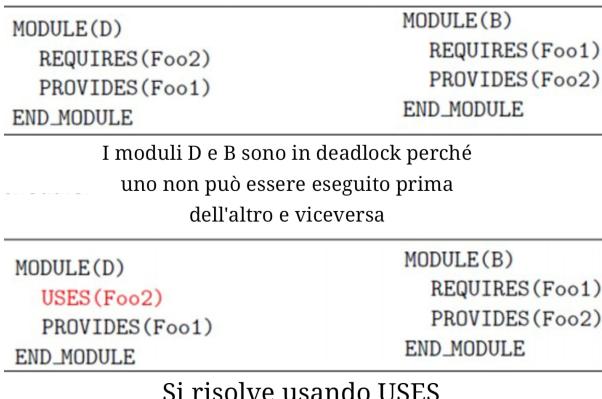


Figure 4.11. poichè usando USES la rappresentazione Foo2 non deve necessariamente essere aggiornata prima che il modulo sia eseguito, si risolve il deadlock e parte prima il modulo D e dopo il modulo B

- PROVIDES dichiara che questo modulo aggiornerà questa rappresentazione. Per ogni rappresentazione, il tempo di esecuzione può essere determinato (es. PROVIDES(BallModel)) e può essere inviato a un PC host o addirittura modificato da esso.
- PROVIDES-WITHOUT-MODIFY fa esattamente la stessa cosa di PROVIDES, con l'unica differenza che la rappresentazione non può essere ispezionata o modificata dal PC host.
- DEFINES-PARAMETERS e LOADS-PARAMETERS consentono la parametrizzazione modificabile dei moduli. Si raccomanda di utilizzare questa opzione per tutti i parametri

```

MODULE(SimpleBallLocator,
{,
    REQUIRES(BallPercept),
    REQUIRES(FrameInfo),
    PROVIDES(BallModel),
    DEFINES_PARAMETERS(
    {,
        (Vector2f)(5.f, 0.f) offset,
        (float)(1.1f) scale,
    }),
});
};

class SimpleBallLocator : public SimpleBallLocatorBase
{
    void update(BallModel& ballModel) override
    {
        if(theBallPercept.wasSeen)
        {
            ballModel.position = theBallPercept.position * scale + offset;
            ballModel.wasLastSeen = theFrameInfo.time;
        }
    }
}

MAKE_MODULE(SimpleBallLocator);

```

Figure 4.12. Template di Modulo in c++

L'istruzione MAKE-MODULE consente di creare un'istanza del modulo. Il secondo parametro opzionale può sovrascrivere la funzione statica che fornisce l'elenco delle rappresentazioni richieste e fornite al sistema. **Mentre l'interfaccia del modulo è generalmente parte del header file, l'istruzione MAKE-MODULE deve essere inserita nel file di implementazione.**

MODULE è una macro che riceve tutte le informazioni sul modulo come parametri, separati da virgole. La macro ignora il suo secondo e il suo ultimo parametro, poiché, per convenzione, questi vengono utilizzati per le parentesi graffe di apertura e chiusura. Questo consente a alcuni strumenti di formattazione del codice sorgente di indentare le definizioni come un blocco. Attualmente, MODULE è limitato a un massimo di 90 definizioni tra le parentesi graffe. Quando la macro viene espansa, crea

molte funzionalità nascoste. Ogni voce che fa riferimento a una rappresentazione si assicura che venga creata nella bacheca quando il modulo viene costruito e liberata quando il modulo viene distrutto. Le informazioni sul fatto che un modulo abbia determinati requisiti e fornisca determinate rappresentazioni non vengono utilizzate solo per generare una classe di base per quel modulo, ma sono anche disponibili per ordinare i provider e possono essere richieste da un PC ospitante. Su un PC ospitante, queste informazioni possono essere utilizzate per modificare la configurazione e per la visualizzazione. Se esiste un MessageID id<representation>, la rappresentazione può anche essere registrata.

Se una rappresentazione fornita definisce un metodo senza parametri chiamato draw, tale metodo verrà chiamato dopo l'aggiornamento della rappresentazione. Se la rappresentazione definisce un metodo senza parametri chiamato verify, tale metodo verrà chiamato nelle build di Debug e Develop dopo l'aggiornamento della rappresentazione. Un metodo verify dovrebbe contenere ASSERT che controllano se i contenuti della rappresentazione sono plausibili. Entrambi i metodi vengono chiamati solo se sono definiti nella rappresentazione stessa e non se sono ereditati dalla sua classe base.

4.5.3 Configurazione providers e threads

I moduli possono fornire diverse rappresentazioni e la configurazione avviene a livello di fornitori. Normalmente, la configurazione viene letta dal file Config/Scenarios/<scenario>/threads.cfg durante l'avvio del programma di controllo del robot, ma può anche essere modificata interattivamente quando il robot ha una connessione di debug a un PC host utilizzando il comando mr.

Questo permette diverse configurazioni in scenari diversi. La sequenza di esecuzione dei fornitori viene determinata automaticamente e solo le configurazioni valide vengono inviate ai thread.

In alcuni casi, una rappresentazione deve essere fornita da un modulo prima di qualsiasi altra rappresentazione dello stesso modulo. Ad esempio, quando il task principale del modulo viene eseguito nel metodo update di quella rappresentazione, e gli altri metodi update si basano sui risultati calcolati nel primo. Questo caso può essere implementato sia richiedendo che fornendo una rappresentazione nello stesso modulo.

4.5.4 Rappresentazioni di Default

Durante lo sviluppo del software di controllo del robot, può essere utile disattivare un fornitore (provider) o modulo. Non fornire una rappresentazione può rendere l'insieme dei fornitori inconsistente e avere un effetto a cascata. In alcuni casi, è possibile disattivare un fornitore senza influire sulle dipendenze tra i moduli, inserendo rappresentazioni in una lista separata nel file di configurazione. Queste rappresentazioni predefinite non cambiano mai il loro valore - quindi rimangono sostanzialmente nel loro stato iniziale - ma esistono ancora nella lavagna, e quindi, tutte le dipendenze possono essere risolte. Tuttavia, in termini di funzionalità, una configurazione che utilizza quella lista non è completa.

4.5.5 Parametrizzare i Moduli

Per un funzionamento adeguato, i moduli richiedono generalmente alcuni parametri. Questi possono essere specificati nella descrizione dell'interfaccia del modulo. I parametri agiscono come componenti di classe protetti e l'accesso a essi avviene nello

stesso modo. Possono anche essere modificati dalla console attraverso i comandi get/set parameters:<ModuleName>.

Ci sono due metodi per inizializzare i parametri nei moduli: uno codificato e uno caricabile. Nel metodo codificato, i valori di inizializzazione sono definiti nel modulo usando la macro DEFINES-PARAMETERS. Nel metodo caricabile, i parametri vengono inizializzati con valori caricati da un file di configurazione al momento della creazione del modulo, utilizzando la macro LOADS-PARAMETERS. Di default, i parametri vengono caricati da un file con lo stesso nome del modulo ma con la prima lettera minuscola e l'estensione.cfg. È anche possibile assegnare un nome personalizzato al file di configurazione di un modulo.

Solo DEFINES-PARAMETERS o LOADS-PARAMETERS possono essere utilizzati in una definizione di modulo. Entrambi possono essere utilizzati solo una volta.

Chapter 5

Arene di Sviluppo

L'intero software del robot è suddiviso in piu' parti. Questi sono raggruppati in un insieme di categorie che riflettono le diverse aree di ricerca e sviluppo.

5.1 Perception

La percezione si occupa di processare le immagini catturate dalle telecamere e di individuare gli oggetti presenti. Attualmente, B-Human è in grado di rilevare la palla, altri robot, le linee del campo, i loro incroci, i marchi di penalità e il cerchio centrale. Il primo passaggio nel processo generale consiste nel suddividere l'immagine YCbCr grezza, fornita dalla telecamera, in immagini piatte di grayscale, saturation e hue.



Figure 5.1. raw, grayscale, saturation, hue

I processi di pre-elaborazione includono l'identificazione di regioni con lo stesso colore sulle linee di scansione e la rilevazione del limite tra il campo e l'ambiente circostante. Successivamente, entrano in gioco moduli specializzati per ogni categoria di oggetto. I percettori delle linee del campo e del cerchio centrale si basano interamente sulle linee di scansione e sull'immagine classificata per colore. La percezione di palle, segni di penalità e incroci di linee si svolge in due fasi: la prima identifica i punti candidati e la seconda li classifica utilizzando reti neurali convoluzionali. La percezione del robot impiega un'altra rete neurale convoluzionale che lavora su un'immagine in scala di grigi ridimensionata e predice direttamente le aree di delimitazione dei robot.

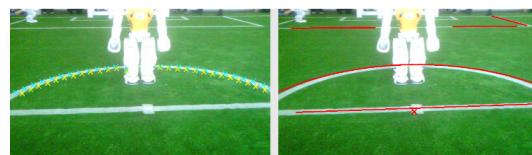


Figure 5.2. contorni campo di gioco

5.2 Modeling

Il ruolo del componente di modeling è quello di incorporare i risultati rumorosi derivanti dalla percezione in un modello uniforme degli oggetti presenti nell'ambiente nel corso del tempo. Questo modello comprende la posizione e la velocità della palla, la posizione e l'orientamento del robot sul campo, e le posizioni e le velocità degli altri robot. In questo contesto, “rumore” non indica solo che le misurazioni sono imprecise, ma che potrebbero anche esserci misurazioni completamente errate (falsi positivi) o nessuna misurazione (falsi negativi). Di conseguenza, tutte le soluzioni aderiscono al principio generale di associare prima le misurazioni con le ipotesi di oggetto formate nel tempo, potenzialmente eliminando i falsi positivi, e successivamente aggiornando un stimatore di stato probabilistico con la misurazione per ogni ipotesi.

B-Human utilizza combinazioni di filtri di Kalman e filtri di particelle per inferire valori non direttamente misurabili come la posizione del robot o la velocità della palla. A causa delle limitazioni del campo visivo, ogni robot ha un modello dell'ambiente incompleto e quindi scambia informazioni tramite WiFi. Queste informazioni vengono incorporate nei processi di stima dello stato.

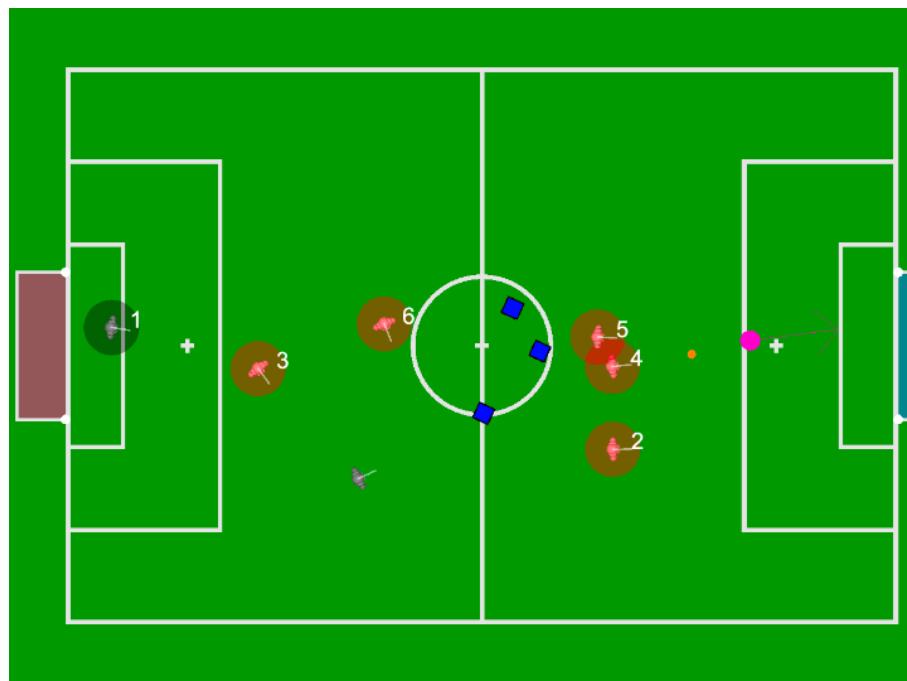


Figure 5.3. esempio di modello del mondo del robot numero 7 durante la finale della RoboCup 2023 contro HTWK Robots

Guardando la figura 5.3, la posizione del robot è mostrata da un piccolo disegno di robot all'interno della propria metà. Le posizioni assunte dei compagni di squadra sono segnate con cerchi e numeri. Le caselle blu indicano i robot avversari recentemente osservati. Il cerchio arancione è l'ultima posizione stimata della palla, mentre il cerchio rosa più grande mostra una stima della posizione della palla basata sulle informazioni comunicate dai compagni di squadra. Tutte queste informazioni costituiscono la base per gli algoritmi decisionali all'interno dei moduli di comportamento.

5.3 Behavior Control

Sulla base del modello del mondo stimato e delle informazioni ricevute dai suoi compagni di squadra, un robot deve determinare la prossima azione da eseguire. **Questo processo è strutturato in vari strati.** Il livello più alto è il **componente strategico**, che emana un comando di alto livello come “tira verso la porta”, “passa a un compagno” o “muoviti verso una posizione”. Questo livello gestisce la scelta di una tattica (ad esempio, difensiva o offensiva a seconda della posizione della palla), le azioni predefinite come i calci d’inizio o i calci liberi, l’assegnazione dei ruoli tra i compagni di squadra e il posizionamento strategico. L’assegnazione dei ruoli cerca di garantire che ce sempre un giocatore che voglia giocare con la palla, ma data la possibile inconsistenza dei modelli del mondo e la comunicazione limitata, esso potrebbe non funzionare. Gli altri giocatori scelgono una posizione obiettivo, sia per possibili passaggi sia per difendere i tiri sulla propria porta.

Sotto la strategia, c’è un **skill layer** che interpreta il comando della strategia. Esso ha una certa autonomia e può sovrascrivere il comando di strategia se questo non può essere eseguito o se sono necessarie altre reazioni a breve termine (l’obiettivo di questo è ridurre il numero di casi da gestire nel codice di alto livello). Ad esempio, il duello con un avversario o l’intercettazione della palla come portiere sono gestiti su questo livello e non sono comandati dalla strategia, ma sono iniziati dal skill layer. Allo stesso modo, il skill layer evita di entrare in aree che sono illegali secondo le regole, ad esempio l’area intorno alla palla quando a un avversario è stato assegnato un calcio libero, o cerca di lasciarle se già all’interno.

Se il comando di alto livello viene accettato, è ancora task di questo modulo riempire i dettagli, ad esempio per un comando di tiro, il tipo di calcio concreto e l’angolo sulla porta devono essere selezionati.

Le Skills formano una gerarchia in cui le skill di livello superiore possono chiamare skill di livello inferiore. Il livello più basso di solito soddisfa una richiesta per il motore di movimento (che funziona in un thread diverso) e sceglie un percorso verso l’obiettivo desiderato. Per le destinazioni vicine, questo significa semplicemente reagire agli ostacoli e allinearsi all’orientamento target. Per le destinazioni più lontane, invochiamo un pianificatore di percorsi che esegue una ricerca su un grafo di visibilità 2D per evitare di rimanere bloccati. Quest’ultimo è visualizzato nell’immagine sottostante, in cui il robot che si trova a sinistra del cerchio centrale pianifica un percorso verso la palla.

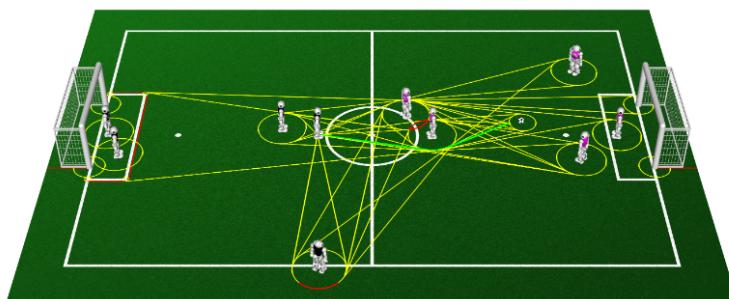


Figure 5.4

Alcune Skill sono implementate come macchine a stati. Per questo usiamo **CABSL**, un linguaggio sviluppato appositamente per questo motivo.

5.3.1 CABSL

C-based Agent Behavior Specification Language (CABSL) è progettato per descrivere e sviluppare un comportamento (behavior) dell'agente come una gerarchia di macchine a stati in C++, ed è derivato da XABSL: eXtensible Agent Behavior Specification Language. Un programma di controllo del robot viene eseguito in cicli. In ogni ciclo, l'agente acquisisce nuovi dati dall'ambiente, ad esempio attraverso i sensori, esegue il suo comportamento e poi esegue i comandi che il comportamento ha calcolato. Questo significa che un programma di controllo del robot è un grande ciclo, ma il behavior è solo una mappatura dallo stato del mondo alle azioni che considerano anche le decisioni prese in precedenza.

CABSL è composto principalmente da: opzioni, stati, transizioni, azioni. **Ogni opzione è una macchina a stati finiti** che descrive una parte specifica del comportamento, come una competenza o un movimento della testa del robot, o combina tali caratteristiche di base. **Il behavior è visto come un insieme di opzioni** che sono disposte in modo da formare un grafo: **option graph**.

In ogni ciclo di esecuzione del programma di controllo del robot, un sottoinsieme di tutte le possibili opzioni vengono eseguite a partire dalla opzione radice del option graph, queste opzioni formano un albero, che è un sottoalbero del più generale option graph e viene chiamato **albero di attivazione delle opzioni**.

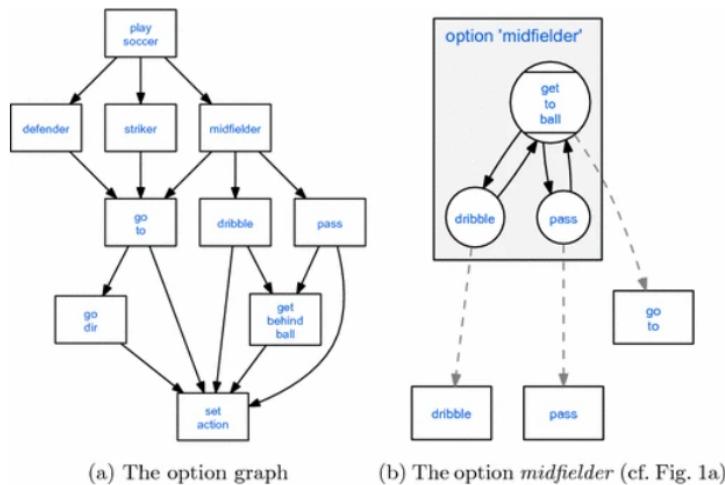


Figure 5.5

Opzioni

Poichè ogni opzione è una macchina a stati finiti, queste opzioni sono composte da più stati e ogni stato ha un **albero decisionale** con **transizioni** verso altri stati nella stessa opzione. Ogni stato può chiamare delle **azioni** che l'opzione esegue se sta attualmente in quello stato, un'azione possibile è la chiamata di un'altra opzione esterna, il che porterebbe a formare un grafo aciclico diretto nel albero di attivazione delle opzioni.

In ogni ciclo di esecuzione, partendo dall'opzione radice, ogni opzione cambia il proprio stato attuale se necessario e poi esegue le azioni elencate nello stato attivo in cui si trova al momento. Le azioni possono impostare valori di output o possono chiamare altre opzioni, che a loro volta potrebbero cambiare il proprio stato attuale

seguito dall'esecuzione di altre azioni. Per ogni ciclo di esecuzione, ogni opzione cambia il proprio stato attuale al massimo una volta

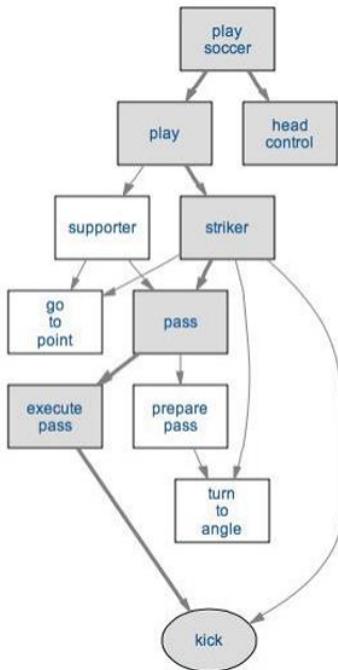


Figure 5.6. albero di attivazione delle opzioni

Stati

Le opzioni possono avere un numero di stati. Uno di questi è lo stato iniziale, nel quale l'opzione inizia quando viene chiamata per la prima volta e al quale ritorna se viene eseguita di nuovo, ma non è stata eseguita nel ciclo precedente, perché nessun'altra opzione l'ha chiamata. Ci sono altri due tipi di stati speciali, ovvero gli stati target e gli stati interrotti. Sono intesi per segnalare all'opzione chiamante che l'opzione corrente è riuscita o fallita. L'opzione chiamante può determinare se l'ultima sotto-opzione che ha chiamato ha raggiunto uno di questi stati. Spetta al programmatore definire cosa significhino effettivamente riuscire o fallire, cioè in quali condizioni questi stati vengono raggiunti.

Transizioni

Gli stati possono avere transizioni verso altri stati. Sono essenzialmente alberi decisionali, in cui ogni foglia contiene un'istruzione goto verso un altro stato. Se nessuna di queste foglie viene raggiunta, l'opzione rimane nel suo stato attuale.

Un'opzione può avere una transizione comune. Il suo albero decisionale viene eseguito prima del albero decisionale nello stato corrente. L'albero decisionale dello stato corrente funziona come il ramo else del albero decisionale nella transizione comune. Se la transizione comune risulta già in un cambiamento dello stato corrente, le transizioni definite negli stati non vengono più controllate. In generale, le transizioni comuni dovrebbero essere usate con parsimonia, perché entrano in conflitto con l'idea generale che ogni stato ha le sue condizioni sotto le quali lo stato viene lasciato.

Azioni

Gli stati hanno azioni che vengono eseguite se l'opzione si trova in quello stato. Queste azioni definiscono simboli di output e/o invocano altre opzioni. Nonostante sia possibile utilizzare qualsiasi istruzione C++ all'interno di un blocco di azioni, è consigliabile evitare le istruzioni di controllo, come i rami e i cicli. Le transizioni di stato gestiscono meglio i rami, mentre i cicli dovrebbero essere implementati nelle funzioni che vengono eseguite prima o dopo il comportamento, o che vengono invocate dal comportamento stesso. È possibile invocare più sotto-opzioni da uno stato, ma solo l'ultima chiamata può essere controllata per aver raggiunto uno stato target o uno stato interrotto.

In seguito vi è un esempio di un opzione con 3 stati:

```
option(exampleOption)
{
    initial_state(firstState)
    {
        transition
        {
            if(booleanExpression)
                goto secondState;
            else if(lib.Example.boolFunction())
                goto thirdState;
        }
        action{
            providedRepresentation.value=requiredRepresentation.value * 3;
        }
    }
    state(secondState)
    {
        action{
            SecondOption();
        }
    }
    state(thirdState)
    {
        transition
        {
            if(booleanExpression)
                goto firstState;
        }
        action{
            providedRepresentation.value=RequiredRepresentation::someEnumValue;
            ThirdOption();
        }
    }
}
```

Figure 5.7. template di un opzione in c++

Inoltre, le opzioni possono avere parametri che possono essere utilizzati come normali parametri di funzione. La specifica dei parametri utilizza la stessa sintassi della definizione del tipo di dati trasmissibili nel nostro sistema. In questo modo, i parametri effettivi di ogni opzione possono essere registrati nei file di log e possono anche essere mostrati nella finestra di dialogo del comportamento.

```

option(OptionWithParameters, int i, bool b, int j = 0)
{
    initial_state(firstState)
    {
        action
        {
            providedRepresentation.intValue = b ? i : j;
        }
    }
}

```

Figure 5.8. esempio di opzione con parametri

HandleGameState	104.79
state = ready	17.77
ArmContact	17.77
ArmObstacleAvoidance	17.77
WalkToKickoffPose	17.77
target = { rotation = -5deg; translation = { x = -3	
state = walkToKickoffPose	17.77
LookActive	17.77
ignoreBall = true	
WalkToPoint	17.77
target = { rotation = 47.5899deg; translation =	
speed = 0.7	
reduceWalkingSpeed = false	
rough = true	
disableStanding = true	
state = walkClose	4.41

Figure 5.9. Nell'applicazione SimRobot vi è una componente chiamata behavior view che mostra tutte le opzioni e skills attualmente attive e gli stati in cui si trovano. Per poterlo visualizzare, si deve inviare la seguente richiesta di debug: dr representation:ActivationGraph

5.3.2 Sistema di Skills e Cards

La decomposizione e la gerarchia sono necessarie per specificare i comportamenti complessi presenti nel calcio. Col passare dei anni il team B-human ha realizzato che potrebbe non essere una buona idea cercare di adattare tutti i livelli di comportamento nella stessa formalizzazione, ma invece dividere il comportamento in due strati: uno che deciderebbe cosa dovrebbe fare il robot, dove le opzioni potrebbero essere facilmente aggiunte o rimosse, e un altro che realizzerà come il robot soddisfa questa richiesta. Questa intuizione è stata la base del nuovo framework di comportamento che è stato chiamato il sistema di Skills e Cards.

Skills

Basato su CABS, le Skills rappresentano il livello più basso del behavoir graph(ossia option graph in CABS), eseguono chiamate dirette al motion engine per azioni come calciare o camminare.

Può ricevere parametri dal suo chiamante. Le skill possono chiamare altre skill per scomporre ulteriormente il comportamento in sotto-task, formando una gerarchia che alla fine riempie i comandi di basso livello al sistema di movimento. Tecnicamente, le skill consistono di due costrutti di codice: un'interfaccia di skill e un'implementazione di skill. L'interfaccia esporta la firma con cui può essere chiamata da altre skill. Questo permette anche di avere molteplici implementazioni per la stessa skill senza che i chiamanti debbano sapere quale di esse utilizzare. Le interfacce delle skill sono dichiarate nello spazio dei nomi Skills nel file Representations/BehaviorControl/Skills.h utilizzando la macro SKILL-INTERFACE.

Il primo argomento della macro è il nome della skill. Ulteriori argomenti facoltativi specificano i parametri che la skill richiede, iniziando con il tipo tra parentesi seguito dal suo nome. È possibile specificare argomenti predefiniti indicando il valore predefinito tra parentesi dopo il tipo, ma come al solito, possono essere seguiti solo da altri argomenti predefiniti.

Questa macro dichiara effettivamente una struct con il nome della skill dato che funge da contenitore per i parametri. Inoltre, viene generata una classe chiamata <name>Skill che è un oggetto proxy che smista le chiamate esterne all'implementazione della skill. La definizione di un'interfaccia di skill è simile a quella di un modulo, ovvero viene generata tramite macro una classe base che viene ereditata dall'implementazione effettiva. I seguenti statements possono esser trovati nella definizione dell'interfaccia di una skill:

- **IMPLEMENTS** dichiara che questa classe contiene un'implementazione per una specifica interfaccia di skill. Aggiunge i metodi virtuali reset, isDone, isAborted, preProcess e postProcess e il metodo puramente virtuale execute. Ognuno di essi prende una costante referenza all'interfaccia della skill, permettendo a un metodo di accedere ai valori che sono stati passati alla skill e sovraccaricarli nel caso in cui più skill siano implementate nella stessa classe. preProcess e postProcess sono chiamati ogni frame prima e rispettivamente dopo che il comportamento è eseguito. Questo è utile, ad esempio, per dichiarare disegni di debug. reset è chiamato immediatamente prima di execute se la skill non è stata chiamata nel frame precedente. isDone e isAborted possono riportare lo stato al chiamante.
- **CALLS** dichiara che questa implementazione di skill chiama un'altra skill. È disponibile sotto il nome di the<name>Skill e può essere chiamata con

l'operatore di chiamata di funzione. Lo stato può essere interrogato con `isDone` e `isAborted`.

- **REQUIRES** dichiara che questa implementazione di skill accede a una rappresentazione che deve essere fornita prima che il comportamento venga eseguito.
- **USES** dichiara che questa implementazione di skill accede a una rappresentazione che non necessariamente deve essere aggiornata prima che il comportamento venga eseguito.
- **MODIFIES** dichiara che questa implementazione di skill modifica una rappresentazione che è fornita dal modulo in cui la skill è eseguita. Questo è limitato alle rappresentazioni che fanno parte del SkillRegistry.
- **DEFINES PARAMETERS** e **LOADS PARAMETERS** hanno esattamente lo stesso significato come nelle definizioni dei moduli. Tuttavia, **LOADS PARAMETERS** antepone BehaviorControl/ al nome del file generato.

Le implementazioni di skill devono essere pubblicate nel sistema utilizzando la macro `MAKE-SKILL-IMPLEMENTATION`, che prende come argomento il nome della classe di implementazione della skill.

```
* This file implements the implementation of the WalkToTarget skill.
*
* @author Arne Hasselbring
*/
#include "Representations/BehaviorControl/Libraries/LibCheck.h"
#include "Representations/BehaviorControl/Skills.h"
#include "Representations/MotionControl/MotionInfo.h"
#include "Representations/MotionControl/MotionRequest.h"

SKILL_IMPLEMENTATION(WalkToTargetImpl,
{
    IMPLEMENTS(WalkToTarget),
    REQUIRES(LibCheck),
    REQUIRES(MotionInfo),
    MODIFIES(MotionRequest),
});

class WalkToTargetImpl : public WalkToTargetImplBase
{
    void execute(const WalkToTarget& p) override
    {
        theMotionRequest.motion = MotionRequest::walk;
        theMotionRequest.walkRequest.mode = WalkRequest::targetMode;
        theMotionRequest.walkRequest.target = p.target;
        theMotionRequest.walkRequest.speed = p.speed;
        theMotionRequest.walkRequest.walkKickRequest = WalkRequest::WalkKickRequest();
        theLibCheck.inc(LibCheck::motionRequest);
    }

    bool isDone(const WalkToTarget&) const override
    {
        return theMotionInfo.motion == MotionRequest::walk;
    }
};

MAKE_SKILL_IMPLEMENTATION(WalkToTargetImpl);
```

Figure 5.10. esempio di implementazione skill WalkToTarget

Per definire gli argomenti di una skill è necessario modificare il file `Representations/Skills.h`. Lì sono definite tutte le interfacce delle skill.

```

namespace Skills
{
    /* This skill executes the get up engine. */
    SKILL_INTERFACE(GetUpEngine);

    /**
     * This skill executes an in walk kick. To define the arguments of a Skill is necessary to modify the file Skills.h
     * @param walkKick The walk kick variant that should be executed
     * @param kickPose The pose at which the kick should be executed in robot-relative coordinates
     */
    SKILL_INTERFACE(InWalkKick, (const WalkKickVariant&) walkKick, (const Pose2f&) kickPose);

    /**
     * This skill executes a kick.
     * @param kickType The kick motion ID of the kick that should be executed
     * @param mirror Whether the kick should be mirrored
     * @param length The desired length of the kick (i.e. distance that the ball moves)
     * @param armsBackFix Use inverse elbow yaw if backLikeDevil is active (motion must be designed for that)
     */
    SKILL_INTERFACE(Kick, (KickRequest::KickMotionID) kickType, (bool) mirror, (float) length, (bool)(true) armsBackFix);

    /**
     * This skill walks to a target using a path planner.
     * @param speed The walking speed as ratio of the maximum speed in [0, 1]
     * @param target The target pose in absolute field coordinates
     */
    SKILL_INTERFACE(PathToTarget, (float) speed, (const Pose2f&) target);
}

```

Figure 5.11. file Representations/Skills.h

Cards

Una Card (Carta) è un componente del behavior che associa le azioni alle condizioni sotto le quali dovrebbero essere eseguite. Mentre le skill eseguono direttamente le richieste, le card possono decidere autonomamente se essere eseguite. Il seguente esempio illustra questo: La skill GoToBallAndKick richiede una posa di calcio e un tipo di calcio come parametri. Non decide se e dove calciare. Al contrario, la KickAtGoalCard valuta se è possibile farlo e poi chiama la skill GoToBallAndKick con i parametri appropriati. Naturalmente, la skill GoToBallAndKick è anche chiamata da altre card.

Le CARDS implementano ogni altro livello del behavior graph che non siano le foglie, e possono chiamare skills o altre cards creando gerarchie fra di loro. Per costruire gerarchie tra cards, il sistema utilizza i cosiddetti **mazzi(Deck)** e **Dealer**. I mazzi sono collezioni di carte da cui un dealer può scegliere e estrarre una carta e attivarla. I deck sono anch'essi gerarchici, e possono avere la proprietà “sticky” che portano ad avere un comportamento leggermente diverso nel dealer: il dealer prima di continuare a pescare nel mazzo, verifica se le postcondizioni della card del frame precedente siano state soddisfatte o no, e se in caso negativo, viene dinuovo runnata la carta precedente finchè non si risolvono le postcondizioni. Questo significa che la card precedentemente selezionata può continuare a funzionare anche se ci sono card di priorità più alta che possono essere eseguite al posto suo.

La struttura delle CARDS è fatta in modo tale che ogni card deve sovrascrivere almeno tre metodi dalle superclassi: **Preconditions()**, **Postconditions()**, **Execute()** e **Reset()**. Le precondizioni specificano le condizioni sotto le quali la carta può essere attivata. Le postcondizioni specificano le condizioni necessarie per uscire dalla carta. **Execute** contiene codice CABSL ed esegue effettivamente la carta. Reset resetta il deck in cui si trova la carta attualmente.

DEFINIZIONE CARD:

Ogni Card deve essere definita all'interno di un mazzo. Le definizioni dei mazzi si trovano nella cartella Config/Scenarios/Default/BehaviorControl e una Card può appartenere a piu' diversi mazzi, quindi posso spostare o escludere una carta da un deck all' altro in base alla situazione di gioco in cui mi trovo.

Ogni Card può chiamare altre card. Per fare ciò, deve caricare i possibili deck come parametri e poi chiamare l'esecuzione di sub-card da questi. Questo tipo di card necessita di un file .cfg specifico per la definizione dei deck disponibili. Le cards

terminali sono quelle che non chiamano altre sub-card ma compongono skills solo per eseguire comportamenti. A differenza delle skills, le card non possono modificare le rappresentazioni di output (per quello dovrebbero chiamare le skill), sono invocate in modo diverso dalle skill (non tramite la macro CALLS) e non dispongono dei metodi isDone e isAborted.

```
CARD(KickAtGoalCard,
{
    CALLS(GoToBallAndKick),
    REQUIRES(BallModel),
    REQUIRES(RobotPose),
    DEFINES_PARAMETERS(
    {
        /**
         * Distance of the ball to the goal to enable kicking.
         */
        | (float)(3000.f) maxGoalDistance,
    }),
});

class KickAtGoalCard : public KickAtGoalCardBase
{
    bool preconditions() const override
    {
        return (theRobotPose * ballPosition - Vector2f(4500.f, 0.f)).norm() < maxGoalDistance;
    }
    bool postconditions() const override
    {
        return (theRobotPose * ballPosition - Vector2f(4500.f, 0.f)).norm() > maxGoalDistance + 1000.f;
    }
    void execute() override
    {
        const Pose2f kickPose = getKickPoseFromBallAndDirection(ballPosition,
            (theRobotPose.inversePose * Vector2f(4500.f, 0.f) - ballPosition).angle(), KickType::walkForwards);
        theGoToBallAndKickSkill(kickPose, KickType::walkForwards);
    }
    const Vector2f& ballPosition = theBallModel.estimate.position;
};

MAKE_CARD(KickAtGoalCard)
```

Figure 5.12. Esempio di Card KickAtGoalCard

```
ownKickoff = {
    sticky = false;
    cards = [
        CodeReleasePositionForKickOffCard
    ];
};
opponentKickoff = {
    sticky = false;
    cards = [
        CodeReleasePositionForKickOffCard
    ];
};
ownFreekick = {
    sticky = false;
    cards = [
        CodeReleaseKickAtGoalCard
    ];
};
opponentFreekick = {
    sticky = false;
    cards = [
        CodeReleaseKickAtGoalCard
    ];
};
normalPlay = {
    sticky = false;
    cards = [
        CodeReleaseKickAtGoalCard
    ];
};
dummyDummieCards = {
    sticky = false;
    cards = [
        DummyTwoCard
    ];
};
```

Figure 5.13. Esempio di mazzi definiti nel file Config/Scenarios/Default/BehaviorControl/gameplayCard.cfg

5.4 Motion Control

Il controllo del movimento nei robot bipedi è suddiviso in diversi moduli, quali la camminata, il calcio da fermo, il calcio in movimento e il rialzarsi. Durante la camminata o la stazione eretta, moduli aggiuntivi gestiscono la testa e le braccia su richiesta del comportamento.

I robot bipedi presentano la sfida di essere instabili e di cadere facilmente. Il Team ha basato l'approccio alla camminata in questo modo: sollevando alternativamente ciascuna gamba, il robot oscilla naturalmente lateralmente senza movimenti effettivi dell'anca, permettendo alla gamba sollevata di muoversi sopra il terreno. L'istante in cui il piede sollevato ritorna a terra, esso è rilevabile dai sensori di pressione del piede, e ciò consente di sincronizzare il ciclo di camminata modellato con lo stato reale, creando un tipo di ciclo di feedback. Infine, per mantenere il robot in posizione eretta, la misurazione del giroscopio attorno all'asse di beccheggio è filtrata e aggiunta direttamente all'angolo di giunzione del piede di richiesta che si trova a terra. La stabilità viene poi incrementata tramite l'aggiustamento della lunghezza dei passi.

Calciare con robot bipedi è difficile perché essi devono sostenere il loro peso su un piede per un periodo prolungato. Essi devono oscillare l'altro piede non a terra e al tempo stesso devono evitare di cadere. Nel nostro software, questi calci sono descritti utilizzando curve di Bezier e sono stabilizzati da semplici controllori di equilibrio. Calci brevi e di media potenza possono essere eseguiti durante la camminata, mentre per calci più forti, il robot deve rimanere fermo per ottenere calci più precisi. Anche rialzarsi è impegnativo, poiché deve funzionare al primo tentativo ma deve essere anche il più veloce possibile. Questi movimenti sono attualmente basati su keyframe. Ci sono due problemi che rendono difficile il rialzarsi. In primo luogo, anche i più piccoli dislivelli nel pavimento o spinte da altri robot possono far cadere nuovamente i robot durante il loro rialzamento. In secondo luogo, alcune giunture delle gambe tendono a bloccarsi per un breve periodo durante questa fase critica, che è la ragione principale dei tentativi falliti di rialzata. Per mantenere i robot il più funzionali e sicuri possibili, essi assumono una posa protettiva non appena la caduta diventa inevitabile. Inoltre, anche solo stando fermi, le loro giunture sono regolate per mantenere una bassa temperatura e corrente. Questi due approcci riducono notevolmente il numero di robot danneggiati e di giunture surriscaldate.

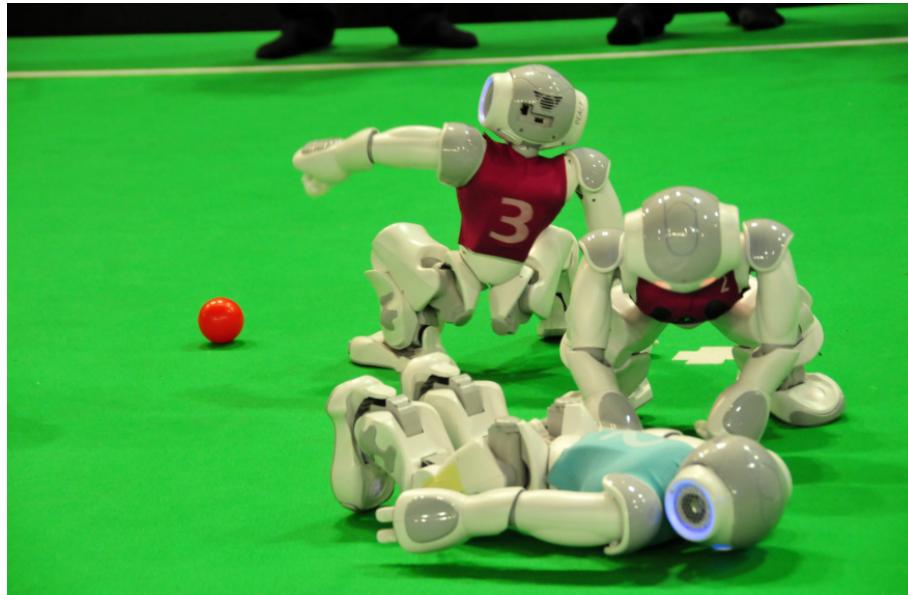


Figure 5.14

Chapter 6

Task assegnata: robot difensore intercettatore

Una volta studiata tutta l'architettura b-human, siamo pronti per risolvere la task assagnata.

6.1 Problema

La task che mi hanno assegnato nella partecipazione dello sviluppo del codice per il team di Robocup SPQR consiste nello sviluppo di un behavior per il robot calciatore difensore che deve intercettare e stoppare il pallone che attraversa il campo nelle vicinanze del robot stesso. Oltre a ciò, il robot deve calcolare la traiettoria a partire da pochi frame della palla individuata, proiettarla nel campo e muoversi fino al punto di intercettazione.

Una volta studiata tutta l'architettura B-Human su cui si basa il codice del team SPQR ed averci preso confidenza per poterci navigare in tranquillità, il quale è stato il lavoro piu' impegnativo del progetto, si può capire che il problema si divide in 2 parti: la parte comportamentale del robot, di cui mi sono occupato io, e la parte fisica della palla, ossia il calcolare il punto nel campo dove la palla può essere intercettata, di cui se ne è occupato il mio collega Emilio Leo. Ci siamo quindi divisi così i task, e dopo che entrambi abbiamo fatto la rispettiva parte, il problema finale era il combinare i 2 lavori per creare un prodotto finito funzionale testabile e simulabile sull' applicazione SimRobot.

Di conseguenza le task erano le seguenti:

- io dovevo programmare una carta per descrivere il comportamento di un robot procinto a intercettare e stoppare una palla in movimento, carta che poi avrei dovuto inserire in un mazzo da cui il dealer la avrebbe selezionata e attivata;
- emilio doveva programmare un modulo che avrebbe fornito alla mia carta la rappresentazione che conteneva la posizione nel campo dove il robot sarebbe dovuto andare per intercettare la palla e stopparla.

6.2 Approccio e risoluzione

6.2.1 Orientamento nella directory

Una volta scaricato l'intero framework e aver compilato e installato le componenti necessarie per runnarlo nel pc (guida github <https://github.com/SPQRTeam/spqr2023>), ho dovuto esplorare tutto il codice e capire dover inserire i vari file che dovrò creare. Le carte da cui potevo prendere spunto si trovano nella directory Src/Modules/BehaviorControl/BehaviorControl/Cards/CodeRelease, di conseguenza dovrò inserire la mia carta in questa cartella. Le varie Skills che dovrò chiamare nella mia carta scopro poi che sono situate in Src/Representations/BehaviorControl/ Skills.h, dove vengono dichiarate tutte le skill possibili che usereme. Da notare che ogni Skills è commentata con l'azione effettiva che esegue ogni skills, il che aiuterà molto ad orientarci nella scelta delle skills senza dover andare a studiare la implementazione specifica di ognuna.

```
namespace Skills
{
    /**
     * This skill turns off the joints of the robot.
     */
    SKILL_INTERFACE(PlayDead);

    /**
     * This skill makes the robot stand.
     * @param high Whether the knees should be stretched
     */
    SKILL_INTERFACE(Stand, (bool)(false) high);

    /**
     * This skill walks with a specified speed.
     * @param speed The walking speed in radians/s for the rotation and mm/s for the translation
     */
    SKILL_INTERFACE(WalkAtAbsoluteSpeed, (const Pose2f&) speed);

    /**
     * This skill walks with a specified speed relative to the configured maximum.
     * @param speed The walking speed as ratio of the maximum speed in [0, 1]
     */
    SKILL_INTERFACE(WalkAtRelativeSpeed, (const Pose2f&) speed);

    /**
     * This skill walks to a (relative) target.
     * @param target The target pose in robot-relative coordinates
     * @param speed The walking speed as ratio of the maximum speed in [0, 1]
     * @param obstacleAvoidance The obstacle avoidance request
     * @param keepTargetRotation Whether the target rotation should be headed for all the time (instead of allowing motion to plan it)
     */
    SKILL_INTERFACE(WalkToPose, (const Pose2f&) target, (const Pose2f&) speed, (const MotionRequest::ObstacleAvoidance&) obstacleAvoid

    /**
     * This skill walks to the ball and kicks it.
     * @param targetDirection The (robot-relative) direction in which the ball should go
     * @param kickType The type of kick that should be used
     */
}
```

Figure 6.1. file Skills.h

Grazie poi ai consigli del Prof. Vincenzo Suriani, il Software Development Leader del team SPQR, che ci ha seguito e aiutato nel percorso di tutta la progettazione delle task, ho scoperto la directory dei mazzi e del Dealer che dovrò poi modificare. il file Config/Scenarios/Default/BehaviorControl/gameplayCard.cfg contiene i deck rilevanti, mentre in Src/Modules/BehaviorControl/BehaviorControl/Cards/System /gameplayCard.cpp è presente il dealer che avrebbe pescato la mia carta. Prima di spiegare come ho risolto la mia task, bisogna spiegare le parti di codice che il mio collega ha dovuto modificare nel framework per implementare il suo rispettivo task, il quale è necessario per il corretto funzionamento del mio task.

6.2.2 Rappresentazione Intercept-ball

Il mio collega Emilio Leo nel completare il suo task ha apportato le seguenti modifiche al codice:

- l'implementazione del modulo Intercept-ballProvider.cpp e definito la sua interfaccia Intercept-ballProvider.h nella directory Src/Modules/BehaviorControl/BehaviorControl . Questo modulo, tramite i dati che il robot ricava dalla individuazione della palla e della sua velocità attuale, calcola usando formule sul moto cinematico della palla il punto nel campo in cui il robot dovrà andare per intercettare in tempo la palla e stopparla, ossia fornisce alla mia carta la rappresentazione Intercept-ball, che indica tale posizione nel campo.
- inserimento del codice della rappresentazione fornita dal modulo, ossia Intercept-ball.h situata in Src/Representations/BehaviorControl/Intercept-ball.h

```
#pragma once
#include "Tools/Streams/AutoStreamable.h"
#include "Tools/Streams/Enum.h"
#include "Tools/Math/Eigen.h"
STREAMABLE(Intercept_ball,{,
    (Vector2f) (Vector2f::Zero()) robot_speed,//vettore velocità del robot
    (Vector2f) (Vector2f::Zero()) position_intercept,//vettore spostamento per arrivare all'intercetto del pallone
    (float) time_to_ball,//tempo impiegato per raggiungere il pallone
    (float) angle_to_position_intercept, //angolo di intercettazione
});
```

Figure 6.2. Intercept-ball.h

```
#include "Representations/Modeling/RobotPose.h"
#include "Tools/Module/Module.h"
#include "Representations/BehaviorControl/Intercept_ball.h"
#include "Representations/Modeling/BallModel.h"
#include "Representations/Configuration/BallSpecification.h"
//Creo Intestazione modulo Intercept_ballProvider
MODULE(Intercept_ballProvider,
{
    ,
    REQUIRES(BallModel),
    REQUIRES(RobotPose),
    REQUIRES(BallSpecification),
    PROVIDES(Intercept_ball),
});

//Creo Classe Modulo Intercept_ballProvider
class Intercept_ballProvider : public Intercept_ballProviderBase
{
private:
    Vector2f position_ball,position_robot,velocity_ball;
public:
    Intercept_ballProvider();
    void update(Intercept_ball &Intercept_ball);
};
```

Figure 6.3. Intercept-ballProvider.h

Figure 6.4. parte 1 Intercept-ballProvider.cpp

Figure 6.5. parte 2 Intercept-ballProvider.cpp

Proseguiamo quindi mostrando come ho progettato la Carta che funziona insieme alla Rappresentazione Intercept-ball.

6.2.3 Carta defgotoball

Prendendo spunto dalle altre carte e seguendo la definizione di carta spiegata nella documentazione, ho implementato una carta che passa tramite 6 stati totali: start, howtoapproach, still, walking, searchBall, seeBall.

Come si passa da uno stato all' altro è descritto dentro la carta tramite codice CABSL. La Figura 6.6 descrive la macchina a stati finiti che descrive il funzionamento della carta defgotoball.cpp . Partendo dallo stato iniziale start, passato un tot. di tempo iniziale definito da initialWaitTime, si transiziona allo stato howtoapproach che è lo stato principale, che può esser visto come l'incrocio stradale circolare dove ogni strada parte e torna. Infatti a seconda dei dati forniti dalle Rappresentazioni indicati

nei REQUIRES, dallo stato howtoapproach mi sposto nei vari stati rimanenti (apparte lo stato iniziale start). Dai altri stati poi potrò tornare nello stato howtoapproach, oppure rimanere nei stessi stati finchè qualche condizione di transizione diventa vera e quindi transizioni di stato. In ogni stato la Carta esegue delle azioni che sono le Skills di basso livello definite in Src/Representations/BehaviorControl/ Skills.h .

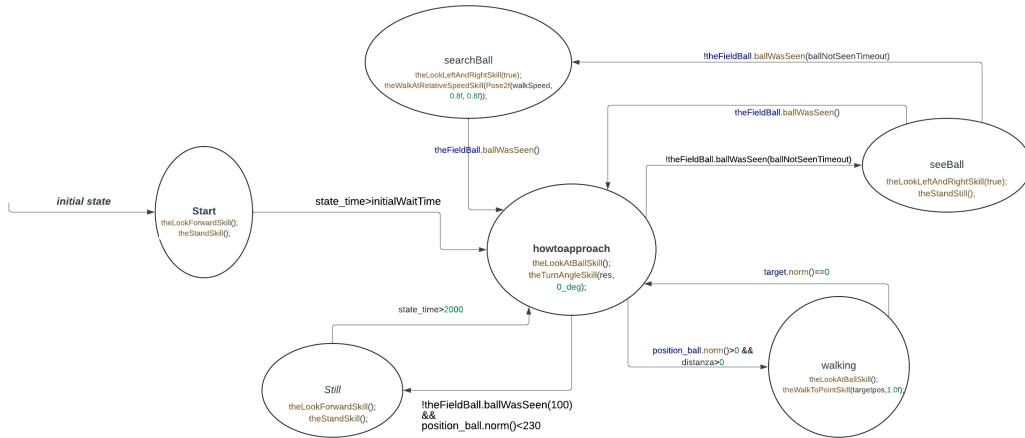


Figure 6.6. Macchina a stati finiti che descrive il behavior di defgottoball

```

6 #include <cmath>
7 #include <string>
8 #include <memory>
9 #include <assert>
10 #include <thread>
11 #include <vector>
12 #include <functional>
13 #include <iostream>
14 #include <algorithm>
15 #include <limits>
16 #include <utility>
17 #include <map>
18 #include <list>
19 #include <functional>
20 #include <functional>
21 #include <memory>
22 #include <Eigen/Eigen>
23 #include <Eigen/Eigen>
24 #include <Eigen/Eigen>
25 #include <Eigen/Eigen>
26 #include <cmath>
27 #include <assert>
28 #include <assert>
29 #include <assert>
30 #include <assert>
31 #include <assert>
32 #include <assert>
33 #include <assert>
34 #include <assert>
35 #include <assert>
36 #include <assert>
37 #include <assert>
38 #include <assert>
39 #include <assert>
40 #include <assert>
41 #include <assert>
42 #include <assert>
43 #include <assert>
44 #include <assert>
45 #include <assert>
46 #include <assert>
47 #include <assert>
48 #include <assert>
49 #include <assert>
50 #include <assert>
51 #include <assert>
52 #include <assert>
53 #include <assert>
54 #include <assert>
55 #include <assert>
56 #include <assert>
57 #include <assert>
58 #include <assert>
59 #include <assert>
60 #include <assert>
61 #include <assert>
62 #include <assert>
63 #include <assert>
64 #include <assert>
65 #include <assert>
66 #include <assert>
67 #include <assert>
68 #include <assert>
69 #include <assert>
70 #include <assert>
71     : action // fa action finche non transisce a un altro stato
72     {
73         theSaySkill("ok",1.0f);
74         theWalkSkill();
75         theStandSkill();
76     }
77     stateTimeApproach()
78     {
79         if(theLookSkill("ok",1.0f))
80             if(theFieldBall.ballWasSeen(ballNotSeenTimeout))
81                 goto seeBall;
82         Vector3f target = theIntercept_ball.position_intercept;
83         Angle angle_to_ball = theIntercept_ball.angle_to_position_intercept;
84         Vector3f position_ball = theBallModel.estimata.position;
85         Angle restante2d(position_ball.y), position_ball.x();
86         if(restante2d < res || res < endl);
87         Angle zero(0.0f);
88         Angle zero(0.0f);
89         Angle zero(0.0f);
90         Angle zero(0.0f);
91         if(theFieldBall.ballWasSeen(100) && position_ball.norm()>230)
92             goto still;
93         float distanza=target.norm();
94         if(distanza>target.norm())&& distanza<0) goto walking;
95         action
96         {
97             theLookSkill();
98             theLookSkill();
99             theLookSkill();
100            theLookSkill();
101            theLookSkill();
102            theLookSkill();
103            theLookSkill();
104            theLookSkill();
105            theLookSkill();
106            theLookSkill();
107            theLookSkill();
108            theLookSkill();
109            theLookSkill();
110            theLookSkill();
111            theLookSkill();
112            theLookSkill();
113            theLookSkill();
114            theLookSkill();
115            theLookSkill();
116            theLookSkill();
117            theLookSkill();
118            stateWalking();
119            transition|
120            if(state_time>2000)
121                | gets howtoapproach;
122            |
123            action|
124            {
125                theSaySkill("ok",1.0f);
126                theWalkSkill();
127                theStandSkill();
128            stateWalking();
129            transition|
130            if(theFieldBall.ballWasSeen())
131                | gets howtoapproach;

```

Figure 6.7. defgottoball.cpp

In seguito spieghiamo pezzo per pezzo la Carta implementata defgottoball.cpp spiegando pure le azioni fatte dalle skills chiamate e quando e come transiziona di stato in stato.

```

2  * @file defgotoball.cpp
3  *
4  * @author jin giacomo
5  */
6 #include <cmath>
7 #include "Representations/BehaviorControl/FieldBall.h"
8 #include "Representations/BehaviorControl/Skills.h" //skill so dichiarate tutte qua
9 #include "Representations/Configuration/FieldDimensions.h"
0 #include "Representations/Modeling/RobotPose.h"
1 #include "Tools/BehaviorControl/Framework/Card/Card.h"
2 #include "Tools/BehaviorControl/Framework/Card/Cabs1Card.h"
3 #include "Tools/Math/BHMath.h"
4 //#include "Representations/BehaviorControl/Libraries/LibDefender.h"
5 #include "Representations/BehaviorControl/Libraries/LibMisc.h"
6 #include "Representations/BehaviorControl/Intercept_ball.h"
7 #include <iostream>
8 #include "Representations/MotionControl/MotionInfo.h"
9 #include "Representations/MotionControl/MotionRequest.h"
0 #include "Representations/Sensing/FallDownState.h"
1 //#include "Tools/BehaviorControl/Framework/Skill/Cabs1Skill.h"
2 #include <Eigen/Geometry>
3 using Line2 = Eigen::Hyperplane<float,2>;

```

Figure 6.8. iniziamo la carta includendo tutte le librerie necessarie.

```

24 CARD(defgotoball,
25 {
26     CALLS(Activity),
27     CALLS(GoToBallAndKick), // Sono le skills che chiama
28     CALLS(LookForward),
29     CALLS(Stand),
30     CALLS(WalkAtRelativeSpeed),
31     CALLS(LookLeftAndRight),
32     CALLS(TurnAngle),
33     CALLS(Say),
34     CALLS(WalkToPoint),
35     CALLS(LookAtBall),
36     CALLS(GoToBallHeadControl),
37     CALLS(TurnToPoint),
38     CALLS(LookAtGlobalBall),
39     REQUIRES(FieldBall),
40     REQUIRES(FieldDimensions), //rappresentazioni richieste
41     REQUIRES(RobotPose),
42     //REQUIRES(LibDefender),
43     REQUIRES(LibMisc),
44     REQUIRES(Intercept_ball),
45     REQUIRES(BallModel),
46     DEFINES_PARAMETERS(
47     {
48         (float)(1.0f) walkSpeed,
49         (int)(1000) initialWaitTime,
50         (int)(3000) ballNotSeenTimeout,
51     },
52
53 });

```

Figure 6.9. come per definizione di carta, definisco tramite CALLS le skills chiamate nella carta, tramite REQUIRES le rappresentazioni richieste, e tramite DEFINES-PARAMETERS parametrizza la carta. Da notare REQUIRES(Intercept-ball) è la rappresentazione fornita dal mio collega emilio discussa prima.

```

54     class defgotosball : public defgotosballBase
55     {
56         bool preconditions() const override{
57             return true;
58         }
59         bool postconditions() const override{
60             return true;
61         }
62     }

```

Figure 6.10. seguiamo e definiamo la classe base defgotosballBase da cui deriviamo la nostra effettiva classe defgotosball; le precondizioni e postcondizioni sono lasciate vuote per comodità poichè nel deck in cui inseriremo questa carta essa sarà l'unica pescabile, e quindi è attivata per forza a prescindere delle precondizioni o postcondizioni.

```

63     option
64     theActivityskill(Behaviorstatus::Striker);
65     initial_state(start){
66         transition{
67             if(state_time>initialWaitTime)
68                 goto howtoapproach;
69         }
70         action // fa action finche non transisce a un altro stato
71     {
72         // theSaySkill("ok",1.0f);
73         theLookForwardSkill();
74         theStandSkill();
75     }
76 }
77

```

Figure 6.11. iniziamo definendo il comportamento della carta tramite il codice CABSIL: lo stato iniziale dopo un tempo initialWaitTime transiziona automaticamente allo stato howtoapproach, l'azione compiuta in questo stato è solo quella di stare in piedi e guardare avanti.

```

78     state(howtoapproach){
79         transition{
80             // theSaySkill("ok",1.0f);
81             if(!theFieldBall.ballWasSeen(ballNotSeenTimeout))
82                 goto seeBall;
83             Vector2f target = theIntercept_ball.position_intercept;
84             Angle angolo(theIntercept_ball.angle_to_position_intercept);
85             Vector2f position_ball = theBallModel.estimate.position;
86             Angle res(atan2f(position_ball.y(), position_ball.x()));
87             //cout << "Angle" << res << endl;
88             Angle zero(0.0f);
89             Angle zero(-0.2f);
90             Angle zero(0.2f);
91             if(!theFieldBall.ballWasSeen(100) && position_ball.norm()<230)
92                 goto still;
93                 float distanza=target.norm();
94                 if(position_ball.norm(>0 && distanza>0) goto walking;
95             }
96         action
97     {
98         theLookAtBallSkill();
99         Vector2f position_ball = theBallModel.estimate.position;
100         Angle res(atan2f(position_ball.y(), position_ball.x()));
101         theTurnAngleSkill(res, 0_deg);
102     }
103 }

```

Figure 6.12. dallo stato howtoapproacch posso transizionare a piu' stati, ciò dipende principalmente dalla rappresentazione FieldBall, ossia dipende dal robot se ha individuato la palla o no. Se non l'ha individuato, va nello stato seeBall; se l'ha vista ma è troppo lontana per essere intercettata, va nello stato still; se l'ha vista ed è vicina abbastanza, va nello stato walking. Le azioni eseguite in questo stato sono: girare la testa verso la palla, e ruotare il corpo del robot affinchè sia di fronte alla palla.

```

104     state(walking){
105         transition{
106             | Vector2f target = theIntercept_ball.position_intercept;
107             | if(target.norm()==0) goto howtoapproach;
108         }
109         action{
110             {
111                 Vector2f target = theIntercept_ball.position_intercept;
112                 theLookAtBallSkill();
113                 Angle angolo(theIntercept_ball.angle_to_position_intercept);
114                 Pose2f targetpos(target.x(),target.y());
115                 theWalkToPointsSkill(targetpos,1.0f);
116             }
117         }

```

Figure 6.13. Nello stato walking, se il robot ha stoppato la palla, allora ritorna nello stato howtoapproach. Le azioni eseguite in questo stato sono: girare la testa verso la palla, e camminare verso il punto di intercettazione data usando pathplanning intelligente.

```

118     state(still){
119         transition{
120             | if(state_time>2000)
121             | goto howtoapproach;
122         }
123         action{
124             | theLookForwardSkill();
125             | theStandSkill();
126         }
127     }

```

Figure 6.14. Nello stato still il robot guarda avanti e sta in piedi fermo, e passato un tot di tempo transiziona nello stato howtoapproach

```

128     state(seeBall){
129         transition{
130             | if(theFieldBall.ballWasSeen())
131             | goto howtoapproach;
132             | if(!theFieldBall.ballWasSeen(ballNotSeenTimeout))
133             | goto searchBall;
134         }
135         action{
136             | theLookLeftAndRightSkill(/* startLeft: */ true, /* maxPan: */ 30_deg, /* tilt: */ 10_deg, /* speed: */ 15_deg);
137             | theStandSkill();
138         }
139     }

```

Figure 6.15. Nello stato seeBall il robot sta fermo e muove la testa alternativamente a destra e a sinistra per individuare la palla, se la individua va nello stato howtoapproach, sennò va nello stato searchBall.

```

140     state(searchBall){
141         transition{
142             | if(theFieldBall.ballWasSeen())
143             | goto howtoapproach;
144         }
145         action{
146             | theLookLeftAndRightSkill(/* startLeft: */ true, /* maxPan: */ 30_deg, /* tilt: */ 10_deg, /* speed: */ 15_deg);
147             | | theWalkAtRelativeSpeedSkill(Pose2f(walkSpeed, 0.8f, 0.8f));
148         }
149     }
150     | |
151 };
152 MAKE_CARD(defgotoball);
153

```

Figure 6.16. Nello stato searchBall il robot si sposta e cammina avanti ad una certa velocità fissata e muove la testa alternativamente a destra e a sinistra per individuare la palla, se la individua va nello stato howtoapproach. Infine il comando MAKE-CARD per implementare la carta

Infine per farsi che la carta venga chiamata, la inseriamo nel mazzo striker e la rendiamo l'unica carta del mazzo (proprietà sticky del mazzo non è attiva)

```

opponentPenaltyKick = {
    sticky = false;
    cards = [
        CodeReleasePositionForKickOffCard
    ];
};

striker = {
    sticky = false;
    cards = [
        #OpponentKickoffCard,
        #StrikerOppKickInCard,
        defgotoball
        #BasicStrikerCard
    ];
};
goalie = {
    sticky = false;
    cards = [
        GoalieCoreCard
    ];
};

```

Figure 6.17. situato in Config/Scenarios/Default/BehaviorControl/gameplayCard.cfg

E modifichiamo il Dealer associato affinchè peschi solo carte dal mazzo di striker attivando forzatamente la nostra carta.

```

64 }else{
65     dealer.deal(striker)->call();
66     /*if(thePlayerRole.role == PlayerRole::striker){
67         dealer.deal(striker)->call();
68         setState("striker");
69     }else if(thePlayerRole.role == PlayerRole::supporter){
70         dealer.deal(supporter)->call();
71         setState("supporter");
72     }else if(thePlayerRole.role == PlayerRole::jolly){
73         dealer.deal(jolly)->call();
74         setState("jolly");
75     }else if(thePlayerRole.role == PlayerRole::defender){
76         dealer.deal(defender)->call();
77         // dealer.deal(defgotoball)->call();
78         setState("defender");
79     }else{
80         dealer.deal(goalie)->call();
81         setState("goalie");
82     }
83 */
84 }
85

```

Figure 6.18. situato in Src/Modules/BehaviorControl/BehaviorControl/Cards/System/gameplayCard.cpp

Chapter 7

Risultati

Per testare se tutto ciò funziona va lanciato SimRobot e bisogna simulare il tutto nell'applicazione per vedere se il robot riesce effettivamente a intercettare e stoppare la palla. Prima di ciò va creata uno scenario apposito per simulare la situazione, ossia il file Intercept-ball.ros2 che inseriremo nella directory Config/Scenes .

```
<Simulation>
  <Include href="Includes/NaoV0HZ5.rsl2"/>
  <Include href="Includes/one_robot.rsl2"/>
  <Include href="Includes/Ball2016SP1.rsl2"/>
  <Include href="Includes/Field2020SP1.rsl2"/>

  <Scene name="RoboCup" controller="SimulatedNao" stepLength="0.012" color="rgb(65%, 65%, 70%)"
    ERP="0.8" CFM="0.001" contactSoftERP="0.2" contactSoftCFM="0.005">
    <Light z="9m" ambientColor="rgb(50%, 50%, 50%)"/>

    <Compound name="teamColors">
      <Appearance name="black"/>
      <Appearance name="blue"/>
    </Compound>

    <Compound ref="robots"/>
    <Compound name="balls">
      <Body ref="ball">
        <Translation z="1m"/>
      </Body>
    </Compound>

    <Compound ref="field"/>
  </Scene>
</Simulation>
```

Figure 7.1. file Intercept-ball.ros2

Una volta apportate tutte le modifiche al codice, compiliamo di nuovo il codice tramite i comandi:

- NO-CLION=true ./Make/Linux/generate
- ./Make/Linux/compile Debug SimRobot

Ora che abbiamo il codice compilato (possibilmente senza errori) siamo pronti a lanciare l'applicazione SimRobot che si trova nella directory Build/Linux/SimRobot/Develop/. Una volta aperta la pagina iniziale di SimRobot selezioniamo su File il nostro Scenario Intercept-ball.ros2 e runniamolo.

per far partire la simulazione inseriamo molto semplicemente i comandi

- gc set
- gc ready
- gc playing

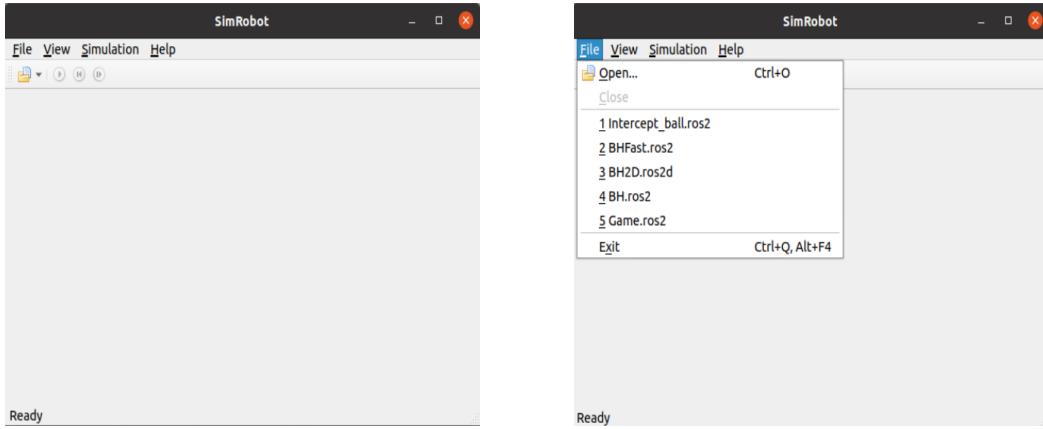


Figure 7.2. pagina iniziale di SimRobot, scegliamo il nostro scenario apposito

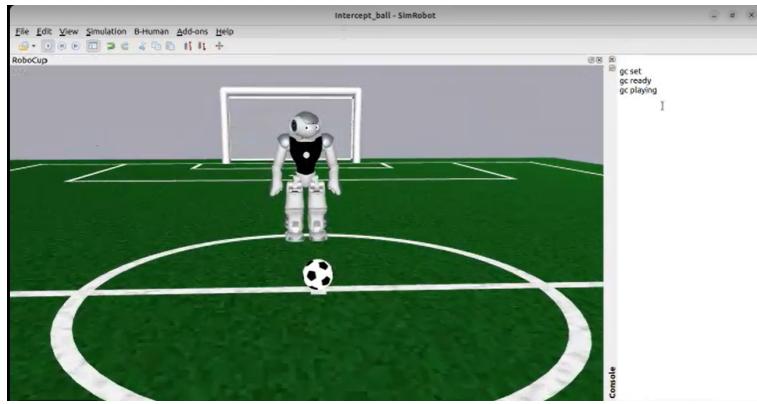


Figure 7.3

7.1 Casistiche verificate di intercettazione

Iniziamo a testare quindi tutti i casi che il robot può incontrare e verifichiamo che:

- se lanciamo la palla molto velocemente e molto lontano dal robot, il robot non si muove poiché la rappresentazione Intercept-ball lo informa che la palla non è possibilmente intercettabile visto che il robot si può muovere al massimo ad una velocità limite molto bassa. Una volta che la palla esce fuori dalla visione del robot, esso va nei stati seeBall e/o searchBall per cercare di reindividuarla.
- se lanciamo la palla ad una velocità relativamente bassa ai piedi del robot o in zone vicine visibili dal robot, esso rileva che la palla è intercettabile, va in stato walking e riesce ad aggiustare la sua angolatura del corpo tale che stoppa e ricevere la palla di fronte a sè.
- se lanciamo la palla ad una velocità maggiore ma comunque non troppo veloce poco a destra o a sinistra rispetto ai piedi del robot, esso riesce a fare dei passi laterali e a intercettare la palla ogni volta che la rappresentazione Intercept-ball lo informa che la palla è intercettabile. Una volta intercetta lo stato walking ed finisce esso torna nello stato howtoapproach.



Figure 7.4. una volta partito possiamo selezionare la casella behavior per vedere i stati in cui sta attualmente e i valori delle vari rappresentazioni richieste nella carta.

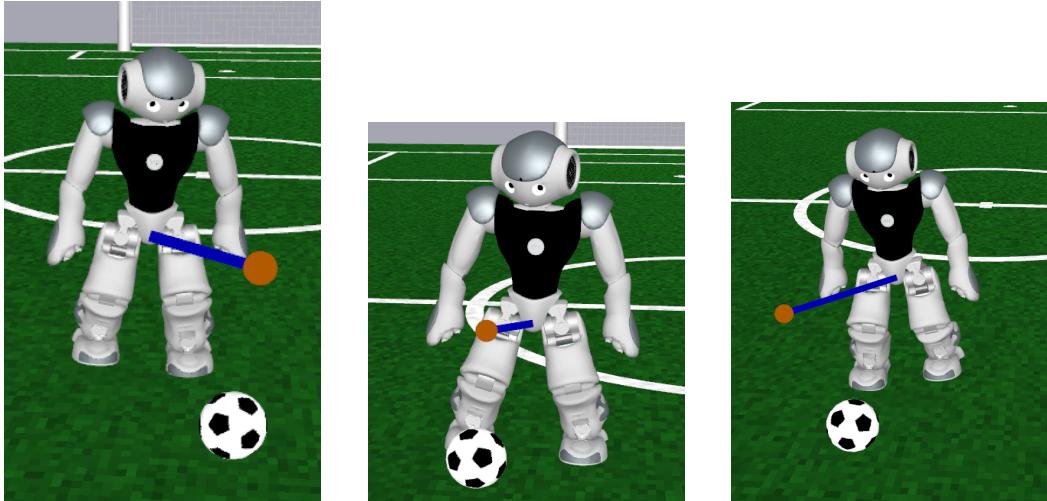


Figure 7.5. qua vengono mostrati i casi in cui la palla viene intercetta, il segmento che parte dal bacino del robot indica la sua angolazione attuale attivabile tramite graphic debug, e si può verificare che il robot è sempre rivolto verso la palla grazie alle skills TurnAngle e LookAtBall. L'essere sempre con la testa e/o il corpo rivolto verso la palla è una parte fondamentale per l'intercettazione, perché permette al robot di visualizzare costantemente la palla e quindi può calcolare meglio il punto di intercettazione.

Chapter 8

Conclusione

Raggiunta la fine di questo progetto, ho realizzato che la complessità strutturale e l'impegno e sforzo necessario per portare avanti un progetto del genere è davvero significante. La Robocup porta uno sviluppo grandissimo in ambito di intelligenza artificiale; team tedeschi come la B-Human che rilasciano open source il loro codice ogni anno hanno dimostrato di aver portato molto avanti lo sviluppo del software robotico, che applicato in altri ambiti oltre al calcio sicuramente può riscontrare risultati positivi. Purtroppo il compito che mi è stato assegnato sfiora soltanto la complessità dell'architettura in gioco, e la difficoltà che ho incontrato nel navigare in questo codice mantenuto da anni e da tanti altri sviluppatori molto piu' esperti è stata veramente enorme. Dopo circa 2 mesi di lavoro io e il mio collega emilio leo alla fine siamo riusciti comunque ad ottenere un risultato accettabile e funzionale. Se avessimo avuto a disposizione piu' tempo, c'erano molte aree nel codice mio che potevano essere sviluppate meglio o piu' approfonditamente. Ad esempio un possibile compito seguente che il prof mi poteva assegnare è l'implementazione delle precondizioni e postcondizioni nella carta, andava studiato pure vari i scenari di gioco con piu' robot in campo. Il Dealer quindi doveva capire quando passare da un mazzo ad un altro e studiare quali sarebbero state le condizioni giuste per attivare la mia carta intercettatore. Ciò nonostante reputo di aver guadagnato una esperienza significativa che mi ha reso piu' familiare nel gestire grosse strutture di codice mantenute da piu' anni e da piu' developers.

Bibliography

- [1] Blanche P.A., Gailly P., et al., “*Volume phase holographic gratings: large size and high diffraction efficiency*”, Optical Engineering, Vol. 43, No.11, November 2004
- [2] Cirasuolo M., et al., “*MOONS Science Report*”, MOONS Document Number: VLT-TRE-MON-14620-0001, Issue: 1.0, 31st January 2013
- [3] European Southern Observatory, <http://www.eso.org>
- [4] storia https://blog.osservatori.net/it_it/storia-intelligenza-artificiale
- [5] robocup https://www.robocup.org/a_brief_history_of_robocup
- [6] b-human2023 <https://docs.b-human.de/coderelease2023/>
- [7] b-human2019 <https://docs.b-human.de/coderelease2019/>
- [8] cabsl https://link.springer.com/chapter/10.1007/978-3-030-00308-1_11
- [9] spqr <http://spqr.diag.uniroma1.it/>
- [10] gitcompile <https://github.com/SPQRTeam/spqr2023>