Competizione tra due squadre di robot utilizzando TeamBots e sGOLOG

~

Tesina di Robotica Cognitiva

Anno Accademico 2003/2004

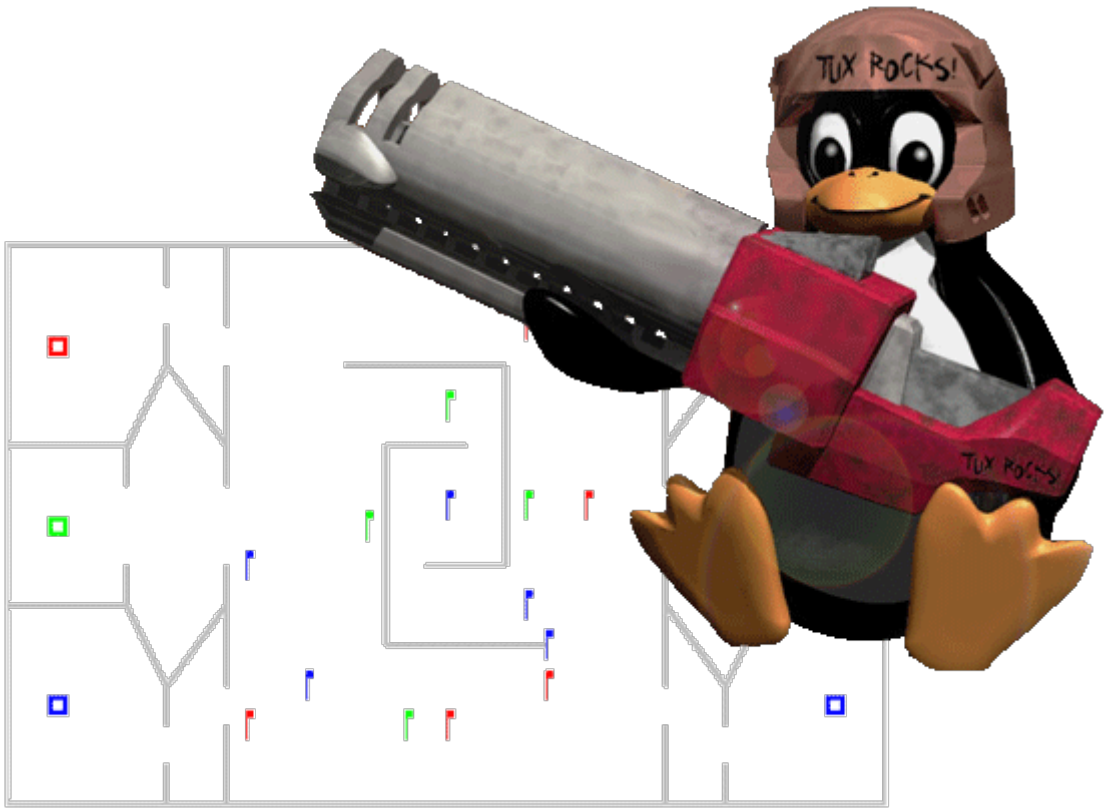
~

Universita´ degli Studi di Palermo

~

Allievi: Salvatore La Bua e Riccardo Marfisi

Nome squadra: TUX



Indice

Introduzione

In questa tesina verra´ trattato un esempio di competizione tra due squadre, ciascuna formata da due robot. La competizione consiste nel riuscire a raccogliere delle bandierine colorate (rosse, verdi e blu) in numero maggiore rispetto a quelle raccolte dalla squadra avversaria. Le bandierine recuperate verranno poi depositate, in base al colore, nei rispettivi contenitori (bin); ciascuna squadra dovra´ partire dal proprio lato del campo da gioco e dovra´ depositare le bandierine nei bin disposti in prossimita´ del bordo opposto del campo.

La competizione verra´ simulata al calcolatore con l'ausilio di software cooperanti tra loro, quali *TeamBots*, in Java, e´ il simulatore per le azioni del robot e l'interprete *ECL^{i}PS^e* che permette l'interfacciamento del TeamBots con *sGOLOG*, linguaggio di programmazione logica derivato dal GOLOG ed avente basi nel *PROLOG*, in grado di introdurre il concetto di non determinismo per la scelta delle azioni da far compiere ai robot. E´ possibile quindi implementare il comportamento dei robot (interamente o solo in parte) tramite una sequenza di azioni sGOLOG ed utilizzare una macchina a stati finiti (*FSA*) con un minor numero di stati.

1 TeamBots & sGOLOG in generale

L'utilizzo di sGOLOG in questa simulazione, ha il grande vantaggio di semplificare molto la realizzazione di una macchina a stati finiti per la gestione del comportamento dei robot, e´ infatti sufficiente (come gia´ accennato) anche un unico stato dell'FSA interamente gestito da sGOLOG stesso.

TeamBots sfrutta le potenzialita´ del linguaggio sGOLOG grazie all'interprete ECL^{i}PS^{e} che e´ in grado di fornire il supporto necessario, in termini di classi, al corretto interfacciamento tra il TeamBots stesso ed sGOLOG.

L'esempio preso in esame in questa tesina si basa infatti principalmente su un unico stato implementato da un database sGOLOG, comprensivo delle regole e dei fatti necessari allo svolgimento dei compiti assegnati ai robot.

Tramite sGOLOG e´ possibile generare alberi di scelta binari in base allo stato percepito dell'ambiente (mediante *fluenti*) e agire di conseguenza. In tali alberi, chiamati *CAT* (*Conditional Action Trees*) si ottiene una nuova diramazione ogni qual volta viene effettuata l'azione di *sense*, con il relativo *branch_on*, su un certo fluente, il che corrisponde ad intraprendere una certa sequenza di azioni anziche´ un'altra, in base al valore del fluente preso in esame. Ciascuna diramazione termina infine con il nodo obiettivo.

All'interno del sorgente java, vengono correlati i fluenti utilizzati in sGOLOG con le percezioni del robot in TeamBots, in modo da permettere il corretto sviluppo dell'albero di decisione generato dal database sGOLOG. Questa operazione e´ nota con il termine di *matching*.

Inoltre, sempre nel codice java, sono presenti tutti i controlli necessari al superamento degli inconvenienti che potrebbero portare a situazioni tali da compromettere il risultato della gara.

Il database sGOLOG possiede delle asserzioni che permetteranno la gestione del piano in funzione dei fluenti percepiti. L'elenco dei fatti e´ riportato sotto:

```
/* Azioni primitive */
primitive_action(goto(_)).
primitive_action(wander).
primitive_action(go_flag).
primitive_action(open_gripper).
primitive_action(send_message(_)).
primitive_action(wait).
primitive_action(sense(_)).

…
```

Vengono definite le *azioni primitive* che saranno poi utilizzate per calcolare l'albero di scelta sGOLOG

```
…

/* Precondizioni */
poss(goto(_),_).
poss(wander,_).
poss(go_flag,_).
poss(open_gripper,_).
poss(send_message(_),_).
poss(wait,_).
poss(sense(_),_).

…
```

Nel blocco delle *precondizioni*, viene specificato quando un'azione puo´ essere eseguita o meno. Nel nostro caso tutte le azioni possono essere eseguite senza alcuna precondizione; tale condizione e´ rappresentata dall'underscore tra parentesi.

```
…

/* Assiomi di stato successore */
holds(something_visible, do(A,S))        :-    holds(something_visible,S);
                                               A = assm(something_visible,1).

holds(red_flag, do(A,S))          :-    holds(something_visible,S), A = go_flag;
                                        A = assm(red_flag,1).
```

```
holds(green_flag, do(A,S))          :-  holds(something_visible,S), A = go_flag;
                                        A = assm(green_flag,1).

holds(blue_flag, do(A,S))           :-  holds(something_visible,S), A = go_flag;
                                        A = assm(blue_flag,1).

holds(load, do(A,S))          :-    holds(load,S);
                                    A = assm(load,1).

holds(in_red_lobby, do(A,S))  :-    holds(in_red_lobby,S);
                                    A = assm(in_red_lobby,1).

holds(in_green_lobby, do(A,S))  :-  holds(in_green_lobby,S);
                                    A = assm(in_green_lobby,1).

holds(in_blue_lobby, do(A,S))  :-   holds(in_blue_lobby,S);
                                    A = assm(in_blue_lobby,1).

holds(can_go, do(A,S))        :-    (holds(red_flag,S),    A = goto(red_attractor)   );
                                    (holds(blue_flag,S),   A = goto(blue_attractor) );
                                    (holds(green_flag,S), A = goto(green_attractor));
                                    A = assm(can_go,1).
```

Gli *assiomi di stato successore* invece specificano i valori di verita´ dei fluenti utilizzati, precisando inoltre che un fluente e´ vero nello stato successivo, se era vero allo stato precedente e non e´ stata compiuta alcuna azione che lo ha reso false oppure se e´ stata compiuta un'azione che ha portato quel fluente allo stato vero.

2 Competizione

2.1 Obiettivi della competizione

Come gia´ precedentemente introdotto, la competizione in esame si basa sul recupero del maggior numero possibile di bandierine colorate rispetto alla squadra avversaria.

I due robot di ciascuna squadra devono inoltre gestire un semplice meccanismo di comunicazione, descritto in seguito, necessario a far depositare le bandierine raccolte da ciascun robot di ogni squadra, soltanto quando entrambi i robot si trovano davanti alla camera dei bin.

Il comportamento generale di ciascun robot deve essere quello di analizzare il campo di gioco con un'esplorazione casuale, vagando in cerca di una bandierina, non appena il robot ne avvista una, deve dirigersi verso quest'ultima e ne deve identificare il colore, in modo tale da dirigersi quindi verso il relativo bin per la consegna.

Le difficolta´ principali durante l'esplorazione del campo da gioco sono sicuramente la gestione della chiocciola centrale, delle anticamere dei bin (dove possono esservi anche delle bandierine) e di situazioni in cui i robot possono rimanere bloccati a causa di minimi locali.

2.2 Problemi riscontrati e soluzioni adoperate

2.2.1 Gestione della mappa

Innanzi tutto, c'e´ da dire che i robot utilizzano una mappa del campo da gioco, al fine di gestire il comportamento di esplorazione casuale (*wander*) in maniera piu´ efficiente, ovvero escludendo, per quanto possibile, di tornare su zone gia´ scandite ed in cui non e´ stata avvistata alcuna bandierina.

Il codice che permette l'utilizzo della mappa e´ il seguente:

```
private NodeMap                MAP;     // dichiarata come globale, all'inizio. Crea l'oggetto MAP di tipo NodeMap

…

MAP = new NodeMap(8.0, 36.0, 28.0, 0.0, 1.0, 0.6, "Mappa LEFT 0", false, abstract_robot);
```

dove i primi quattro argomenti del costruttore dell'oggetto MAP rappresentano rispettivamente il limite sinistro, destro, superiore e inferiore della mappa rispetto alle coordinate del campo da gioco, il quinto argomento indica la dimensione delle celle (quadrate), il sesto e´ il raggio di appartenenza alla cella, che viene impostato per sicurezza ad un valore leggermente maggiore alla meta´ della dimensione della cella stessa, il settimo e´ il nome della mappa che verra´ visualizzato come titolo della finestra, l'ottavo argomento e´ un flag booleano che indica se la mappa dovra´ essere visibile o meno e l'ultimo argomento e´ il robot che dovra´ utilizzare la mappa.

Tramite l'utilizzo della mappa appena citata, il robot contrassegna le celle su cui e´ gia´ passato come visitate, evitando di ritornarvi.

2.2.2 Gestione delle situazioni di stallo

Qui si puo´ intuire l'insorgenza del primo problema, quello di rimanere bloccato a causa della contrassegnazione di tutte le celle attorno al robot, situazione che si verifica soprattutto in presenza di angoli e/o insenature nel campo.

Il metodo da noi utilizzato per ovviare a tale inconveniente e´ quello di liberare (con il metodo *setCellState* dell'oggetto MAP) le celle contrassegnate come occupate dopo un certo tempo in cui il robot permane in una determinata posizione, effettuando ad ogni *takestep* (l'unita´ minima di tempo della simulazione) il confronto della posizione attuale con quella all'istante precedente e considerando una certa tolleranza (liberta´ di movimento) per cui il robot e´ considerato nella stessa posizione (probabilmente il robot e´ bloccato). Il controllo viene effettuato soltanto per l'azione wander, che e´ l'unica che utilizza la mappa.

Qui di seguito il codice:

```
/**********
Procedura per liberare il robot dalle zone chiuse
**********/

private static Vec2         curr_pos;                   // dichiarate come globali, all'inizio
private static Vec2         curr_pos_temp;
private static short        counter_pos = 0;
private static short        counter_pos_2 = 0;

…

curr_pos = abstract_robot.getPosition(curr_time);    // posizione attuale, ad ogni takestep

…
```

curr_pos sara´ la variabile contenente la posizione attuale del robot, aggiornata ad ogni takestep.

curr_pos_temp invece conterra´ la posizione del robot al primo takestep, necessaria a far iniziare il conteggio.

Le altre due variabili invece tengono conto del numero di takestep in cui viene rilevata la stessa posizione del robot, in modo da agire con un'azzeramento delle celle visitate dopo un certo time out.

```
…

// Vengono usati nel caso in cui serva una tolleranza con un massimo di due cifre decimali
long x_pos = (Math.round(curr_pos.x * 100)) / 100;
long y_pos = (Math.round(curr_pos.y * 100)) / 100;
long x_pos_temp = (Math.round(curr_pos_temp.x * 100)) / 100;
long y_pos_temp = (Math.round(curr_pos_temp.y * 100)) / 100;

double toll = 0.15;          // tolleranza entro la quale il robot viene considerato nella stessa posizione

…
```

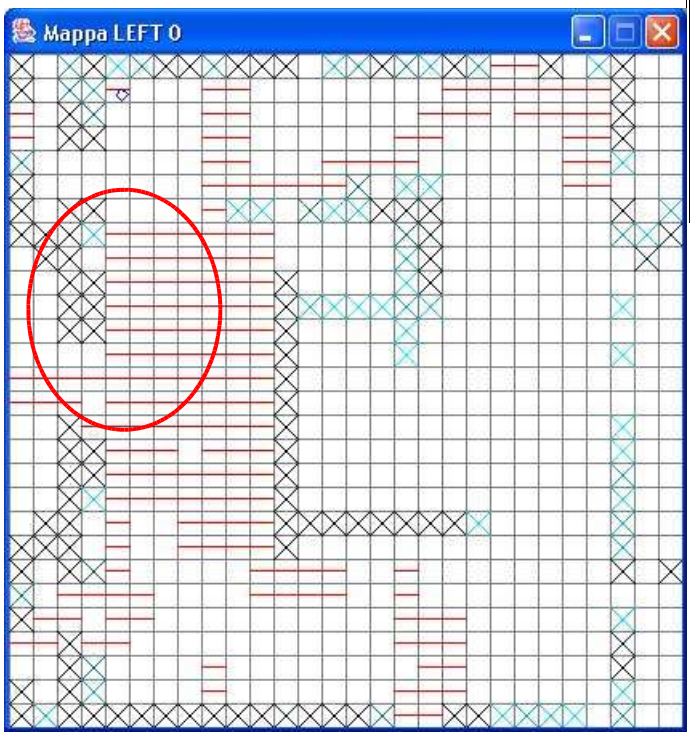
Le assegnazioni sopra elencate, permettono di effettuare delle valutazioni sulle coordinate del robot per verificare se il robot sia fermo, o comunque si muova entro una certa piccola zona, specificata dalla variabile *toll*, tolleranza.

```
…

if(counter_pos != 0){
        if((x_pos >= x_pos_temp-toll && x_pos <= x_pos_temp+toll) && (y_pos >= y_pos_temp-toll && y_pos <=
x_pos_temp+toll)){
                ++counter_pos;
                if(counter_pos == 500){
                        // Libera una matrice di celle attorno al robot
                        for(int i= -3; i<=3; i++){
                                for(int j= -5; j<=5; j++){
                                        MAP.setCellState(x_pos + i, y_pos + j, 0);
                                }
                        }
                        abstract_robot.setSteerHeading(curr_time, abstract_robot.getSteerHeading(curr_time)
+Math.PI*2/3);
                        counter_pos = 0;
                }
                counter_pos_2 = 0;
        }

…
```

Se il robot viene rilevato fermo, viene incrementato il relativo contatore fino ad un valore di 500 takestep, a questo punto viene azzerata una matrice di celle di mappa centrata sul robot e di dimensioni 7x11, ritenute come un giusto bilanciamento tra perdita di memoria e sicurezza di poter fare liberare il robot dalle situazioni di ostruzione, come illustrato nell'immagine a fianco:



Utilizzo del metodo setCellState per contrassegnare le celle come libere per consentire la liberazione del robot.

Viene quindi ruotato l'orientamento del robot di ⅔ π, per cercare di farlo tornare indietro ed intraprendere un diverso percorso, evitando di tornare nel punto in cui si era bloccato.

```
…

        else{
                curr_pos_temp = curr_pos;
                ++counter_pos_2;
                if(counter_pos_2 == 300){
                        // Libera una matrice di celle attorno al robot
                        for(int i= -1; i<=1; i++){
                                for(int j= -4; j<=4; j++){
                                        MAP.setCellState(x_pos + i, y_pos + j, 0);
                                }
                        }
                        abstract_robot.setSteerHeading(curr_time, abstract_robot.getSteerHeading(curr_time)
+Math.PI*2/3);
                        counter_pos_2 = 0;
                }
        }
}else{
        ++counter_pos;
}
```

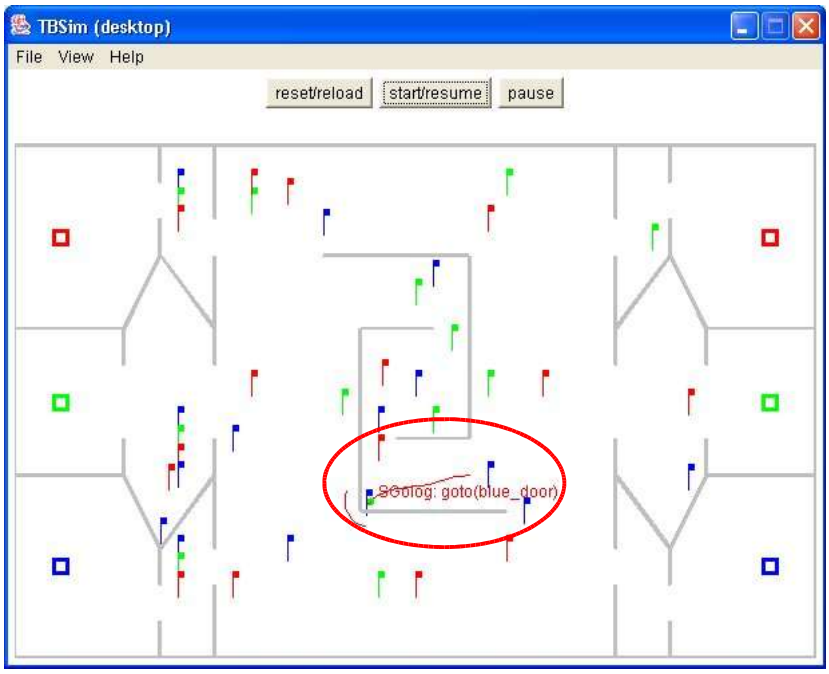
Non essendoci pero´ abbastanza precisione nel calcolo della posizione del robot, capita che esso venga rilevato in movimento anche quando e´ fermo, per cui abbiamo deciso di effettuare un altro conteggio come condizione alternativa, con time out diverso ed una minore cancellazione di celle, dato che l'errata valutazione di robot fermo e´ un inconveniente che accade con minore probabilita´ rispetto all'evento di corretta rilevazione e potrebbe essere sufficiente contrassegnare come libere un minor numero di celle attorno al robot.

Ciascun caso ha il proprio contatore, incrementato ad ogni takestep in cui si verifica la condizione appena discussa.

2.2.3 Gestione della chiocciola

Una volta gestito il primo problema presentatosi, possiamo adesso considerarne un altro, altrettanto importante.

Se ad esempio un robot raccoglie una bandierina all'interno della chiocciola centrale, questo verrebbe attratto verso il punto di attesa davanti al bin di consegna, incorrendo cosi´ nell'*intrappolamento* tra le pareti della chiocciola; e´ il tipico problema di local minima, come mostra l'immagine seguente:



Dimostrazione di minimo locale all'interno della chiocciola
che giustifica l'utilizzo della procedura snail.

Questo e´ stato risolto effettuando un controllo sulle coordinate del robot e deducendo se esso si trovi o meno all'interno della chiocciola.

Ecco il codice:

```
/**********
Procedura per capire se sono dentro la chiocciola
**********/
```

```
private boolean am_in_snail = false;        // e' un flag globale che indica se sono dentro la chiocciola

…

if(!am_in_snail){
        if((x_pos > 19.0 && x_pos <= 25.0) && (y_pos > 8.0 && y_pos < 18.0)){
                am_in_snail = true;
                // invoco la procedura snail del piano sGOLOG per gestire la chiocciola
                plan.setPlan(client.invoke("do(snail,s0,S)"), 0);
        }
}

…
```

Se il robot non si trova dentro la chiocciola (condizione vera al primo takestep, inizio della simulazione) si puo´ effettuare il test successivo, ovvero verificare se le coordinate del robot ricadono o meno all'interno dell'area di campo considerata come chiocciola man mano che esso si muove in cerca delle bandierine. Nonostante la chiocciola abbia limiti per la coordinata x identici per le due squadre, il test viene effettuato con valori differenti per la squadra che parte da sinistra e per quella che parte da destra.

Infatti potrebbe essere opportuno far percorrere la chiocciola forzatamente, anche quando il robot non vi entra per caso, per evitare ad esempio situazioni in cui viene raccolta una bandierina che produce un'attrazione lineare contro le pareti esterne della chiocciola stessa per consegnare la bandierina dal lato opposto del campo, causando cosi ´ una situazione di stallo, dovuta a minimi locali.

Se il test ha successo, ovvero il robot viene rilevato all'interno della chiocciola, viene aggiornata la variabile *am_in_snail* a true, indicando cosi´ che il robot vi si trova all'interno.

A questo punto viene invocato il piano sGOLOG in grado di gestire il percorso all'interno della chiocciola. Tale piano consiste in una procedura dedicata solamente alla gestione di quest'area del campo. La procedura chiamata *snail*, effettua la gestione della chiocciola che riesce a far uscire il robot e a farlo proseguire nel percorso verso il punto di attesa davanti ai bin, per la consegna di un'eventuale bandierina raccolta.

```
/* Procedura che provvede a gestire la chiocciola */
```

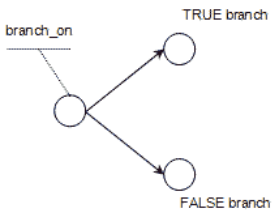
```
proc(snail,
        move_in_snail(snail_1):
        move_in_snail(snail_2):
        move_in_snail(snail_3):
        move_in_snail(snail_4):
        move_in_snail(snail_5):
        move_in_snail(snail_6)
).

/* Procedura chiamata da snail per raggiungere i singoli punti della procedura */
proc(move_in_snail(P),
        (sense(load):
                branch_on(load):
                if(load,
                        goto(P),
                        (sense(something_visible):
                                branch_on(something_visible):
                                if(something_visible,
                                        go_flag : goto(P),          /* true branch */
                                        goto(P)                     /* false branch */
                                )
                        )
                )
        )
).
```

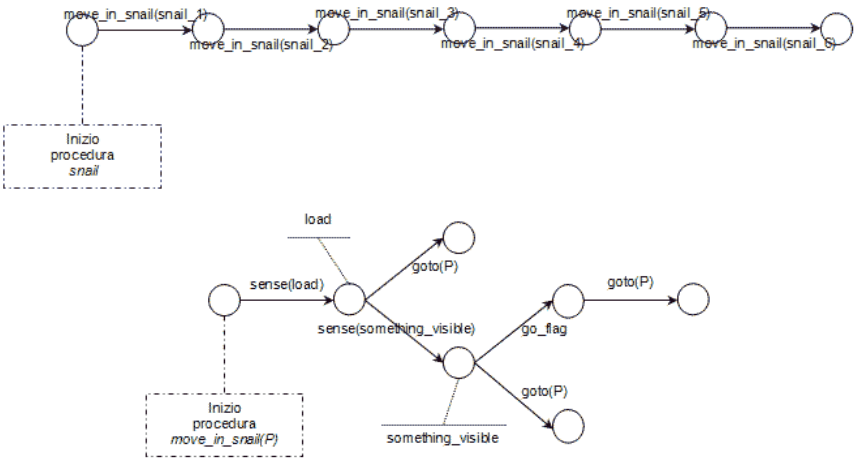
Come si puo´ vedere, la procedura consiste nel chiamare un certo numero di volte una sottoprocedura in grado di rilevare un'eventuale bandierina posseduta dal robot all'ingresso della chiocciola (con il sense sul fluente *load*) e di proseguire all'interno della chiocciola fino all'uscita opposta.

Di seguito sono riportati gli alberi della procedura snail e della sottoprocedura *move_in_snail* con la seguente convenzione:

Il TRUE branch e´ la diramazione verso l'alto,
mentre il FALSE branch e´ quella verso il basso.



Ecco gli alberi di scelta:



Se il robot non ha gia´ una bandierina, viene fatto il controllo durante tutto il percorso e, se ne viene avvistata una, verra´ raccolta e depositata.

E´ da notare che se il robot finisce per caso (durante il wander) dentro la chiocciola, e´ in grado di rilevare la presenza di eventuali bandierine e di consegnarle a destinazione.

```
…

if(am_in_snail && (x_pos >= 22.0 && x_pos <= 24.0) && (y_pos >= 20.0 && y_pos <= 22.0)){
        am_in_snail = false;
        // invoco il normale piano sGOLOG execute
        plan.setPlan(client.invoke("do(execute,s0,S)"), 0);
}
```

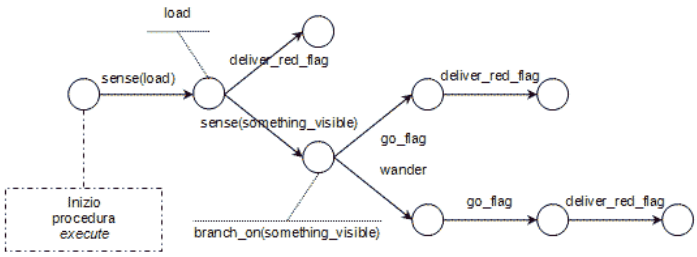
Una volta che il robot ha terminato il suo percorso dentro la chiocciola, viene reimpostata a false la variabile am_in_snail e richiamato il piano completo sGOLOG, la procedura *execute* di seguito riportata:

```
/* Procedura che esegue la pianificazione */
proc(execute,
        (sense(load):
                branch_on(load):
                if(load,
                        deliver_red_flag,
                        (sense(something_visible):
                                branch_on(something_visible):
```

```
                        if(something_visible,
                                go_flag,                /* true branch */
                                wander : go_flag        /* false branch */
                        )
                ):
                deliver_red_flag
        )       /* end if(load) */
    )
).
```

La procedura execute effettua il sense sul fluente load per verificare se il robot ha gia´ nel gripper una bandierina da depositare nel rispettivo bin. Se il fluente e´ vero, procedo con il deposito della bandierina, dopo averne effettuato il riconoscimento del colore, altrimenti viene verificata la presenza di qualche bandierina nel cono di visione del robot durante il suo movimento in condizioni di ricerca. Se la ricerca termina a buon fine (il fluente *something_visible* risulta vero), il robot si dirige verso di essa eseguendo l'azione *go_flag*, altrimenti permane nella condizione di wander finche´ non ne viene avvistata una. Viene quindi richiamata la procedura *deliver_red_flag* che e´ delegata a controllare il colore della bandierina raccolta e agire di conseguenza.

Di seguito e´ riportato l'albero della procedura execute:

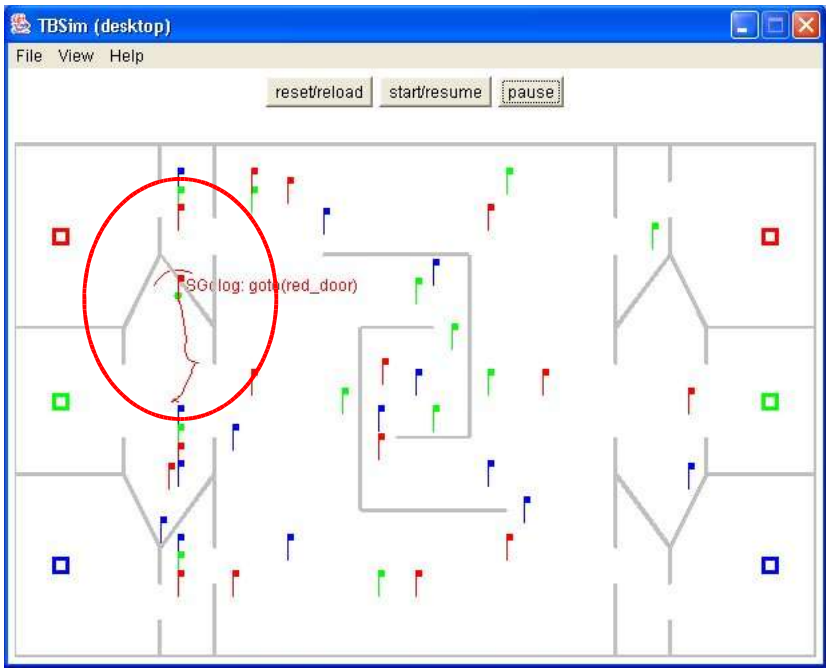


Come illustrato successivamente, la procedura *deliver_red_flag* effettuera´ un controllo ricorsivo sul colore della bandierina fino a quando viene invocata la corretta (in funzione del colore) sequenza di azioni per depositare la bandierina.

Tornando alla gestione della chiocciola, una volta eseguite tutte le azioni specificate, il robot riprende quindi il normale funzionamento; ovvero se ha una bandierina in mano la va a depositare, altrimenti ritorna allo stato di ricerca delle bandierine, il wander.

2.2.4 Gestione delle anticamere

Si e´ detto in precedenza che e´ anche possibile trovare bandierine nelle anticamere dei bin, quindi questa situazione e´ causa di altri problemi di minimo locale, come illustrato in figura:



Dimostrazione di minimo locale all'interno delle anticamere.

Si deve quindi effettuare un ulteriore controllo sulle coordinate del robot, per capire se esso si trovi o meno all'interno delle anticamere.

```
/**********
Procedura per capire se sono dentro le anticamere e in quale mi trovo
**********/

private NodeBooleanDinamic    PF_IN_RED_LOBBY;      // dichiarate come globali, sono le perceptual
private NodeBooleanDinamic    PF_IN_GREEN_LOBBY;    // features che indicano se il robot e´ dentro
private NodeBooleanDinamic    PF_IN_BLUE_LOBBY;     // un'anticamera

private boolean               am_in_snail = false;
private boolean               gripper_open = false; // indica se e´ stato aperto il gripper
```

```
…

if(!am_in_snail){          // il robot puo´ trovarsi dentro le anticamere solamente se non e´ gia´ dentro la chiocciola
        if(x_pos < 11 || x_pos > 33){
                if(y_pos >= 22){
                        PF_IN_RED_LOBBY.setValue(true);
                        PF_IN_GREEN_LOBBY.setValue(false);
                        PF_IN_BLUE_LOBBY.setValue(false);
                }
                if(y_pos >= 10 && y_pos <= 18){
                        PF_IN_RED_LOBBY.setValue(false);
                        PF_IN_GREEN_LOBBY.setValue(true);
                        PF_IN_BLUE_LOBBY.setValue(false);
                }
                if(y_pos <= 6){
                        PF_IN_RED_LOBBY.setValue(false);
                        PF_IN_GREEN_LOBBY.setValue(false);
                        PF_IN_BLUE_LOBBY.setValue(true);
                }

…
```

Effettuo un semplice controllo sulle coordinate del robot per capire se mi trovo dentro ad un'anticamera, ed in caso affermativo, viene determinata quale sia l'anticamera di interesse, se quella antistante il bin rosso, a quello verde oppure a quello blu.

Vengono utilizzate delle *perceptual features* di tipo *NodeBooleanDinamic* poiche´ consente, rispetto al tipo *NodeBoolean*, l'assegnazione dinamica dei valori di verita´ alle features stesse.

```
…

                if(((y_pos > 18 && y_pos < 22) || (y_pos > 6 && y_pos < 10)) && (x_pos > 6 && x_pos < 11)){
                        abstract_robot.setGripperFingers(curr_time, 1);
                        gripper_open = true;
                }

…
```

Questa porzione di codice gestisce le aree triangolari o trapezoidali dentro ogni anticamera, ricavate per esclusione dopo aver considerato le aree rettangolari inscritte in ciascuna anticamera.

Se il robot si trova all'interno di una di esse (solo per la fascia sinistra della squadra sinistra e solo per la fascia destra per la squadra destra), il robot apre il gripper rendendosi incapace di raccogliere bandierine, onde evitare situazioni in cui non sarebbe in grado di uscire dall'anticamera per consegnarla poiche´ quest'area irregolare non viene percepita come appartenente ad alcuna anticamera.

```
…

        else{
                        if(gripper_open){
                                plan.setPlan(client.invoke("do(execute,s0,S)"), 0);
                                gripper_open = false;
                        }
                }

…
```

Appena il robot esce dalla zone indicate, viene richiamata la procedura execute per tornare al normale funzionamento.

```
…

        }else{
                PF_IN_RED_LOBBY.setValue(false);
                PF_IN_GREEN_LOBBY.setValue(false);
                PF_IN_BLUE_LOBBY.setValue(false);
        }
}

/*********/
```

Se invece il robot non si trova dentro alcuna anticamera, tutte le perceptual features vengono impostate a false e il comportamento del robot permane nello stato di funzionamento attuale.

Adesso e´ importante vedere come opera il database sGOLOG a riguardo.

Una volta accertato che il robot si trova dentro le anticamere ed identificata la singola anticamera, relativa al bin rosso, verde o blu, appena il robot raccoglie una bandierina, viene effettuato il sense sul fluente che ne identifica il colore (*red_flag*, *green_flag* o *blue_flag*), per capire in quale bin dovra´ essere depositata.

La procedura seleziona quindi il percorso diretto, se la bandierina deve essere depositata nel bin dell'anticamera in cui il robot si trova attualmente, oppure il percorso che fara´ uscire il robot dall'anticamera e lo condurra´ a destinazione.

```
proc(deliver_red_flag,
        (sense(red_flag):
                branch_on(red_flag):
                if(red_flag,
                        (sense(in_red_lobby):
                                branch_on(in_red_lobby):
                                if(in_red_lobby,
                                        goto(red_attractor) : go_bin(red_door, red_attractor, red_bin,
to_field_from_red),
                                        (sense(in_green_lobby):
                                                branch_on(in_green_lobby):
                                                if(in_green_lobby,
                                                        goto(green_door) : goto(to_field_from_green) : goto
(red_door) : goto(red_attractor) : go_bin(red_door, red_attractor, red_bin, to_field_from_red),
                                                        (sense(in_blue_lobby):
                                                                branch_on(in_blue_lobby):
                                                                if(in_blue_lobby,
                                                                        goto(blue_door) : goto
(to_field_from_blue) : goto(red_door) : goto(red_attractor) : go_bin(red_door, red_attractor, red_bin,
to_field_from_red),
                                                                        goto(red_door) : goto(red_attractor) :
go_bin(red_door, red_attractor, red_bin, to_field_from_red)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        ),
                        deliver_green_flag
                )
        )
```

).

La procedura sopra riportata e´ relativa alla consegna della bandierina rossa, similmente si possono ottenere quelle relative alle bandierine verdi e blu.

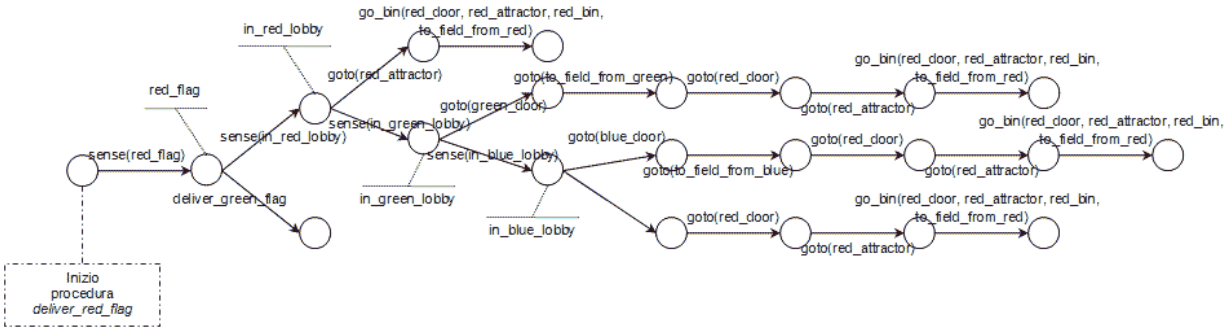
Dapprima, viene effettuato il sense sul fluente red_flag che e´ vero se la bandierina appena acquisita e´ di colore rosso. In tal caso, vengono effettuati in sequenza tutti i sense per verificare in quale anticamera si trova il robot e agire di conseguenza. L'ultimo caso e´, per esclusione, quello in cui il robot non si trova in alcuna anticamera procedendo con la normale consegna della bandierina.

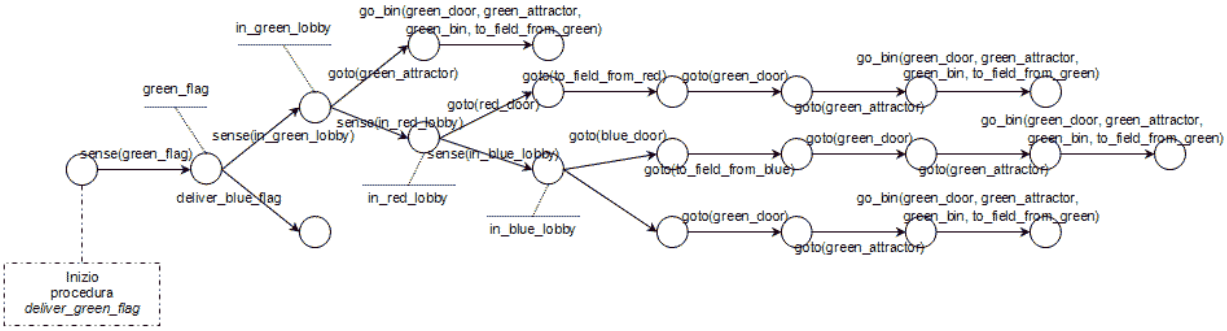
In base al valore dei fluenti che identificano le anticamere, il robot eseguira´ le azioni corrette che lo porteranno eventualmente ad uscire dall'anticamera e a depositare la bandierina.

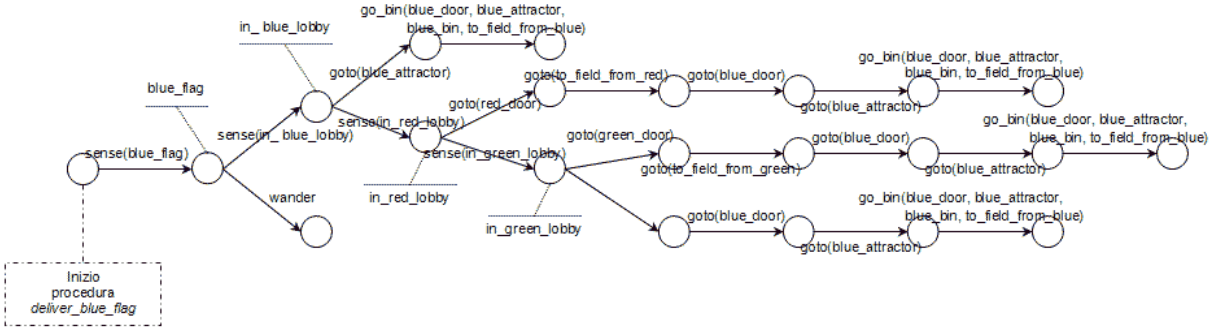
E´ da notare che se il fluente red_flag ha valore falso, viene chiamata la procedura green_flag che effettuera´ gli stessi controlli effettuati dalla procedura per la bandierina rossa. Se la bandierina non e´ nemmeno verde, verra´ chiamata la procedura relativa alla bandierina blu.

Come caso alternativo all'ultima procedura, e´ stato utilizzato il comportamento wander che viene richiamato quando ad esempio una bandierina non viene afferrata correttamente e viene di conseguenza persa.

L'albero di scelta sGOLOG generato dalle procedure deliver_red_flag, deliver_green_flag e deliver_blue_flag e´ il seguente:







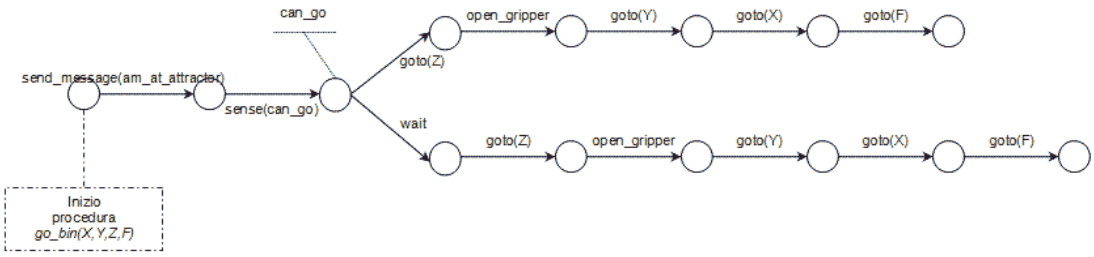
Le procedure deliver richiamano la sottoprocedura go_bin, riportata di seguito:

```
/* Procedura per evitare i minimi locali quando trovo o poso le bandierine nei pressi delle anticamere */
proc(go_bin(X,Y,Z,F),
        send_message(am_at_attractor) :
        (sense(can_go):
                branch_on(can_go):
                if(can_go,
                        goto(Z),
                        wait : goto(Z)
                )
        ):
        open_gripper : goto(Y) : goto(X) : goto(F)
).
```

La prima azione che viene effettuata e´ quella di mandare il messaggio e settare in questo modo il fluente *can_go*, usato di seguito, a true, informando il robot compagno che e´ pronto per la consegna della bandierina. Se il sense sul fluente can_go e´ riconosciuto vero, e quindi l'altro robot ha gia´ mandato il messaggio ed e´ in wait, effettuo l'azione *goto(Z)* che corrisponde al dirigersi verso il bin, altrimenti se can_go e´ falso il robot

rimane in attesa finche´ il compagno, chiamando la stessa procedura go_bin, provvede ad inviare il proprio messaggio, dando l'autorizzazione a completare la procedura. Questo stato di attesa in cui si trova il robot e´ il wait.

L'albero generato dalla procedura go_bin e´ invece:



3 Interfacciamento TeamBots - sGOLOG

3.1 Matching tra comportamenti TeamBots e azioni sGOLOG

Per poter utilizzare entrambi gli strumenti a nostra disposizione, e´ necessario stabilire le corrispondenze tra le azioni utilizzate dal TeamBots e quelle utilizzate invece dal database sGOLOG. Questo puo´ essere realizzato dalla seguente porzione di codice presente nel file java:

```
// Matching tra azioni primitive sGOLOG e azioni corrispondenti agli stati del robot
plan.addAction("goto",
                    AS_GOTO,
                    AS_GOTO,
                    PF_CLOSE_TO,
                    0,
                    new int[]{1,3,5,7,9,11,13,15},
                    NodeAction.LASER_ON);

plan.addAction("wander",
                    AS_GO_CLOSEST_ZONE,
                    AS_GO_CLOSEST_ZONE,
                    PF_SOMETHING_VISIBLE,
                    0,          // chiudo comunque il gripper nello stato di wander
                    new int[]{0,2,4,6,8,10,12,14},
                    NodeAction.LASER_ON);

plan.addAction("go_flag",
                    AS_GO_FLAG,
                    AS_GO_FLAG,
                    PF_TRIGGER_FLAG,
                    -1,
                    new int[]{},
                    NodeAction.LASER_OFF);

plan.addAction("open_gripper",
                    NodeAction.NO_STEERING,
```

```
                NodeAction.NO_TURRET,
                NodeAction.IMMEDIATE_TRIGGER,
                1,
                new int[]{},
                NodeAction.LASER_OFF);

plan.addAction("send_message",
                NodeAction.NO_STEERING,
                NodeAction.NO_TURRET,
                NodeAction.IMMEDIATE_TRIGGER,
                NodeAction.NO_GRIPPER,
                new int[]{},
                NodeAction.LASER_OFF);

plan.addAction("wait",
                AS_GOTO,
                AS_GOTO,
                PF_OTHER_ROBOT,
                NodeAction.NO_GRIPPER,
                new int[]{},
                NodeAction.LASER_OFF);
```

L'elenco sopra riportato rappresenta il matching tra le azioni sGOLOG e le corrispondenti azioni utilizzate in TeamBots.

3.2 Matching tra triggers TeamBots e fluenti sGOLOG

Quello che segue, e´ l'elenco delle corrispondenze tra i fluenti sGOLOG e i triggers TeamBots:

```
…

// Matching tra fluenti di sGOLOG e trigger di TeamBots
plan.addAction("something_visible", PF_SOMETHING_VISIBLE);

plan.addAction("red_flag", PF_RED_FLAG_IN_GRIPPER);

plan.addAction("green_flag", PF_GREEN_FLAG_IN_GRIPPER);

plan.addAction("blue_flag", PF_BLUE_FLAG_IN_GRIPPER);
```

```
plan.addAction("can_go", PF_OTHER_ROBOT);

plan.addAction("load", PF_SOMETHING_IN_GRIPPER);

plan.addAction("in_red_lobby", PF_IN_RED_LOBBY);

plan.addAction("in_green_lobby", PF_IN_GREEN_LOBBY);

plan.addAction("in_blue_lobby", PF_IN_BLUE_LOBBY);
```

Altre corrispondenze devono essere stabilite per le coordinate del punto attrattore dinamico, utilizzato dall'azione *goto*.

Il punto attrattore potra´ essere, di volta in volta, un bin, un punto intermedio oppure un punto utilizzato per la gestione della chiocciola.

Segue l'elenco di tali corrispondenze:

```
if(action.startsWith("goto(")){
        if(plan.getParameter()[0].equals("red_bin")){
                PS_ATTRACTOR.setValue(curr_time, 2.5, 23.0);
        }else
        if(plan.getParameter()[0].equals("green_bin")){
                PS_ATTRACTOR.setValue(curr_time, 2.5, 14.0);
        }else
        if(plan.getParameter()[0].equals("blue_bin")){
                PS_ATTRACTOR.setValue(curr_time, 2.5, 5.0);
        }

…
```

red_bin, *green_bin* e *blue_bin* sono le coordinate dei contenitori di bandierine.

Tali coordinate variano a seconda se la squadra parte da sinistra o destra. Quelle sopra riportate sono relative alla squadra che parte da destra e che deve depositare le bandierine a sinistra.

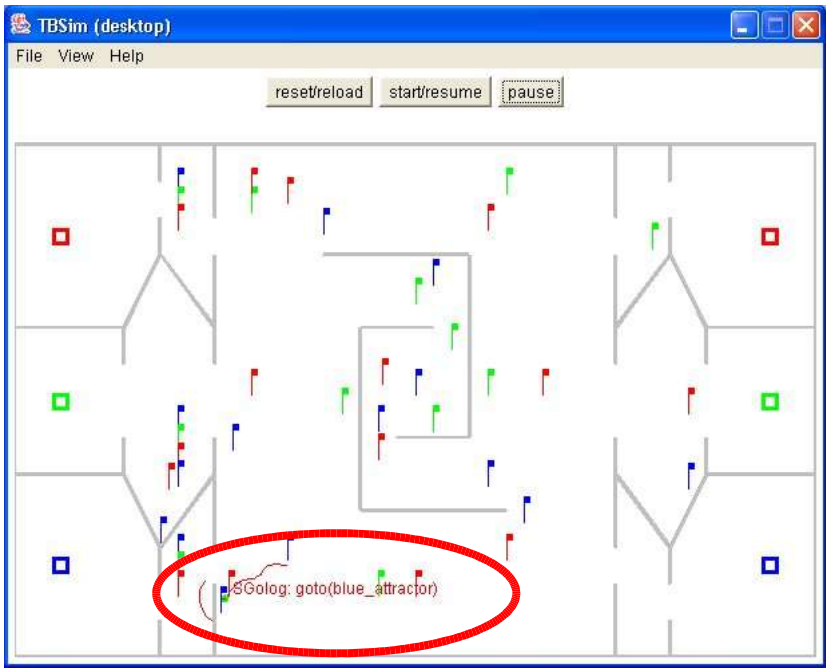
```
…
```

```
        else
        if(plan.getParameter()[0].equals("red_attractor")){
                PS_ATTRACTOR.setValue(curr_time, 8.0, 25.2);
        }else
        if(plan.getParameter()[0].equals("green_attractor")){
                PS_ATTRACTOR.setValue(curr_time, 6.0, 14.6);
        }else
        if(plan.getParameter()[0].equals("blue_attractor")){
                PS_ATTRACTOR.setValue(curr_time, 8.0, 3.2);
        }else
        if(plan.getParameter()[0].equals("red_door")){
                PS_ATTRACTOR.setValue(curr_time, 11.5, 22.5);
        }else
        if(plan.getParameter()[0].equals("green_door")){
                PS_ATTRACTOR.setValue(curr_time, 11.5, 14.0);
        }else
        if(plan.getParameter()[0].equals("blue_door")){
                PS_ATTRACTOR.setValue(curr_time, 11.5, 5.5);
        }else
        if(plan.getParameter()[0].equals("to_field_from_red")){
                PS_ATTRACTOR.setValue(curr_time, 14.0, 22.5);
        }else
        if(plan.getParameter()[0].equals("to_field_from_green")){
                PS_ATTRACTOR.setValue(curr_time, 14.0, 14.0);
        }else
        if(plan.getParameter()[0].equals("to_field_from_blue")){
                PS_ATTRACTOR.setValue(curr_time, 14.0, 5.5);
        }

…
```

I punti con suffisso *attractor* sono quelli in cui i robot devono aspettarsi a vicenda per la consegna delle bandierine e si trovano al limite tra la stanza dei bin e la relativa anticamera.

Quelli con suffisso *door* sono stati utilizzati per agevolare soprattutto l'ingresso, nonche´ l'uscita nella/dalla anticamera dei bin. Infatti, come si puo´ notare dalla figura che segue, vi sarebbero notevoli difficolta´ ad ogni consegna delle bandierine.



Dimostrazione di minimo locale se non si utilizzassero i punti attrattori con suffisso door.

I punti invece con prefisso *to_field_from* sono stati utilizzati per far allontanare il robot dalle anticamere una volta depositate le bandierine, onde evitare un comportamento di wander concentrato davanti o addirittura
dentro le anticamere stesse.

```
…

        else
        if(plan.getParameter()[0].equals("snail_1")){
                PS_ATTRACTOR.setValue(curr_time, 20.0, 10.5);
        }else
        if(plan.getParameter()[0].equals("snail_2")){
                PS_ATTRACTOR.setValue(curr_time, 20.2, 15.6);
        }else
        if(plan.getParameter()[0].equals("snail_3")){
                PS_ATTRACTOR.setValue(curr_time, 22.0, 15.0);
        }else
        if(plan.getParameter()[0].equals("snail_4")){
                PS_ATTRACTOR.setValue(curr_time, 23.0, 13.0);
        }else
        if(plan.getParameter()[0].equals("snail_5")){
                PS_ATTRACTOR.setValue(curr_time, 24.0, 17.5);
```

```
        }else
        if(plan.getParameter()[0].equals("snail_6")){
                PS_ATTRACTOR.setValue(curr_time, 23.0, 21.0);
        }
}
```

I punti con prefisso *snail* invece vengono utilizzati per percorrere la chiocciola ed evitare un eventuale situazione di stallo all'interno della stessa.

4
Codice utilizzato per la competizione

4.1 Database sGOLOG

```
/* FILE Competizione.pl */
/* Azioni primitive */
primitive_action(goto(_)).
primitive_action(wander).
primitive_action(go_flag).
primitive_action(open_gripper).
primitive_action(send_message(_)).
primitive_action(wait).
primitive_action(sense(_)).

/* Precondizioni */
poss(goto(_),_).
poss(wander,_).
poss(go_flag,_).
poss(open_gripper,_).
poss(send_message(_),_).
poss(wait,_).
poss(sense(_),_).

/* Assiomi di stato successore */
holds(something_visible, do(A,S))   :-   holds(something_visible,S);
                                         A = assm(something_visible,1).

holds(red_flag, do(A,S))            :-   holds(something_visible,S), A = go_flag;
                                         A = assm(red_flag,1).

holds(green_flag, do(A,S))          :-   holds(something_visible,S), A = go_flag;
                                         A = assm(green_flag,1).

holds(blue_flag, do(A,S))           :-   holds(something_visible,S), A = go_flag;
                                         A = assm(blue_flag,1).

holds(load, do(A,S))                :-   holds(load,S);
                                         A = assm(load,1).

holds(in_red_lobby, do(A,S))                :-   holds(in_red_lobby,S);
                                         A = assm(in_red_lobby,1).

holds(in_green_lobby, do(A,S))      :-   holds(in_green_lobby,S);
                                         A = assm(in_green_lobby,1).

holds(in_blue_lobby, do(A,S))               :-   holds(in_blue_lobby,S);
                                         A = assm(in_blue_lobby,1).

holds(can_go, do(A,S))              :-   (holds(red_flag,S),   A = goto(red_attractor)  );
                                         (holds(blue_flag,S),  A = goto(blue_attractor) );
                                         (holds(green_flag,S), A = goto(green_attractor));
                                         A = assm(can_go,1).

/* Non sono necessari fatti sulla situazione iniziale s0 */


/* Procedura che esegue la pianificazione */
proc(execute,
        (sense(load):
```

```
                branch_on(load):
                if(load,
                        deliver_red_flag,
                        (sense(something_visible):
                                branch_on(something_visible):
                                if(something_visible,
                                        go_flag,            /* true branch */
                                        wander : go_flag    /* false branch */
                                )
                        ):
                        deliver_red_flag
                )       /* end if(load) */
        )
).

proc(deliver_red_flag,
        (sense(red_flag):
                branch_on(red_flag):
                if(red_flag,
                        (sense(in_red_lobby):
                                branch_on(in_red_lobby):
                                if(in_red_lobby,
                                        goto(red_attractor) : go_bin(red_door, red_attractor, red_bin, to_field_from_red),
                                        (sense(in_green_lobby):
                                                branch_on(in_green_lobby):
                                                if(in_green_lobby,
                                                        goto(green_door) : goto(to_field_from_green) : goto(red_door) :
goto(red_attractor) : go_bin(red_door, red_attractor, red_bin, to_field_from_red),
                                                        (sense(in_blue_lobby):
                                                                branch_on(in_blue_lobby):
                                                                if(in_blue_lobby,
                                                                        goto(blue_door) : goto(to_field_from_blue) :
goto(red_door) : goto(red_attractor) : go_bin(red_door, red_attractor, red_bin, to_field_from_red),
                                                                        goto(red_door) : goto(red_attractor) : go_bin
(red_door, red_attractor, red_bin, to_field_from_red)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        ),
                        deliver_green_flag
                )
        )
).

proc(deliver_green_flag,
        (sense(green_flag):
                branch_on(green_flag):
                if(green_flag,
                        (sense(in_green_lobby):
                                branch_on(in_green_lobby):
                                if(in_green_lobby,
                                        goto(green_attractor) : go_bin(green_door, green_attractor, green_bin,
to_field_from_green),
                                        (sense(in_red_lobby):
                                                branch_on(in_red_lobby):
                                                if(in_red_lobby,
                                                        goto(red_door) : goto(to_field_from_red) : goto(green_door) :
goto(green_attractor) : go_bin(green_door, green_attractor, green_bin, to_field_from_green),
                                                        (sense(in_blue_lobby):
                                                                branch_on(in_blue_lobby):
                                                                if(in_blue_lobby,
                                                                        goto(blue_door) : goto(to_field_from_blue) :
goto(green_door) : goto(green_attractor) : go_bin(green_door, green_attractor, green_bin, to_field_from_green),
                                                                        goto(green_door) : goto(green_attractor) :
go_bin(green_door, green_attractor, green_bin, to_field_from_green)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
```

```
                                        ),
                                        deliver_blue_flag
                                )
                        )
).

proc(deliver_blue_flag,
        (sense(blue_flag):
                branch_on(blue_flag):
                if(blue_flag,
                        (sense(in_blue_lobby):
                                branch_on(in_blue_lobby):
                                if(in_blue_lobby,
                                        goto(blue_attractor) : go_bin(blue_door, blue_attractor, blue_bin,
to_field_from_blue),
                                        (sense(in_red_lobby):
                                                branch_on(in_red_lobby):
                                                if(in_red_lobby,
                                                        goto(red_door) : goto(to_field_from_red) : goto(blue_door) : goto
(blue_attractor) : go_bin(blue_door, blue_attractor, blue_bin, to_field_from_blue),
                                                        (sense(in_green_lobby):
                                                                branch_on(in_green_lobby):
                                                                if(in_green_lobby,
                                                                        goto(green_door) : goto
(to_field_from_green) : goto(blue_door) : goto(blue_attractor) : go_bin(blue_door, blue_attractor, blue_bin, to_field_from_blue),
                                                                        goto(blue_door) : goto(blue_attractor) :
go_bin(blue_door, blue_attractor, blue_bin, to_field_from_blue)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        ),
                        wander
                )
        )
).

/* Procedura per evitare i minimi locali quando trovo o poso le bandierine nei pressi delle anticamere */
proc(go_bin(X,Y,Z,F),
        send_message(am_at_attractor) :
        (sense(can_go):
                branch_on(can_go):
                if(can_go,
                        goto(Z),
                        wait : goto(Z)
                )
        ):
        open_gripper : goto(Y) : goto(X) : goto(F)
).

/* Procedura che provvede a gestire la chiocciola */
proc(snail,
        move_in_snail(snail_1):
        move_in_snail(snail_2):
        move_in_snail(snail_3):
        move_in_snail(snail_4):
        move_in_snail(snail_5):
        move_in_snail(snail_6)
).

/* Procedura chiamata da snail() per raggiungere i singoli punti della procedura snail */
proc(move_in_snail(P),
        (sense(load):
                branch_on(load):
                if(load,
                        goto(P),
                        (sense(something_visible):
                                branch_on(something_visible):
                                if(something_visible,
                                        go_flag : goto(P),    /* true branch */
                                        goto(P)               /* false branch */
```

```
                                        )
                                )
                        )
                )
).
```

4.2 Codice dei robot

Viene riportato il codice del primo robot della squadra sinistra (TUX_1_LEFT.java) in quanto i sorgenti dei robot sono uguali a meno di varianti per le coordinate dei bin o di alcuni punti attrattori.

```
// TUX_1_LEFT.java

// Tesina di Robotica Cognitiva
// Gara 2003-2004

/*      Autori:
                Riccardo Marfisi
                Salvatore La Bua
*/


import EDU.gatech.cc.is.util.Vec2;
import EDU.gatech.cc.is.clay.*;
import EDU.gatech.cc.is.communication.*;

import CSAI.unipa.clay.*;
import CSAI.unipa.knowledgment.*;
import CSAI.unipa.abstractrobot.*;
import CSAI.unipa.SGolog.*;

import java.awt.*;
import java.io.*;
import java.util.Enumeration;


public class TUX_1_LEFT extends ControlSystemMFN150Explore{

        public final static boolean             DEBUG = true;
        private NodeVec2                        turret_configuration;
        private NodeVec2                        steering_configuration;
        private NodeDouble                              gripper_fingers_configuration;
        private NodeBoolean                     sonar_configuration;
        private NodeBoolean                     laser_configuration;
        private NodeGain                        mtggain = new NodeGain(1.0);            // sono i pesi dei comportamenti elementari
        private NodeGain                        swirlgain = new NodeGain(0.8);                  //
        private NodeGain                        avoidgain = new NodeGain(0.8);                  //
        private NodeGain                        noisegain = new NodeGain(0.2);                  //
        private NodeInt                         state_monitor;
        private i_FSA_ba                        STATE_MACHINE;
        private NodeVec2                        PS_GLOBAL_POS;
        private NodeVec2Array                   PS_OBS;                         // contengono gli ostacoli rilevati
        private NodeVec2Array                   PS_ALL_OBS;                     //
        private NodeVec2Array                   PS_SONAR;       // contengono le rilevazioni dei sonar
        private NodeVec2Array                   PS_LASER;       // contengono le rilevazioni del laser
        private NodeBooleanDinamic                      PF_OTHER_ROBOT;         // e' utilizzato per la comunicazione tra i
robot
        private NodeBooleanDinamic                      PF_IN_RED_LOBBY; // sono le perceptual features che gestiscono le
anticamere
        private NodeBooleanDinamic                      PF_IN_GREEN_LOBBY;              //
```

```
	private NodeBooleanDinamic		PF_IN_BLUE_LOBBY;		//
	private v_DinamicPoint_		PS_ATTRACTOR;
	private Enumeration		bufferMessage;
	private sGologClient		client;
	private SGologPlan			plan;
	private int				id;			// e' l'id di questo robot
	private int				other_id;		// e' l'id dell'altro robot ricavato per esclusione
	private NodeMap			MAP;		// e' l'oggetto mappa
	private static Vec2			curr_pos;			// in queste variabili viene salvata la posizione del
robot
	private static Vec2			curr_pos_temp;	// ad ogni takestep
	private static short			counter_pos = 0;	// sono le variabili che contengono il conteggio di quanti
takestep
	private static short			counter_pos_2 = 0;			// il robot permane nella stessa posizione
	private boolean			am_in_snail = false;		// e' un flag che indica se il robot e' dentro la
chiocciola
	private boolean			gripper_open = false;		// indica se e' stato aperto il gripper

	public void configure(){

		// Configurazione Iniziale
		abstract_robot.setBaseSpeed(abstract_robot.MAX_TRANSLATION);
		abstract_robot.setObstacleMaxRange(3.0);
		abstract_robot.setGripperHeight(-1, 0);		// abbassamento dell'artiglio
		abstract_robot.setGripperFingers(-1, -1);		// modalità pronto (secondo -1), 0 chiuso, 1 aperto e -1 pronto

		// Variabile temporanea della posizione del robot utilizzata quando si blocca nel campo in zone chiuse
		curr_pos_temp = abstract_robot.getPosition(abstract_robot.getTime());

		// Setup della cintura sonar
		abstract_robot.turnOffAllSonar();

		/* Viene creata la mappa passando i limiti (left, right, up, down), dimensione delle celle (quadrate),
		   raggio di appartenenza alla cella, nome della mappa, se la griglia si deve vedere ed il robot che la sta creando
		*/
		MAP = new NodeMap(8.0, 36.0, 28.0, 0.0, 1.0, 0.6, "Mappa LEFT 0", false, abstract_robot);

		// Il metodo getReceiveChannel() restituisce un oggetto EDU.gatech.cc.is.util.CircularBufferEnumeration
		bufferMessage = abstract_robot.getReceiveChannel();


		/*****************************************************************************************
					PERCEPTUAL SCHEMAS
		*****************************************************************************************/

		// Posizione Globale del robot
		PS_GLOBAL_POS = new v_GlobalPosition_r(abstract_robot);

		// Posizione degli ostacoli
		PS_OBS = new va_Obstacles_r(abstract_robot);

		PS_SONAR = new va_SonarSensed_r(abstract_robot);

		PS_LASER = new va_LaserSensed_r(abstract_robot);

		/* Contiene tutti gli ostacoli rilevati dai sonar, dal laser e dal vecchio sistema di rilevamento.
		   (E' usata per calcolare la traiettoria che fa evitare gli ostacoli al robot)
		*/
		NodeVec2Array
		PS_TEMP = new va_Merge_vava(PS_LASER, PS_SONAR);
		PS_ALL_OBS = new va_Merge_vava(PS_TEMP, PS_OBS);

		// E' il punto dinamico che sfrutto poi nel database sGOLOG
		PS_ATTRACTOR = new v_DinamicPoint_(PS_GLOBAL_POS, 0.0, 0.0);

		/* public v_Closest_mr(NodeMap im1, EDU.gatech.cc.is.abstractrobot.SimpleInterface ar)
		   Parameters: im1 - NodeVec2, the embedded (inciso) node that generates a list of items (componenti) to scan
		   Finds the closest in a list of Vec2s. Assumes the vectors point egocentrically to the objects, so that the
		   closest one has the shortest r value.
		*/
		// Fornisce la cella piu' vicina non visitata
		NodeVec2
```

```
PS_CLOSEST_ZONE = new v_Closest_mr(MAP, abstract_robot);

// Bandierine di classe visuale 0, associate al colore blu
NodeVec2Array
PS_BLUE_FLAG_EGO = new va_VisualObjects_r(0, abstract_robot);

// Bandierine di classe visuale 1, associate al colore verde
NodeVec2Array
PS_GREEN_FLAG_EGO = new va_VisualObjects_r(1, abstract_robot);

// Bandierine di classe visuale 2, associate al colore rosso
NodeVec2Array
PS_RED_FLAG_EGO = new va_VisualObjects_r(2, abstract_robot);

// Riutilizzo la variabile PS_TEMP
PS_TEMP = new va_Merge_vava(PS_GREEN_FLAG_EGO, PS_RED_FLAG_EGO);
NodeVec2Array
PS_ALL_TARGET = new va_Merge_vava(PS_TEMP, PS_BLUE_FLAG_EGO);

// Contiene le bandierine dalla piu' vicina
NodeVec2
PS_CLOSEST_FLAG = new v_Closest_va(PS_ALL_TARGET);

// Tipo di oggetto nel gripper
NodeInt
PS_IN_GRIPPER = new i_InGripper_r(abstract_robot);


/*******************************************************************************************
                        PERCEPTUAL FEATURES
*******************************************************************************************/

NodeBoolean
PF_CLOSE_TO = new b_CloseDinamic_vv(0.6, PS_GLOBAL_POS, PS_ATTRACTOR);

NodeBoolean
PF_SOMETHING_VISIBLE = new b_NonZero_v(PS_CLOSEST_FLAG);

PF_OTHER_ROBOT = new NodeBooleanDinamic(false);

// Sono le perceptual features che gestiscono le anticamere
PF_IN_RED_LOBBY = new NodeBooleanDinamic(false);

PF_IN_GREEN_LOBBY = new NodeBooleanDinamic(false);

PF_IN_BLUE_LOBBY = new NodeBooleanDinamic(false);

NodeBoolean
PF_BLUE_FLAG_IN_GRIPPER = new b_Equal_i(0, PS_IN_GRIPPER);

NodeBoolean
PF_GREEN_FLAG_IN_GRIPPER = new b_Equal_i(1, PS_IN_GRIPPER);

NodeBoolean
PF_RED_FLAG_IN_GRIPPER = new b_Equal_i(2, PS_IN_GRIPPER);

NodeBoolean
PF_SOMETHING_IN_GRIPPER = new b_Persist_s(4.0, new b_NonNegative_s(PS_IN_GRIPPER));

NodeBoolean
PF_NOT_SOMETHING_VISIBLE = new b_Not_s(PF_SOMETHING_VISIBLE);

NodeBoolean
PF_TRIGGER_FLAG = new b_Or_bb(PF_NOT_SOMETHING_VISIBLE, PF_SOMETHING_IN_GRIPPER);


/*******************************************************************************************
                        MOTOR SCHEMAS
*******************************************************************************************/

NodeVec2
MS_AVOID_OBSTACLES = new v_AvoidSonar_va(2.0, abstract_robot.RADIUS + 0.2, PS_OBS,
```

```
                                                                    PS_GLOBAL_POS);

        // Attrazione lineare verso l'attrattore dinamico temporaneo definito precedentemente
        NodeVec2
        MS_MOVE_TO = new v_LinearAttraction_v(0.4, 0.0, PS_ATTRACTOR);

        NodeVec2
        MS_MOVE_TO_FLAG = new v_LinearAttraction_v(0.4, 0.0, PS_CLOSEST_FLAG);

        NodeVec2
        MS_MOVE_TO_CLOSEST_ZONE = new v_LinearAttraction_v(0.4, 0.0, PS_CLOSEST_ZONE);

        NodeVec2
        MS_NOISE_VECTOR = new v_Noise_(20, seed);

        /*
        public v_Swirl_vav(double soe, double s, NodeVec2Array im1, NodeVec2 im2)
        Parameters:
        soe - double, the sphere of influence beyond which the hazards are not considered.
        s - double, the safety zone, inside of which a maximum repulsion from the object is generated.
        im1 - NodeVec2Array, the embedded perceptual schema that generates a list of items to avoid.
        im2 - NodeVec2, the embedded perceptual schema that generates a pointer to the goal (must be egocentric).
        */
        NodeVec2
        MS_SWIRL_CLOSEST_ZONE = new v_SwirlLeft_va(2.0, abstract_robot.RADIUS + 0.2, PS_OBS);

        NodeVec2
        MS_SWIRL_TO = new v_Swirl_vav(2.0, abstract_robot.RADIUS + 0.2, PS_OBS, PS_ATTRACTOR);


        ////////////////////////////////////////////////////////////////////////////////////////////
        //          COMPORTAMENTI JAVA DEL ROBOT                    //
        ////////////////////////////////////////////////////////////////////////////////////////////

        /*************************************
                AS_GOTO
        *************************************/
        // Azione di movimento verso un punto con destinazione variabile

        v_StaticWeightedSum_va
        AS_GOTO = new v_StaticWeightedSum_va();

        AS_GOTO.weights[0]  = avoidgain.Value(abstract_robot.getTime());
        AS_GOTO.embedded[0] = MS_AVOID_OBSTACLES;

        AS_GOTO.weights[1]  = noisegain.Value(abstract_robot.getTime());
        AS_GOTO.embedded[1] = MS_NOISE_VECTOR;

        AS_GOTO.weights[2]  = mtggain.Value(abstract_robot.getTime());
        AS_GOTO.embedded[2] = MS_MOVE_TO;

        AS_GOTO.weights[3]  = swirlgain.Value(abstract_robot.getTime());
        AS_GOTO.embedded[3] = MS_SWIRL_TO;


        /*************************************
                AS_GO_FLAG
        *************************************/
        // Azione di movimento verso la bandierina

        v_StaticWeightedSum_va
        AS_GO_FLAG = new v_StaticWeightedSum_va();

        AS_GO_FLAG.weights[0]  = avoidgain.Value(abstract_robot.getTime());
        AS_GO_FLAG.embedded[0] = MS_AVOID_OBSTACLES;

        AS_GO_FLAG.weights[1]  = noisegain.Value(abstract_robot.getTime());
        AS_GO_FLAG.embedded[1] = MS_NOISE_VECTOR;

        AS_GO_FLAG.weights[2]  = mtggain.Value(abstract_robot.getTime());
        AS_GO_FLAG.embedded[2] = MS_MOVE_TO_FLAG;
```

```
/*************************************
      AS_GO_CLOSEST_ZONE
*************************************/
// Azione di movimento verso la zona della mappa non ancora esplorata

v_StaticWeightedSum_va
AS_GO_CLOSEST_ZONE = new v_StaticWeightedSum_va();

AS_GO_CLOSEST_ZONE.weights[0]  = mtggain.Value(abstract_robot.getTime());
AS_GO_CLOSEST_ZONE.embedded[0] = MS_MOVE_TO_CLOSEST_ZONE;

AS_GO_CLOSEST_ZONE.weights[1]  = avoidgain.Value(abstract_robot.getTime());
AS_GO_CLOSEST_ZONE.embedded[1] = MS_AVOID_OBSTACLES;

AS_GO_CLOSEST_ZONE.weights[2]  = noisegain.Value(abstract_robot.getTime());
AS_GO_CLOSEST_ZONE.embedded[2] = MS_NOISE_VECTOR;

AS_GO_CLOSEST_ZONE.weights[3]  = swirlgain.Value(abstract_robot.getTime());
AS_GO_CLOSEST_ZONE.embedded[3] = MS_SWIRL_CLOSEST_ZONE;



// Viene consultato il file che contiene l'interprete sGOLOG
File eclipseProgram = new File(".\\Database\\sGolog.pl");
client = new sGologClient(eclipseProgram);

// Viene consultato il file che contiene database sGOLOG utilizzato nella competizione
eclipseProgram = new File(".\\Database\\Competizione.pl");
client.consultFile(eclipseProgram);

// La classe SGologPlan serve per eseguire il CAT generato con l'utilizzo della classe sGologClient
plan = new SGologPlan(abstract_robot);

// Se il parametro e' true, visualizza in output l'albero generato da sGOLOG
plan.debugging(false);

// Matching tra azioni primitive sGOLOG e azioni corrispondenti agli stati del robot
plan.addAction("goto",
                        AS_GOTO,
                        AS_GOTO,
                        PF_CLOSE_TO,
                        0,
                        new int[]{1,3,5,7,9,11,13,15},
                        NodeAction.LASER_ON);

plan.addAction("wander",
                        AS_GO_CLOSEST_ZONE,
                        AS_GO_CLOSEST_ZONE,
                        PF_SOMETHING_VISIBLE,
                        0,          // chiudo comunque il gripper nello stato di wander
                        new int[]{0,2,4,6,8,10,12,14},
                        NodeAction.LASER_ON);

plan.addAction("go_flag",
                        AS_GO_FLAG,
                        AS_GO_FLAG,
                        PF_TRIGGER_FLAG,
                        -1,
                        new int[]{},
                        NodeAction.LASER_OFF);

plan.addAction("open_gripper",
                        NodeAction.NO_STEERING,
                        NodeAction.NO_TURRET,
                        NodeAction.IMMEDIATE_TRIGGER,
                        1,
                        new int[]{},
                        NodeAction.LASER_OFF);

plan.addAction("send_message",
                        NodeAction.NO_STEERING,
```

```
                        NodeAction.NO_TURRET,
                        NodeAction.IMMEDIATE_TRIGGER,
                        NodeAction.NO_GRIPPER,
                        new int[]{},
                        NodeAction.LASER_OFF);

plan.addAction("wait",
                        AS_GOTO,
                        AS_GOTO,
                        PF_OTHER_ROBOT,
                        NodeAction.NO_GRIPPER,
                        new int[]{},
                        NodeAction.LASER_OFF);


// Matching tra fluenti di sGOLOG e trigger di TeamBots
plan.addAction("something_visible", PF_SOMETHING_VISIBLE);

plan.addAction("red_flag", PF_RED_FLAG_IN_GRIPPER);

plan.addAction("green_flag", PF_GREEN_FLAG_IN_GRIPPER);

plan.addAction("blue_flag", PF_BLUE_FLAG_IN_GRIPPER);

plan.addAction("can_go", PF_OTHER_ROBOT);

plan.addAction("load", PF_SOMETHING_IN_GRIPPER);

plan.addAction("in_red_lobby", PF_IN_RED_LOBBY);

plan.addAction("in_green_lobby", PF_IN_GREEN_LOBBY);

plan.addAction("in_blue_lobby", PF_IN_BLUE_LOBBY);


// Genera il CAT
plan.setPlan(client.invoke("do(execute,s0,S)"), 0);
/* E' il piano generico all'invocazione di execute

?- do(execute, s0, S).

S = [sense(load), [load, [sense(red_flag), [red_flag, [sense(in_red_lobby), [in_red_lobby, [goto(red_attractor),
send_message(am_at_attractor), sense(can_go), [can_go, [goto(...), ...], [...]]], [sense(in_green_lobby),
[in_green_lobby, [goto(green_door), goto(...), ...], [sense(...), ...]]]]], [sense(green_flag), [green_flag,
[sense(in_green_lobby), [in_green_lobby, [goto(green_attractor), send_message(...), ...], [sense(...), ...]]],
[sense(blue_flag), [blue_flag, [sense(...), ...], [...]]]]]], [sense(something_visible), [something_visible,
[go_flag, sense(red_flag), [red_flag, [sense(in_red_lobby), [in_red_lobby, [goto(...), ...], [...]]],
[sense(green_flag), [green_flag, [...], ...]]]], [wander, go_flag, sense(red_flag), [red_flag, [sense(in_red_lobby),
[in_red_lobby, ...]], [sense(green_flag), [...]]]]]]]
*/


/*****************************************************************************************
                        STATE_MACHINE
*****************************************************************************************/
// FSA del robot, costituito da un unico stato stato sGOLOG (viene ripetuto ciclicamente)

STATE_MACHINE = new i_FSA_ba();

STATE_MACHINE.state = 0;

STATE_MACHINE.triggers[0][0]  = plan.isFinished(abstract_robot.getTime());
STATE_MACHINE.follow_on[0][0] = 0;  // al termine dello stato, viene eseguito il loop sullo stesso

state_monitor = STATE_MACHINE;


/************************************
STEERING
************************************/

v_Select_vai
```

```
        STEERING = new v_Select_vai((NodeInt) STATE_MACHINE);

        STEERING.embedded[0] = plan.getSteering(abstract_robot.getTime());


        /*************************************
        TURRET
        *************************************/

        v_Select_vai
        TURRET = new v_Select_vai((NodeInt) STATE_MACHINE);

        TURRET.embedded[0] = plan.getTurret(abstract_robot.getTime());


        /*************************************
        GRIPPER_FINGERS
        *************************************/

        d_Select_i
        GRIPPER_FINGERS = new d_Select_i(STATE_MACHINE);

        GRIPPER_FINGERS.embedded[0] = plan.getGripperFingers(abstract_robot.getTime());


        /*************************************
        SONAR_CONFIGURATION
        *************************************/
        // Associo il comportamento del sonar all'FSA

        d_SonarControl_ir
        SONAR_CONFIGURATION = new d_SonarControl_ir(STATE_MACHINE, abstract_robot);

        SONAR_CONFIGURATION.sonarActived[0]    = plan.getSonar(abstract_robot.getTime());
        SONAR_CONFIGURATION.sonarPrecision[0] = SonarObjectSensor.MEDIUM;


        /*************************************
        LASER_CONFIGURATION
        *************************************/
        // Associo il comportamento del laser all'FSA

        b_Select_ir
        LASER_CONFIGURATION = new b_Select_ir(STATE_MACHINE, abstract_robot);

        LASER_CONFIGURATION.embedded[0] = plan.getLaser(abstract_robot.getTime());


        /* Adesso associo questi comportamenti alle variabili globali che selezioneranno
           il comportamento adeguato relativamente allo stato in cui il robot si trova
        */
        laser_configuration = LASER_CONFIGURATION;
        sonar_configuration = SONAR_CONFIGURATION;
        turret_configuration = TURRET;
        steering_configuration = STEERING;
        gripper_fingers_configuration = GRIPPER_FINGERS;

        id = abstract_robot.getPlayerNumber(abstract_robot.getTime());
        other_id = ((id==0) ? 1 : 0);

}       /* end configure() */


// takeStep e' la funzione chiamata per ogni unita' di tempo dal simulatore
public int takeStep(){

        long    curr_time = abstract_robot.getTime();
        String  action;

        int state = STATE_MACHINE.Value(curr_time);

        MAP.setCellVisited(curr_time);                  // sfruttato per evitare di passare sulle celle gia' visitate
```

```
MAP.setObstacle(PS_SONAR.Value(curr_time), abstract_robot.getPrecisionRivelation(), curr_time);
MAP.setObstacle(PS_LASER.Value(curr_time), LaserFinderObjectSensor.FUZZY, curr_time);

/**********
Procedura per liberare il robot dalle zone chiuse
**********/

curr_pos = abstract_robot.getPosition(curr_time);              // posizione attuale, ad ogni takestep

// Vengono usati nel caso in cui serva una tolleranza con un massimo di due cifre decimali
long x_pos = (Math.round(curr_pos.x * 100)) / 100;
long y_pos = (Math.round(curr_pos.y * 100)) / 100;
long x_pos_temp = (Math.round(curr_pos_temp.x * 100)) / 100;
long y_pos_temp = (Math.round(curr_pos_temp.y * 100)) / 100;

double toll = 0.15;   // tolleranza entro la quale il robot viene considerato fermo nella stessa posizione

if(counter_pos != 0){
        if((x_pos >= x_pos_temp-toll && x_pos <= x_pos_temp+toll) && (y_pos >= y_pos_temp-toll && y_pos <=
                                                                                x_pos_temp+toll)
){
                ++counter_pos;
                if(counter_pos == 500){
        /* Libera una matrice di celle attorno al robot, segnate come occupate, in modo da poter farlo uscire
           dalle zone in cui si potrebbe chiudere
        */
                        for(int i= -3; i<=3; i++){
                                for(int j= -5; j<=5; j++){
                                        MAP.setCellState(x_pos + i, y_pos + j, 0);
                                }
                        }
                /* Al momento dell'azzeramento, il robot ruota su se stesso di 120 gradi, per avere la
possibilita'
                   di uscire da dove era venuto
                */
                        abstract_robot.setSteerHeading(curr_time,
                                                abstract_robot.getSteerHeading(curr_time)+Math.PI*2/3);
                        counter_pos = 0;
                }
                counter_pos_2 = 0;
        }else{
                curr_pos_temp = curr_pos;
                ++counter_pos_2;
                if(counter_pos_2 == 300){
        /* Libera una matrice di celle attorno al robot, segnate come occupate, in modo da poter farlo uscire
           dalle zone in cui si potrebbe chiudere
        */
                        for(int i= -1; i<=1; i++){
                                for(int j= -4; j<=4; j++){
                                        MAP.setCellState(x_pos + i, y_pos + j, 0);
                                }
                        }
                /* Al momento dell'azzeramento, il robot ruota su se stesso di 120 gradi, per avere la
possibilita'
                   di uscire da dove era venuto
                */
                        abstract_robot.setSteerHeading(curr_time,
                                                abstract_robot.getSteerHeading(curr_time)+Math.PI*2/3);
                        counter_pos_2 = 0;
                }
        }
}else{
        ++counter_pos;
}

/**********
Procedura per capire se sono dentro la chiocciola
**********/

if(!am_in_snail){     // am_in_snail e' un flag che indica se il robot e' dentro la chiocciola
        // Effettuo un test sulle coordinate dei punti che delimitano l'area della chiocciola
        if((x_pos > 19.0 && x_pos <= 25.0) && (y_pos > 8.0 && y_pos < 18.0)){
```

```
                        am_in_snail = true;

                        // invoco la procedura snail del piano sGOLOG per gestire la chiocciola
                        plan.setPlan(client.invoke("do(snail,s0,S)"), 0);
/* E' il piano generico all'invocazione di snail, la procedura per gestire la chiocciola

  ?- do(snail, s0, S).

  S = [sense(load), [load, [goto(snail_1), sense(load), [load, [goto(snail_2), sense(load), [load, [goto(snail_3),
  sense(load), [load, [...], ...]], [goto(snail_3), sense(load), [load, ...]]]], [goto(snail_2), sense(load),
  [load, [goto(snail_3), sense(load), [load, ...]], [goto(snail_3), sense(load), [...]]]]]], [sense(something_visible),
  [something_visible, [go_flag, goto(snail_1), sense(load), [load, [goto(snail_2), sense(load), [load, ...]],
  [sense(something_visible), [something_visible, ...]]]], [goto(snail_1), sense(load), [load, [goto(snail_2),
  sense(load), [load, ...]], [sense(something_visible), [something_visible, ...]]]]]]]
*/
            }
}
if(am_in_snail && (x_pos >= 22.0 && x_pos <= 24.0) && (y_pos >= 20.0 && y_pos <= 22.0)){
            am_in_snail = false;
            plan.setPlan(client.invoke("do(execute,s0,S)"), 0);
}

/**********
Procedura per capire se sono dentro le stanze e in quali stanze sono
**********/

if(!am_in_snail){     // il robot puo' trovarsi dentro le anticamere solamente se non e' gia' dentro la chiocciola
            if(x_pos < 11 || x_pos > 33){
                        if(y_pos >= 22){
                                    PF_IN_RED_LOBBY.setValue(true);
                                    PF_IN_GREEN_LOBBY.setValue(false);
                                    PF_IN_BLUE_LOBBY.setValue(false);
                        }
                        if(y_pos >= 10 && y_pos <= 18){
                                    PF_IN_RED_LOBBY.setValue(false);
                                    PF_IN_GREEN_LOBBY.setValue(true);
                                    PF_IN_BLUE_LOBBY.setValue(false);
                        }
                        if(y_pos <= 6){
                                    PF_IN_RED_LOBBY.setValue(false);
                                    PF_IN_GREEN_LOBBY.setValue(false);
                                    PF_IN_BLUE_LOBBY.setValue(true);
                        }
                        if(((y_pos > 18 && y_pos < 22) || (y_pos > 6 && y_pos < 10)) && (x_pos > 6 && x_pos < 11)){
                                    abstract_robot.setGripperFingers(curr_time, 1);
                                    gripper_open = true;
                        }else{
                                    if(gripper_open){
                                                plan.setPlan(client.invoke("do(execute,s0,S)"), 0);
                                                gripper_open = false;
                                    }
                        }
            }else{
                        PF_IN_RED_LOBBY.setValue(false);
                        PF_IN_GREEN_LOBBY.setValue(false);
                        PF_IN_BLUE_LOBBY.setValue(false);
            }
}

/*********/


// Aggiorna il piano e dopo...
plan.update(curr_time, state);
// ... controlla se si e' passati da un'azione sGOLOG ad un'altra
if(plan.isChangedState()){
            action = plan.getActionName();            // action e' una stringa che contiene il nome dell'azione corrente
            // Gestisce in base all'azione corrente l'asseganzione dinamica dei punti del comportamento AS_GOTO
            if(action.startsWith("goto(")){
                        if(plan.getParameter()[0].equals("red_bin")){
                                    PS_ATTRACTOR.setValue(curr_time, 2.5, 23.0);
                        }else
```

```
                                        if(plan.getParameter()[0].equals("green_bin")){
                                                PS_ATTRACTOR.setValue(curr_time, 2.5, 14.0);
                                        }else
                                        if(plan.getParameter()[0].equals("blue_bin")){
                                                PS_ATTRACTOR.setValue(curr_time, 2.5, 5.0);
                                        }else
                                        if(plan.getParameter()[0].equals("red_attractor")){
/* Le posizioni degli attractor sono sfalsate di 0.4 per red e blu e 1.2 per green, in modo che i robot possano arrivare al punto di
send_message senza problemi di ingombro
*/
                                                PS_ATTRACTOR.setValue(curr_time, 8.0, 25.2);
                                        }else
                                        if(plan.getParameter()[0].equals("green_attractor")){
                                                PS_ATTRACTOR.setValue(curr_time, 6.0, 14.6);
                                        }else
                                        if(plan.getParameter()[0].equals("blue_attractor")){
                                                PS_ATTRACTOR.setValue(curr_time, 8.0, 3.2);
                                        }else
                                        if(plan.getParameter()[0].equals("red_door")){
                                                PS_ATTRACTOR.setValue(curr_time, 11.5, 22.5);
                                        }else
                                        if(plan.getParameter()[0].equals("green_door")){
                                                PS_ATTRACTOR.setValue(curr_time, 11.5, 14.0);
                                        }else
                                        if(plan.getParameter()[0].equals("blue_door")){
                                                PS_ATTRACTOR.setValue(curr_time, 11.5, 5.5);
                                        }else
                                        if(plan.getParameter()[0].equals("to_field_from_red")){
                                                PS_ATTRACTOR.setValue(curr_time, 14.0, 22.5);
                                        }else
                                        if(plan.getParameter()[0].equals("to_field_from_green")){
                                                PS_ATTRACTOR.setValue(curr_time, 14.0, 14.0);
                                        }else
                                        if(plan.getParameter()[0].equals("to_field_from_blue")){
                                                PS_ATTRACTOR.setValue(curr_time, 14.0, 5.5);
                                        }else
                                        if(plan.getParameter()[0].equals("snail_1")){
                                                PS_ATTRACTOR.setValue(curr_time, 20.0, 10.5);
                                        }else
                                        if(plan.getParameter()[0].equals("snail_2")){
                                                PS_ATTRACTOR.setValue(curr_time, 20.2, 15.6);
                                        }else
                                        if(plan.getParameter()[0].equals("snail_3")){
                                                PS_ATTRACTOR.setValue(curr_time, 22.0, 15.0);
                                        }else
                                        if(plan.getParameter()[0].equals("snail_4")){
                                                PS_ATTRACTOR.setValue(curr_time, 23.0, 13.0);
                                        }else
                                        if(plan.getParameter()[0].equals("snail_5")){
                                                PS_ATTRACTOR.setValue(curr_time, 24.0, 17.5);
                                        }else
                                        if(plan.getParameter()[0].equals("snail_6")){
                                                PS_ATTRACTOR.setValue(curr_time, 23.0, 21.0);
                                        }
                                }else
                                /* Altrimenti gestisce in base all'azione corrente (ovvero ho raccolto la bandierina e sono "Pronto")
                                   l'invio dei messaggi all'altro compagno
                                */
                                if(action.startsWith("send_message(am_at_attractor")){
                                        try{
                                                LongMessage idMessage = new LongMessage();
                                                idMessage.val = abstract_robot.getPlayerNumber(curr_time);
                                                abstract_robot.unicast(other_id, idMessage);
                                                StringMessage message = new StringMessage();
                                                message.val = "Pronto";
                                                abstract_robot.unicast(other_id, message);
                                        }
                                        catch(CommunicationException e){
                                        }
                                }else
                        /* Gestisce in base all'azione corrente il settaggio a true o false della variabile PF_OTHER_ROBOT
                           usata come trigger per capire se il robot puo' andare (can_go) o meno.
```

```
                    Con il test sull'azione wander sorgono degli inconvenienti, in quanto PF_OTHER_ROBOT viene impostato
                    a false anche quando l'altro robot ancora in wander, per caso tornava di nuovo in wander dopo un tentativo
                    di go_flag fallito
                    Viene effettuato quindi il test su open_gripper cosi' e' sicuro che il settaggio a false della variabile
                    avviene solamente dopo che e' stata depositata la bandierina
                */
                        if(action.startsWith("open_gripper")){
                                PF_OTHER_ROBOT.setValue(false);
                        }
                }       /* end if(plan.isChangedState()) */

                if(bufferMessage.hasMoreElements()){
                        try{
                                LongMessage idMessage = (LongMessage) bufferMessage.nextElement();
                                if(idMessage.val == other_id && bufferMessage.hasMoreElements()){
                                        StringMessage message = (StringMessage) bufferMessage.nextElement();
                                        if(message.val == "Pronto"){
                                                PF_OTHER_ROBOT.setValue(true);
                                        }
                                }
                        }
                        catch(ClassCastException e){
                        }
                }       /* end if(bufferMessage.hasMoreElements()) */

                 // Commentare la successiva linea per poter visualizzare le azioni sGOLOG
                abstract_robot.setDisplayString("TUX 1 LEFT");

                return CSSTAT_OK;
        }       /* end takeStep() */
}       /* end classe TUX_1_LEFT */
```