INTERFACCIA TEAMBOTS - SGOLOG

REALIZZATO DURANTE IL CORSO DI ROBOTICA 2002/2003

DEL PROF. ING. ANTONIO CHELLA E DOTT. ING. ROSARIO SORBELLO

PRESSO L'UNIVERSITÀ DI PALERMO FACOLTÀ DI INGEGNERIA INFORMATICA

DAGLI ALLIEVI INGEGNERI MARCO DI STEFANO E CARMELO SCOZZOLA

INDICE

Introduzione

Lo scopo di questa tesina è quello di sfruttare le potenzialità del ***situation calculus*** all'interno del simulatore TeamBots. Si è scelto di utilizzare il linguaggio sGOLOG, discendente del GOLOG, un linguaggio di programmazione logica che, in aggiunta alle azioni primitive del ***situation calculus***, permette la definizione di azioni complesse attraverso l'uso di costrutti di programmazione noti dai convenzionali linguaggi di programmazione. I programmi in sGOLOG sono quelli GOLOG con l'aggiunta di azioni di percezione per i termini. La differenza principale, se paragonato con il GOLOG originale, è che l'interprete adesso produce un albero binario di azioni (*Conditional action tree*), dove ogni percorso lungo esso rappresenta una sequenza lineare di situazioni.

Per poter sfruttare le potenzialità di sGOLOG all'interno di TeamBots, si è deciso di utilizzare l'interprete Prolog Ecl^{i}ps^{e} il quale fornisce un insieme di classi Java, contenute all'interno del package *com.parctechnologies.eclipse*, che costituiscono appunto una interfaccia tra Java ed Ecl^{i}ps^{e}. Si è cercato infine di adattare tali strategie ad un possibile uso in TeamBots.

La sezione 1 contiene le nozioni principali per introdurre il linguaggio sGOLOG. Nella sezione 2 viene data una breve spiegazione di come è stato realizzato l'interfacciamento tra Java ed Ecl^{i}ps^{e}. Infine, nelle sezioni 3 e 4, vengono presentate le classi realizzate per implementare tali strategie, e viene fornito un esempio completo di realizzazione di un robot con uno stato creato a partire da un database sGOLOG.

1. sGOLOG

L'interprete sGOLOG estende GOLOG attraverso l'aggiunta di azioni di percezione per percepire il valore di verità di termini. Piuttosto che produrre una sequenza lineare di azioni primitive, l'interprete sGOLOG genera un albero di azioni, se ne esiste uno, con l'idea che la scelta dei rami da seguire sia condizionata dai risultati di azioni di percezione.
Nel seguito verrà presentata una breve introduzione sul GOLOG e sulle innovazioni introdotte da sGOLOG. Viene anche mostrato un semplice esempio sull'utilizzo di sGOLOG.

1.1. Cenni sul GOLOG

GOLOG è un linguaggio di programmazione logica che, in aggiunta alle azioni primitive del ***situation calculus***, permette la definizione di azioni complesse attraverso l'uso di costrutti noti dai convenzionali linguaggi di programmazione. Ecco una lista dei costrutti disponibili in GOLOG:

- A azione primitiva
- $\phi?$ test
- $(\rho_1;\rho_2)$ sequenza
- $(\rho_1 \mid \rho_2)$ non-determinismo di azioni
- $(\pi x.\rho)$ non-determinismo di argomenti
- $\rho*$ non-determinismo di ciclo
- ***if*** ϕ ***then*** ρ_1 ***else*** ρ_2 ***endif*** condizione
- ***while*** ϕ ***do*** ρ ***endwhile*** ciclo
- ***proc*** ρ(x) ***endproc*** procedura

Ciò che particolarizza il GOLOG è che il significato di questi costrutti è completamente definito da proposizioni nel ***situation calculus***. Per questo scopo, viene introdotta una macro $Do(\rho, s, s')$ il cui significato intuitivo è quello di eseguire il programma ρ nella situazione s per raggiungere la situazione s'.
Data una teoria del ***situation calculus*** AX su un certo dominio, eseguire un programma ρ vuol dire trovare prima una sequenza di azioni primitive $\boldsymbol{a}$ come:

$$AX \models Do(\rho, S_0, do(\mathbf{a}, S_0))$$

e poi far manipolare la sequenza $\boldsymbol{a}$ ad un appropriato modulo che si prende cura di eseguire l'attuazione di queste azioni nel mondo reale.

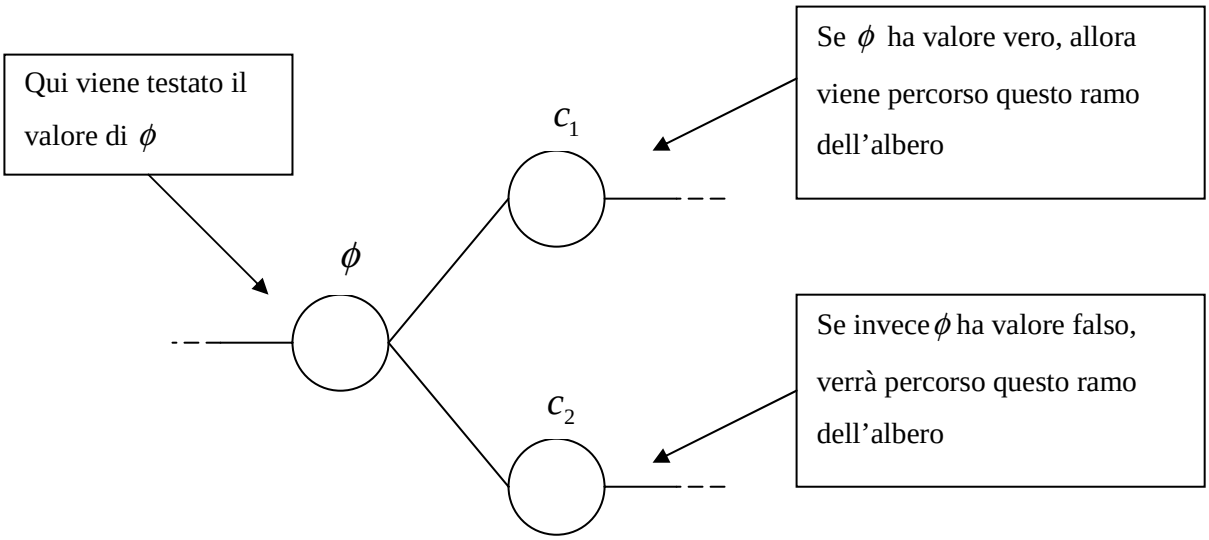
1.2. Conditional action trees (CAT)

L'idea di base di sGOLOG è che, invece di avere soltanto una storia di azioni lineare (i.e. situazioni), l'interprete produca un albero binario di azioni, dove ogni percorso lungo esso rappresenti una sequenza di situazioni. A questo scopo sono state introdotte due nuove classi, una classe di formule per quelle formule che non dipendono dalle situazioni ed una classe CAT (*Conditional action tree*) per alberi di azioni condizionali. Si usano ϕ e c con possibili sub- o super-scripts per classi del tipo formula e CAT, rispettivamente.

I termini CAT sono costituiti da:

- una costante speciale ε, che denota il CAT vuoto;
- le azioni primitive;
- due costruttori $a \cdot c$ e $[\phi, c_1, c_2]$, dove a è una azione, ϕ è un termine di una certa classe di formule e c, c_1 e c_2 sono essi stessi CAT.

ϕ viene anche chiamata *branch-formula*, c_1 e c_2 sono chiamati rispettivamente *true-branch* e *false-branch* (per ϕ).



I programmi in sGOLOG sono quelli GOLOG con l'aggiunta di azioni di percezione per i termini. La differenza principale, se paragonato con il GOLOG originale, è che l'interprete adesso produce CAT invece che una sequenza lineare di situazioni.

L'introduzione di nuovi *branch* viene lasciata sotto il controllo dell'utente attraverso l'azione speciale $branch\_on(\phi)$, il cui effetto è quello di creare un nuovo CAT $[\phi, \varepsilon, \varepsilon]$. Poiché vogliamo che sGOLOG produca dei CAT che siano pronti per l'esecuzione, abbiamo bisogno di essere sicuri che il valore di verità della formula che decide quale *branch* percorrere sia noto.

Per questo motivo (ed anche per altre ragioni legate alla implementazione di sGOLOG in Prolog) viene ristretto l'insieme di azioni di percezione ed i loro usi ai seguenti casi:

- soltanto il valore di verità di fatti atomici può essere percepito. In particolare le azioni di percezione hanno la forma $sense(P)$, dove P è un fluente;
- il valore di verità di un fluente non è mai utilizzato prima che la corrispondente azione di percezione sia stata eseguita;
- le azioni $branch\_on(P)$ sono permesse soltanto nel caso in cui P sia un fluente percepito;
- ogni volta che viene raggiunta una azione $branch\_on(P)$, entrambi i valori di verità sono assegnabili a P.

Se è possibile che P possa assumere entrambi i valori di verità, si può tranquillamente ipotizzare una pianificazione per entrambi i casi quando si valuta $branch\_on(P)$. L'idea è che il *true-branch* $C1$ del CAT $[P, C1, C2]$ da una azione $branch\_on(P)$, è costruito assumendo che P sia vero nella situazione corrente. In modo simile, il *false-branch* $C2$ è sviluppato assumendo che P sia falso nella situazione corrente. Questo è stato implementato introducendo nuove azioni primitive $assm(P,1)$ e $assm(P,0)$ il cui solo effetto è di settare il valore di P a vero e falso, rispettivamente. Da notare che le azioni che avvengono dopo $assm$ possono modificare il valore di verità di P.

Tecnicamente l'interprete sGOLOG è definito in modo molto simile al GOLOG originale. Per la pianificazione viene utilizzata una macro di tre argomenti $Do(\rho, s, c)$ che si espande in una formula del ***situation calculus*** che ha per argomento dei CAT. Potrebbe essere letta come "esegui il programma ρ nella situazione s per produrre il CAT c". Da notare la differenza con la macro Do originale, dove l'ultimo argomento era una situazione piuttosto che un CAT.

1.3. Un semplice esempio

Per meglio spiegare l'utilizzo di sGOLOG si illustra un esempio classico di utilizzo.
Si supponga che il nostro agente si trovi all'aeroporto, ma non sappia ancora quale sia il gate verso cui deve dirigersi per prendere l'aereo. Prima di salire a bordo, l'agente vuole acquistare un giornale e prendere un caffè. Nel caso in cui debba dirigersi verso il gate A è preferibile che prenda il caffè al gate, altrimenti se deve dirigersi verso il gate B (per semplicità supporremo che ci siano soltanto due gate) conviene prendere il caffè prima. Viene introdotto il fluente *it_is_gate_A*, il cui valore di verità può essere percepito e che ci dice verso quale gate l'agente deve dirigersi; si introducano inoltre le seguenti azioni primitive:

- *goto(x)*: dirigiti verso il gate *x*;
- *buy_paper*: compra il giornale;
- *buy_coffee*: prendi il caffè;
- *board_plane*: sali a bordo dell'aereo;
- *sense(x)*: percepisci il valore di verità del fluente *x*.

Tutte le azioni sono possibili in qualunque situazione (non richiedono quindi precondizioni), tranne *board_plane* che richiede che l'agente si trovi al gate giusto per poter prendere l'aereo. Si introducano infine gli assiomi di stato successore per dirigersi verso il gate assegnato, e la procedura *catch_plane*, che viene utilizzata per effettuare la pianificazione:

```
proc(catch_plane,
            (sense(it_is_gate_A):
            buy_paper:
            branch_on(it_is_gate_A):
            if(it_is_gate_A,
                  goto(gate_A):buy_coffee,
                  buy_coffee:goto(gate_B)):
            board_plane)
      ).
```

Tale procedura può essere letta come "controlla se devi dirigerti verso il gate A e compra il giornale (questa azione viene eseguita comunque, sia che l'agente debba dirigersi verso il gate A, sia che debba dirigersi verso il gate B); se devi andare al gate A, dirigiti là, prendi il caffè e sali a bordo dell'aereo, altrimenti prendi prima il caffè e poi dirigiti verso il gate B".

Il database sGOLOG è il seguente:

```
/* dichiarazione delle azioni primitive */
primitive_action(goto(_)).
primitive_action(buy_paper).
primitive_action(buy_coffee).
primitive_action(board_plane).
primitive_action(sense(_)).

/* tutte le azioni sono sempre possibili eccetto board_plane */
/* che richiede che ci si trovi al gate corretto */
poss(goto(_),_).
poss(buy_paper,_).
poss(buy_coffee,_).
poss(sense(_),_).

/* salire a bordo dell'aereo richiede che ci si trovi al gate giusto */
poss(board_plane,S) :-  holds(it_is_gate_A,S) ->
                        holds(am_at(gate_A),S) ; holds(am_at(gate_B),S).

/* assiomi di stato successore */
/* Mi trovo al gate X se ci sono appena andato oppure */
/* se mi trovavo già là e non sono andato da nessuna altra parte */
holds(am_at(X),do(A,S)) :- A = goto(X);
                           (holds(am_at(X),S), not (A = goto(_))).
holds(it_is_gate_A,do(A,S))  :- holds(it_is_gate_A,S);
                           A = assm(it_is_gate_A,1).

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

/* procedura che esegue la pianificazione */

proc(catch_plane,
            (sense(it_is_gate_A):   /* percepisci il valore del fluente */
            buy_paper:
            branch_on(it_is_gate_A):  /* esegui un branch */
            if(it_is_gate_A,
                  goto(gate_A): buy_coffee, /* true-branch */
                  buy_coffee: goto(gate_B)): /* false branch */
                  board_plane)
).
```

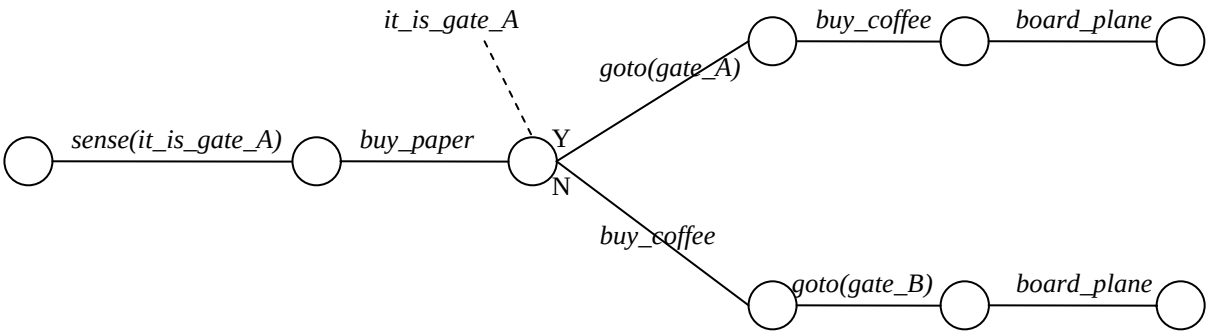
Se adesso si invoca dal terminale la seguente query:

```
> do(catch_plane, s0, S)
```

si ottiene il seguente output:

```
> S = [sense(it_is_gate_A), buy_paper,
            [it_is_gate_A, [goto(gate_A), buy_coffee, board_plane],
            [buy_coffee, goto(gate_B), board_plane]]]
```

È possibile rappresentare graficamente il CAT risultante con un albero i cui nodi sono situazioni (il nodo radice rappresenta la situazione iniziale $s0$) ed i cui rami sono invece le azioni che fanno transitare da una situazione ad un'altra:



2. Usare l'interfaccia Java – Ecl^{i}ps^e

Per poter sfruttare le potenzialità di sGOLOG all'interno di TeamBots, si è deciso di utilizzare l'interprete Prolog Ecl^{i}ps^e il quale fornisce un insieme di classi Java, contenute all'interno del package *com.parctechnologies.eclipse*, che costituiscono appunto una interfaccia tra Java ed Ecl^{i}ps^e. Nel seguito verranno introdotte le principali classi e relative caratteristiche del package.

2.1. Rappresentazione in Java dei dati Ecl^{i}ps^e

L'interfaccia Java – Ecl^{i}ps^e utilizza un insieme di convenzioni e classi Java in modo da fornire una rappresentazione comune dei dati di Ecl^{i}ps^e. La rappresentazione di dati Ecl^{i}ps^e è utile per:

- costruire goal sotto forma di termini da mandare ad Ecl^{i}ps^e per essere eseguiti;
- ottenere informazioni utili dai risultati di Ecl^{i}ps^e;
- comunicare termini attraverso *queues*.

La regola generale è quella di mandare dati ad Ecl^{i}ps^e come istanze di classi corrispondenti ai tipi di dati desiderati; il risultato fornito da Ecl^{i}ps^e sarà un oggetto della classe generica *java.lang.Object*, che può essere ricondotta ad una particolare classe di Java mediante una opportuna operazione di *cast* con il tipo di dato voluto.

2.2. Creazione e gestione di motori Ecl^{i}ps^{e} da Java

Per la creazione di un motore Ecl^{i}ps^{e} da Java viene utilizzata la classe *OutOfProcessEclipse*; in questo modo il motore Ecl^{i}ps^{e} è un processo figlio della JVM. La classe *OutOfProcessEclipse* implementa l'interfaccia *EclipseEngine* e viene istanziata usando un oggetto di tipo *EclipseEngineOptions*.

Configurare un oggetto di tipo *EclipseEngineOptions*

Prima che un motore Ecl^{i}ps^{e} sia creato, deve essere creato e configurato un oggetto di tipo *EclipseEngineOptions*. Un oggetto di tale classe rappresenta la configurazione delle nostre scelte per un nuovo motore Ecl^{i}ps^{e}. Le opzioni possono essere specificate sia passando al costruttore della un classe un oggetto di tipo *java.util.Properties*, sia invocando i metodi "*set*" della classe *EclipseEngineOptions*.

Uso della classe *OutOfProcessEclipse*

Quando si utilizza la classe *OutOfProcessEclipse*, il motore Ecl^{i}ps^{e} è un processo figlio della JVM. Le conseguenze importanti di questo modello sono:

- il motore Ecl^{i}ps^{e} usa memoria ed altre risorse separate dalla JVM, in base a come il sistema operativo alloca le risorse tra i processi;
- diverse istanze della classe *OutOfProcessEclipse* possono esistere contemporaneamente all'interno della stessa JVM.

Inizializzazione di un *OutOfProcessEclipse*: la classe *OutOfProcessEclipse* possiede un unico costruttore che riceve come parametro un oggetto di tipo *EclipseEngineOptions*. E' permessa l'istanziazione di *OutOfProcessEclipse* multipli.

Compilazione di un database Prolog: la classe *OutOfProcesEclipse* fornisce il metodo *compile* per consultare database Prolog. Nel seguente esempio viene creato un oggetto di tipo *EclipseEngineOptions* con un costruttore senza parametri (utilizzando quindi valori di default per le opzioni) e attraverso il metodo *setUseQueues* gli stream standard di Ecl^{i}ps^{e} (`stdin`, `stdout`, `stderr`) vengono connessi con gli stream standard della JVM. Poi viene creato il motore Ecl^{i}ps^{e} come oggetto della classe *OutOfProcesEclipse*:

```
...
    // Viene creato un nuovo oggetto di tipo EclipseEngineOptions
    EclipseEngineOptions
        eclipseEngineOptions = new EclipseEngineOptions();
    // Viene connesso lo stream standard di Eclipse a quello della JVM
    eclipseEngineOptions.setUseQueues(false);
    File database = new File("./MyDatabase”);
    try {
        OutOfProcessEclipse
            eclipse = new OutOfProcessEclipse(eclipseEngineOptions);
        eclipse.compile(database);
    }
    catch(IOException ioe) {
        System.out.println("File not found");
    }
    catch(EclipseException ee) {
        System.out.println("Eclipse exception: " + ee);
    }
...
```

Terminazione di un *OutOfProcessEclipse*: per terminare sia il processo Ecl^{i}ps^{e} viene invocato il metodo *destroy()* della classe *OutOfProcessEclipse*. Una volta che il metodo *destroy()* è stato invocato, l'invocazione di un qualunque altro metodo della classe *OutOfProcessEclipse* che richieda l'uso del motore Ecl^{i}ps^{e} lancerà una eccezione (*EclipseTerminatedException*). Tuttavia l'invocazione del metodo *destroy()* non impedisce la creazione di un nuovo *OutOfProcessEclipse* durante la stessa sessione della JVM

2.3. Esecuzione di un goal Ecl^{i}ps^{e} *da Java e processamento del risultato*

L'interfaccia *EclipseConnection* fornisce un unico metodo *rpc* (*Remote Predicate Call*) per l'esecuzione di goal in Ecl^{i}ps^{e}; questo metodo riceve come parametro il goal da eseguire.

La maniera più semplice di usare un *rpc* è di passare come parametro al metodo un oggetto di tipo *java.lang.String* che dovrebbe essere il goal così come lo scriveremmo sul terminale di Ecl^{i}ps^{e}.

Il metodo *rpc* ritorna un oggetto che implementa l'interfaccia *CompoundTerm*. Questo oggetto è la rappresentazione in Java del termine goal, con la sostituzione applicata alle variabili fornita dalla soluzione. Quest'ultima può essere scomposta utilizzando il metodo *arg* dell'oggetto *CompoundTerm* ritornato. Questo metodo riceve come parametro un intero (la posizione

dell'argomento all'interno del termine) e ritorna un oggetto di tipo *Object* che rappresenta il corrispondente in Java dell'argomento in Ecl^{i}ps^e in quella posizione.
Il codice seguente è un esempio di come un *CompoundTerm* ritornato può essere scomposto per estrarre la variabile voluta:

```
...
   // creiamo la nostra query
   String query = "do(execute, s0, S)";
   // result contiene il goal con le variabili unificate
   CompoundTerm result = eclipse.rpc(query);
   // a noi interessa il terzo argomento del goal
   Object goal = result.arg(3);
...
```

3. sGOLOG e TeamBots

A questo punto non resta che adattare al motore di TeamBots i risultati finora ottenuti.
L'idea di base è quella di inserire i piani generati da sGOLOG all'interno di stati dell'FSA, in modo da comprimere macrostati in un unico stato il cui andamento dipende dal particolare CAT generato e quindi dalle percezioni del robot sull'ambiente esterno. In questo modo si riduce notevolmente la complessità generale della definizione dell'automa a stati finiti. Bisogna inoltre considerare che il database sGOLOG è abbastanza generale e in fase di creazione non risente delle particolari implementazioni Java a cui verrà sottoposto; in questo modo è possibile pianificare sequenze di azioni piuttosto complesse senza lavorare effettivamente al codice del robot; verrà in un secondo tempo il lavoro di adattamento del robot al database sGOLOG, lavoro che si limiterà all'associazione di azioni definite all'interno del database sGOLOG con comportamenti del robot.
Vi è però una differenza sostanziale nell'interpretazione del CAT rispetto a quella presentata precedentemente; mentre un *Conditional action tree* è costituito da nodi che rappresentano situazioni e da archi che rappresentano le azioni che portano da una situazione ad un'altra, il corrispondente albero generato in Java è costituito da nodi che rappresentano stati del robot, quindi comportamenti, e da archi che rappresentano i trigger per transitare da uno stato ad un altro.
Risulta quindi necessario effettuare un *matching* fra azioni definite in sGOLOG e stati del robot.
Nel seguito verranno presentate le classi: *SGologClient*, *SGologPlan*, *NodeAction* ed *ActionList* che implementano le strategie descritte e che sono contenute all'interno del package *CSAI.unipa.SGolog*.

3.1. SGologClient

La classe *SGologClient* costituisce l'interfaccia tra TeamBots ed ECL^{i}PS^e. Essa utilizza le classi contenute all'interno del package *com.parctechnologies.eclipse* (fornito da ECL^{i}PS^e) che permettono di utilizzare l'interprete Prolog ECL^{i}PS^e direttamente da Java. In particolare, la classe *SGologClient* viene utilizzata per consultare database sGOLOG e per invocare query.

3.1.1. Costruttore

- **`public SGologClient`**`(File dataBase)`

È presente un solo costruttore il quale riceve come parametro il file contenente l'interprete sGOLOG. All'interno del costruttore viene creato il motore ECL^{i}PS^e e viene compilato il file contente l'interprete sGOLOG.

3.1.2. Metodi

La classe *SGologClient* mette a disposizione una serie di metodi per consultare file contenenti database sGOLOG, per invocare query e per togliere ed aggiungere fatti al database.

- **`public void consultFile`**`(File f)`

Il metodo *consultFile* serve per consultare un file contenente un database sGOLOG. Esso utilizza il metodo *compile* della classe *OutOfProcessEclipse*.

- **`public`** `List` **`invoke`**`(String query)`

Il metodo *invoke* permette l'esecuzione di goal. La generica query non è altro che un goal sGOLOG, quindi è una stringa del tipo *do(a, s0, C)* , dove *a* rappresenta la procedura principale che esegue la pianificazione, *s0* è lo stato iniziale della pianificazione *C* il CAT risultante. Il metodo *invoke* crea l'oggetto plan che conterrà il piano generato; l'invocazione vera e propria della query viene eseguita attraverso il metodo *rpc*. Il CAT ottenuto viene quindi convertito in una stringa e filtrato attraverso un tokenizer.

- **`public void add`**`(String fact)`

Questo metodo permette di aggiungere fatti al database; viene sempre utilizzato il metodo *rpc* che riceve in ingresso una stringa del tipo *assert(fatto)*, un costrutto del Prolog che serve appunto ad aggiungere fatti al nostro database.

- **`public void delete`**`(String fact)`

Questo metodo, analogo al precedente, permette di togliere fatti dal database Prolog; viene utilizzato il metodo *rpc* che riceve come parametro una stringa del tipo *retract(fatto)*, costrutto Prolog analogo al precedente.

3.2. SGologPlan

La classe *SGologPlan* si occupa dell'esecuzione di un CAT generato per mezzo della classe *SGologClient*. Tale piano viene memorizzato all'interno di una variabile sotto forma di array di stringhe. Nel seguito vengono presentate le classi *NodeAction* ed *ActionList* utilizzate da *SGologPlan* per effettuare il *matching* fra azioni del piano e comportamenti del robot.

NodeAction

La classe *NodeAction* rappresenta un singolo stato del robot. Viene utilizzata per effettuare una corrispondenza biunivoca fra nomi di azioni definite in un generico database sGOLOG e stati del robot, dove per stato del robot si intende la definizione di tutti i comportamenti delle singole parti di cui esso è costituito. Risulta inoltre necessario associare ad ogni azione, un trigger che ne indica la completa esecuzione. All'interno di questa classe sono contenuti una serie di campi che memorizzano: il nome dell'azione con cui lo stato deve essere associato, i vari comportamenti delle parti del robot ed il trigger che ne verifica la terminazione.
Inoltre la classe contiene dei campi statici utili per la definizione di particolari comportamenti:

- `public static final NodeVec2 NO_STEERING = null`
  usato per lasciare invariato il comportamento dello steering rispetto allo stato precedente;
- `public static final NodeVec2 NO_TURRET = null`
  usato per lasciare invariato il comportamento del turret rispetto allo stato precedente;
- `public static final int NO_GRIPPER = 1000`
  usato per lasciare invariato lo stato del gripper rispetto allo stato precedente;
- `public static final int[] NO_SONAR = new int[]{1000}`
  usato per lasciare invariato lo stato dei sonar rispetto allo stato precedente;
- `public static final int NO_LASER = -1`
  usato per lasciare invariato lo stato del laser rispetto allo stato precedente;
- `public static final int LASER_ON = 1`
  usato per porre il laser nello stato di scansione;
- `public static final int LASER_OFF = 0`
  usato per far terminare la scansione del laser;
- `public static final NodeBooleanDinamic IMMEDIATE_TRIGGER = new NodeBooleanDinamic(true)`
  usato per definire un trigger che ha valore immediatamente vero; risulta utile quando si devono definire azioni che avvengono istantaneamente come l'apertura\chiusura del gripper, spegnimento\accensione dei sonar o laser, o l'invio di messaggi.

ActionList

La classe *ActionList* mette a disposizione una serie di metodi per gestire una lista di oggetti di tipo *NodeAction*. Viene utilizzata all'interno della classe *SGologPlan* per effettuare il *matching* tra le azioni definite all'interno del piano sGOLOG e gli stati del robot definiti in TeamBots.

3.2.1. Campi principali

- `private List goal`: rappresenta il piano generato dalla classe SGologClient;
- `private String action` : il nome dell'azione corrente;
- `private ActionList actions` : contiene la lista delle azioni a cui si associano stati del robot;
- `private NodeBooleanDinamic finished` : trigger utilizzato per comunicare all'FSA del robot quando l'esecuzione del piano è terminata;
- `private int gologState` : stato dell'FSA a cui deve essere associata l'esecuzione del piano;
- `private List sensedTrigger` : lista dei trigger percepiti e di cui può essere eseguita l'azione *branch_on*.

3.2.2. Metodi

- `public void setPlan(List plan, int state)` : questo metodo riceve come parametri in ingresso il piano da eseguire e lo stato dell'FSA a cui deve essere associato;
- `public void addAction(String n, NodeVec2 steer, NodeVec2 turr, NodeBoolean tr, int grip)` : aggiunge un elemento alla lista di azioni *actions*;
- `public void addAction(String n, NodeVec2 steer, NodeVec2 turr, NodeBoolean tr, int grip, int[] son_conf, int las)` : aggiunge un elemento alla lista di azioni *actions* permettendo di definire anche lo stato di sonar e laser per il robot *MultiForageN150ExploreSim*;
- `public void addAction(String n, NodeBoolean tr)` : aggiunge un trigger alla lista di azioni da identificare con un'azione *sense*;
- `public String[] getParameter()`: ritorna la lista dei parametri dell'azione corrente;
- `public String getActionName()`: ritorna il nome sGOLOG dell'azione corrente;
- `public boolean isChangedState()`: ritorna il valore vero solamente quando si passa da uno stato ad un altro (da un'azione sGOLOG alla successiva);
- `public int update(long timestamp, int state)` : viene invocato dal metodo *takeStep()* e gestisce l'esecuzione del piano.

3.2.3. Funzionamento del metodo *update(long timestamp, int state)*

L'esecuzione del piano avviene per mezzo di un analizzatore lessicale che scorre le stringhe contenute nella lista *goal* e che si comporta nel seguente modo:

- se incontra la parola chiave *sense* ricerca l'azione da percepire nella lista *actions*, percepisce il valore del trigger ad essa associata e lo aggiunge alla lista *sensedTrigger*;
- se incontra il nome di un'azione, verifica se è presente in *actions* ed in caso positivo configura il robot con i comportamenti associati a quella particolare azione. A questo punto l'analisi lessicale (ad ogni invocazione del metodo) non procede fino a quando il trigger associato all'azione corrente non viene verificato;
- se incontra la parola chiave *branch_on*, ricerca l'azione associata, ne verifica la presenza all'interno della lista *sensedTrigger* e, dopo averne verificato il valore, esegue un taglio nell'albero eliminando fisicamente dal piano il *true_branch* o il *false_branch* a seconda che il valore del trigger sia rispettivamente falso o vero;
- appena arriva alla fine del piano assegna valore vero al trigger *finished*, il quale comunica all'FSA che il piano sGOLOG è terminato e che può quindi passare al successivo stato.

4. Esempio

Il seguente esempio mostra l'utilizzo delle classi *SGologClient* e *SGologPlan*, in cui due squadre, ciascuna composta da due robot, competono per la raccolta di bandierine su un campo di gara.
Per ogni squadra vengono realizzati due robot aventi un FSA con un unico stato il quale contiene un piano sGOLOG che viene ripetuto ciclicamente; in questo modo l'automa a stati finiti varia di volta in volta in base alle caratteristiche percepite dall'ambiente. I due robot lavorano in squadra per la raccolta di bandierine colorate ed il deposito nei contenitori dei rispettivi colori (rosso, verde e blu). Un robot non può depositare una bandierina se anche il suo compagno non ne ha raccolta una ma deve dirigersi di fronte al contenitore, comunicare al compagno di averne raccolta una e rimanere lì in attesa di un medesimo messaggio.

4.1. Il database sGOLOG

Analizziamo poco alla volta il database sGOLOG di uno dei robot (bisogna sottolineare il fatto che il database è uguale per tutti e 4 i robot).

Dobbiamo prima di tutto dichiarare le nostre azioni primitive:

- `primitive_action(goto(_)).` – spostamento sul campo di gioco;
- `primitive_action(wander).` – ricerca;

- `primitive_action(go_flag).` – spostamento verso la bandierina più vicina;
- `primitive_action(sense(_)).`– percezione di un trigger;
- `primitive_action(open_gripper).` – apertura del gripper;
- `primitive_action(send_message(_)).`– invio di un messaggio;
- `primitive_action(wait).` – attesa;

Tutte le azioni sono sempre possibili:

- `poss(goto(_),_).`
- `poss(wander,_).`
- `poss(sense(_),_).`
- `poss(open_gripper,_).`
- `poss(go_flag,_).`
- `poss(send_message(_),_).`
- `poss(wait,_).`

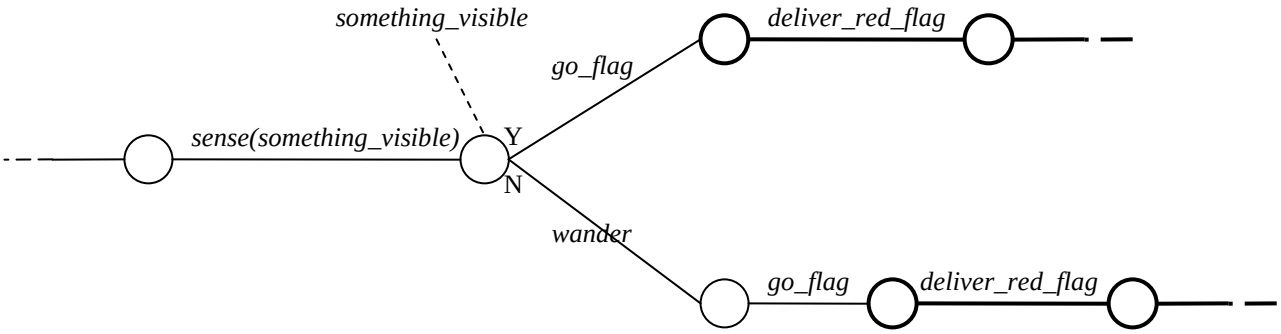
All'interno del database vengono definiti i seguenti fluenti:

- `something_visible` : assume valore vero se c'è almeno una bandierina in un intorno del robot; viene utilizzato per passare dallo stato di ricerca di target allo stato di raccolta;
- `red_flag` : assume valore vero quando il robot ha raccolto una bandierina di colore rosso; viene utilizzato per il deposito nel contenitore del colore corrispondente;
- `green_flag`: analogo a *red_flag*, ma per le bandierine verdi;
- `blue_flag` : analogo a *red_flag*, ma per le bandierine blu;
- `can_go` : assume valore vero quando entrambi i robot hanno raccolto una bandierina; viene utilizzato per poter passare alla fase di deposito;
- `red_attractor` : assume valore vero quando il robot si trova sull'attrattore posto davanti al contenitore rosso;
- `green_attractor` : analogo a *red_attractor*, ma per il contenitore verde;
- `blue_attractor` : analogo a *red_attractor*, ma per il contenitore blu.

Analizziamo adesso gli assiomi di stato successore:

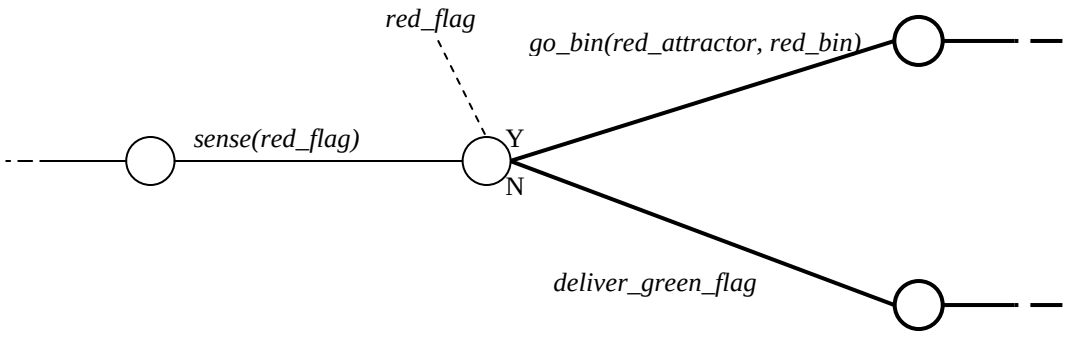
- ```
  holds(something_visible,do(A,S)):-
        holds(something_visible,S);
        A = assm(something_visible,1).
  ```
  something_visible ha valore vero nella situazione *S* se aveva già valore vero nella situazione precedente, oppure se l'azione che mi ha portato ad *S* ha assegnato al fluente il valore vero;
  ```

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

  red_flag ha valore vero nella situazione *S*, se nella situazione precedente *something_visible* aveva valore vero e l'azione eseguita è stata quella di dirigersi verso una bandierina, oppure se l'azione che mi ha portato ad *S* ha assegnato al fluente il valore vero;

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

  analogo a *red_flag*;

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

  analogo a *red_flag*;

- ```
  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).
  ```

  can_go ha valore vero se nella situazione precedente il robot ha raccolto una bandierina e si è diretto davanti al contenitore corrispondente (in attesa del messaggio del compagno) oppure se l'azione che mi ha portato ad *S* ha assegnato il valore vero a *can_go*.

Infine, nel database sono presenti le seguenti procedure di cui viene data una rappresentazione grafica dei CAT che generano (le parti disegnate in neretto sono dei CAT a loro volta):

- ```
  proc(execute,
        (sense(something_visible):
         branch_on(something_visible):
              if(something_visible,
                    go_flag,
                    wander:go_flag):
              deliver_red_flag)
  ).
  ```

  Questa procedura verifica se c'è qualche bandierina in un intorno del robot ed in caso positivo dice al robot di dirigersi verso la bandierina più vicina, altrimenti lo manda in
  ```

esplorazione e poi verso la bandierina più vicina (lo stato di ricerca termina quando il robot vede almeno una bandierina); in ogni caso, esegue la procedura *deliver_red_flag*;



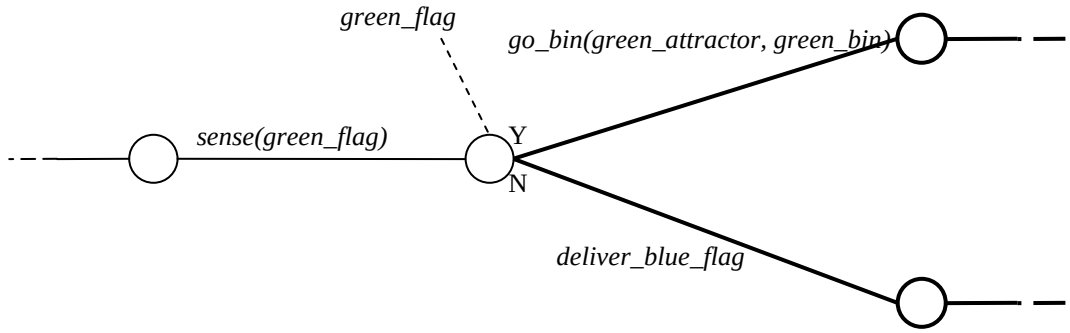

- ```
  proc(deliver_red_flag,
        (sense(red_flag):
         branch_on(red_flag):
         if(red_flag,
              go_bin(red_attractor,red_bin),
              deliver_green_flag))
  ).
  ```

  Questa procedura verifica se la bandierina raccolta è rossa ed in caso positivo invoca la procedura *go_bin* passandole come parametri *red_attractor* e *red_bin*, altrimenti chiama la procedura *deliver_red_flag*;



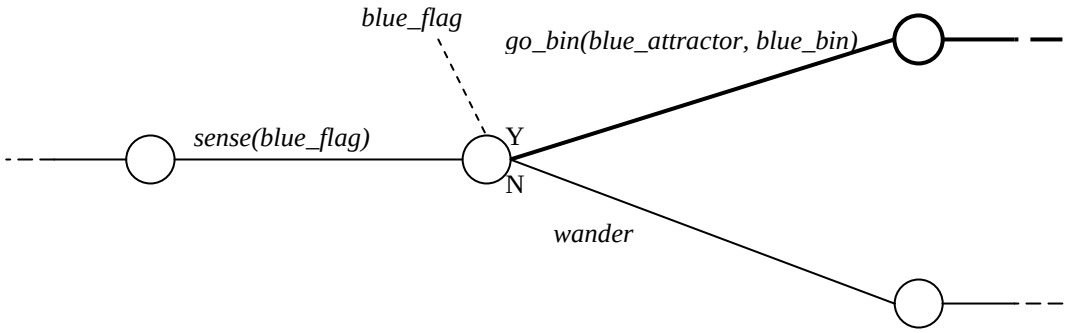
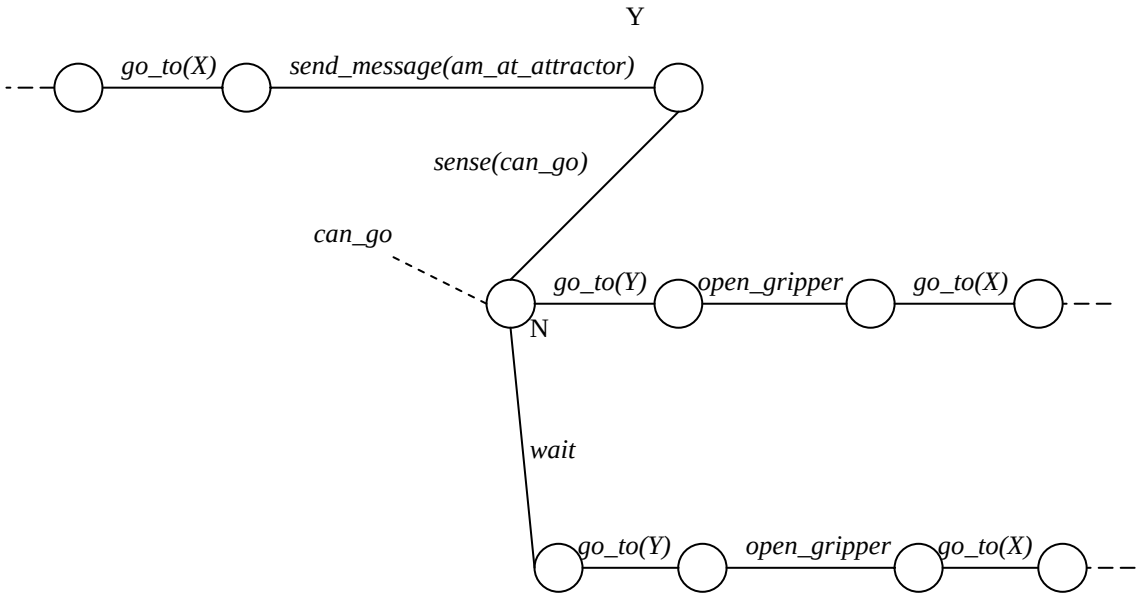

- ```
  proc(deliver_green_flag,
        (sense(green_flag):
         branch_on(green_flag):
         if(green_flag,
              go_bin(green_attractor,green_bin),
              deliver_blue_flag))
  ).
  ```

Questa procedura viene chiamata da *deliver_red_flag* nel caso in cui la bandierina raccolta non è rossa; essa verifica se la bandierina è di colore verde ed in caso positivo invoca la procedura *go_bin* passandole come parametri *green_attractor* e *green_bin*, altrimenti chiama la procedura *deliver_blue_flag*;




- ```
  proc(deliver_blue_flag,
        (sense(blue_flag):
         branch_on(blue_flag):
         if(blue_flag,
              go_bin(blue_attractor,blue_bin),
              wander))
  ).
  ```

  Questa procedura viene chiamata quando la bandierina raccolta non è rossa nè verde (quindi è sicuramente blu); essa verifica comunque il valore del fluente *blue_flag* ed in caso positivo invoca la procedura *go_bin* passandole come parametri *blue_attractor* e *blue_bin*, altrimenti chiama la procedura manda il robot alla ricerca di altre bandierine;


  ```

- ```
  proc(go_bin(X,Y),
        goto(X):send_message(am_at_attractor):
        (sense(can_go):
         branch_on(can_go):
         if(can_go,
              goto(Y):open_gripper:goto(X),
              wait:goto(Y):open_gripper:goto(X)))
  ).
  ```

  La procedura riceve due parametri: `X` che rappresenta l'attrattore posto davanti ad un contenitore, e `Y` che rappresenta il contenitore corrispondente. La procedura invia il robot sull'attrattore (*go_to(x)*) e gli dice di inviare un messaggio al compagno per comunicargli di aver raccolto la bandierina e di essersi posizionato davanti al contenitore (*send_message(am_at_attractor)*); verifica quindi se anche il compagno ha fatto lo stesso (*can_go* ha valore vero) ed in caso positivo si passa al deposito della bandierina : il robot si dirige sul contenitore (*go_to(Y)*), apre il gripper (*open_gripper*) e si dirige nuovamente sull'attrattore posto davanti al contenitore prima di ritornare nello stato di ricerca (questo viene fatto a causa della presenza di muri che delimitano le zone dei contenitori, contro cui il robot potrebbe bloccarsi); se invece il compagno non ha ancora raccolto una bandierina (*can_go* ha valore falso), il robot resta in attesa di ricevere il messaggio dal compagno (*wait*) e quindi passa al deposito della bandierina.


  ```

Il database che realizza il comportamento del nostro robot è il seguente:

```
primitive_action(goto(_)).
primitive_action(wander).
primitive_action(go_flag).
primitive_action(sense(_)).
primitive_action(open_gripper).
primitive_action(send_message(_)).
primitive_action(wait).

poss(goto(_),_).
poss(wander,_).
poss(turn_off_sonar,_).
poss(sense(_),_).
poss(open_gripper,_).
poss(go_flag,_).
poss(send_message(_),_).
poss(wait,_).

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(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).

proc(execute,
            (sense(something_visible):
            branch_on(something_visible):
            if(something_visible,
               go_flag,
               wander: go_flag):
            deliver_red_flag)
).
```

```
proc(deliver_red_flag,
            (sense(red_flag):
            branch_on(red_flag):
            if(red_flag,
                  go_bin(red_attractor,red_bin),
                  deliver_green_flag))
).

proc(deliver_green_flag,
            (sense(green_flag):
            branch_on(green_flag):
            if(green_flag,
                  go_bin(green_attractor, green_bin),
                  deliver_blue_flag))
).

proc(deliver_blue_flag,
            (sense(blue_flag):
            branch_on(blue_flag):
            if(blue_flag,
               go_bin(blue_attractor, blue_bin),
               wander))
).

proc(go_bin(X,Y),
      goto(X) : send_message(am_at_attractor) :
      (sense(can_go):
       branch_on(can_go):
       if(can_go,
            goto(Y) : open_gripper : goto(X),
            wait: goto(Y) : open_gripper : goto(X)))
).
```

4.2. Implementazione in TeamBots

Vediamo adesso cosa deve essere definito all'interno di TeamBots. Vengono create le classi necessarie per la pianificazione e vengono consultati i file che contengono l'interprete sGOLOG ed il database precedente:

```
...
    File eclipseProgram = new File(".\\Database\\sGolog.pl");
    client = new sGologClient(eclipseProgram);

    eclipseProgram = new File(".\\Database\\Competition.pl");
    client.consultFile(eclipseProgram);

    plan = new SGologPlan(abstract_robot);
...
```

Vengono creati i vari schemi percettivi, schemi motori, trigger e così via. I comportamenti Java del robot sono:

- `AS_GO_TO` : azione di movimento verso un punto con destinazione variabile (per fare questo si è utilizzata la classe *v_DinamicPoint_* presente all'interno del package *CSAI.unipa.clay*); quest'ultima viene determinata dal piano sGOLOG. Usiamo quindi questo comportamento per far dirigere il robot verso i contenitori e verso gli attrattori davanti ai contenitori;
- `AS_GO_FLAG` : azione di movimento verso la bandiera più vicina;
- `AS_GO_CLOSEST_ZONE`: azione di movimento verso la zona inesplorata della mappa più vicina al robot (per realizzare la mappa viene utilizzata la classe *NodeMap* contenuta all'interno del package *CSAI.unipa.knowledgment*).

Viene effettuato il *matching* tra le azioni primitive di sGOLOG e le azioni corrispondenti agli stati del robot:

```
...
    plan.addAction("goto",
                   AS_GOTO,
                   AS_GOTO,
                   PF_CLOSE_TO,
                   0,
                   new int[]{},
                   NodeAction.NO_LASER);
```

```
    plan.addAction("open_gripper",
                   NodeAction.NO_STEERING,
                   NodeAction.NO_TURRET,
                   NodeAction.IMMEDIATE_TRIGGER,
                   1,
                   NodeAction.NO_SONAR,
                   NodeAction.NO_LASER);

    plan.addAction("go_flag",
                   AS_GO_FLAG,
                   AS_GO_FLAG,
                   PF_TRIGGER_FLAG,
                   -1,
                   NodeAction.NO_SONAR,
                   NodeAction.NO_LASER);
     plan.addAction("wander",
                    AS_GO_CLOSEST_ZONE,
                    AS_GO_CLOSEST_ZONE,
                    PF_SOMETHING_VISIBLE,
                    NodeAction.NO_GRIPPER,
                    new int[]{0,2,4,6,8,10,12,14},
                    NodeAction.NO_LASER);

     plan.addAction("send_message",
                    NodeAction.NO_STEERING,
                    NodeAction.NO_TURRET,
                    NodeAction.IMMEDIATE_TRIGGER,
                    NodeAction.NO_GRIPPER,
                    NodeAction.NO_SONAR,
                    NodeAction.NO_LASER);

     plan.addAction("wait",
                    AS_GOTO,
                    AS_GOTO,
                    PF_OTHER_ROBOT,
                    NodeAction.NO_GRIPPER,
                    NodeAction.NO_SONAR,
                    NodeAction.NO_LASER);
...
```

In modo analogo, si realizza il *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);
...
```

E viene quindi generato il CAT:

```
...
   plan.setPlan(client.invoke("do(execute,s0,S)"),0);
...
```

L'FSA del robot è allora costituito da un unico stato, il CAT sGOLOG, che viene ripetuto ciclicamente. Bisogna inoltre assegnare i comportamenti a tutte e parti del robot:

```
...
   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;
...
   STEERING.embedded[0] = plan.getSteering(abstract_robot.getTime());
...
   TURRET.embedded[0] = plan.getTurret(abstract_robot.getTime());
...
   GRIPPER_FINGERS.embedded[0] =
       plan.getGripperFingers(abstract_robot.getTime());
...
   SONAR_CONFIGURATION.sonarActived[0] =
       plan.getSonar(abstract_robot.getTime());
...
    LASER_CONFIGURATION.embedded[0] = plan.getLaser(abstract_robot.getTime());
...
```

All'interno del metodo *takeStep()* viene aggiornato il piano e si controlla se si è passati da un'azione sGOLOG ad un'altra per settare i valori delle azioni con parametri:

```
...
    plan.update(curr_time, state);
    if(plan.isChangedState()){
       action = plan.getActionName();
...
```

In base al valore della stringa *action*, che rappresenta l'azione corrente, vengono gestiti :

- l'assegnazione dinamica dei punti del comportamento *AS_GO_TO* : in base al valore del parametro dell'azione *go_to(X)* possiamo settare le coordinate del punto di attrazione; ad esempio, se il valore del parametro è *"red_bin"* allora verranno impostate al punto di attrazione le coordinate del contenitore rosso (-9.0, -5.5), e analogamente per gli altri contenitori e per gli attrattori di fronte (N.B: sta all'utente definire il metodo *setValue* per assegnare i parametri a dei particolari comportamenti):

```
...
    if (action.startsWith("goto(")){
        if(plan.getParameter()[0].equals("red_bin")){
            PS_ATTRACTOR.setValue(curr_time, -9.0, -5.5);
        } else
        if(plan.getParameter()[0].equals("green_bin")){
            PS_ATTRACTOR.setValue(curr_time, -9.0, 0.0);
        } else
        if(plan.getParameter()[0].equals("blue_bin")){
            PS_ATTRACTOR.setValue(curr_time, -9.0, 6.0);
        } else
        if(plan.getParameter()[0].equals("red_attractor")){
            PS_ATTRACTOR.setValue(curr_time, -6.0, -5.5);
        } else
        if(plan.getParameter()[0].equals("green_attractor")){
            PS_ATTRACTOR.setValue(curr_time, -6.0, 0.0);
        } else
        if(plan.getParameter()[0].equals("blue_attractor")){
            PS_ATTRACTOR.setValue(curr_time, -6.0, 6.0);
        }
    }
...
```

- l'invio dei messaggi: se il valore dell'azione corrente è *send_message(am_at_attractor)* vuol dire che il robot ha raccolto una bandierina e si è posizionato di fronte al contenitore in stato di attesa; il nostro robot allora deve comunicare di aver raccolto una bandierina al 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){}
     }
...
```

- il controllo del trigger *PF_OTHER_ROBOT*: *PF_OTHER_ROBOT* è associato al fluente *can_go*, quindi ogni volta che entrambi i robot escono dal deposito delle bandierine devono settarne a falso il valore; *PF_OTHER_ROBOT* deve essere settato a vero ogni volta che viene ricevuto il messaggio "Pronto":

```
...
    if(action.startsWith("wander")) PF_OTHER_ROBOT.setValue(false)
...
    if(message.val == "Pronto")
                    PF_OTHER_ROBOT.setValue(true);
...
```

5. Codice delle classi

Di seguito viene riportato il codice delle classi implementate.

5.1. Classe SgologClient

```
package CSAI.unipa.SGolog;
import com.parctechnologies.eclipse.*;
import java.io.*;
import java.awt.event.*;
import java.awt.*;
import java.util.StringTokenizer;
import javax.swing.text.*;

/**
 *
 * @author Marco Di Stefano & Carmelo Scozzola
 */
public class sGologClient {

    private EclipseEngineOptions        eclipseEngineOptions;
    private OutOfProcessEclipse         eclipse;
    private List                        plan = new List();
    private File                        file;
    private static final String         fileSuffix = "pl";

    /** Creates a new instance of GologEditor */
    public sGologClient(File dataBase) {
        //---Embedding golog
        eclipseEngineOptions = new EclipseEngineOptions();
        eclipseEngineOptions.setUseQueues(false);
        try {
            eclipse = new OutOfProcessEclipse(eclipseEngineOptions);
        }
        catch(Exception e) {
        }
        // Compile the eclipse program.
        this.consultFile(dataBase);
    }

    /** compile a file **/
    public void consultFile(File f) {
        try {
            eclipse.compile(f);
        }
        catch(IOException ioe) {
            System.out.println("File not found");
        }
        catch(EclipseException ee) {
            System.out.println("Eclipse exception");
        }
```

```
}

/** add a fact using the assert command **/
public void add(String fact) {
    try{
        eclipse.rpc("assert("+fact+")");
    }
    catch(EclipseException e)
    {
        System.out.println("EclipseException: ");
    }
    catch(IOException ioe)
    {
        System.out.println("IOException: ");
    }
}

/** delete a fact using retract command **/
public void delete(String fact) {
    try{
        eclipse.rpc("retract("+fact+")");
    }
    catch(EclipseException e)
    {
        System.out.println("EclipseException: ");
    }
    catch(IOException ioe)
    {
        System.out.println("IOException: ");
    }
}

/** invoke the result of a query **/
public List invoke(String query) {
    try {
        plan = new List();
        CompoundTerm result =eclipse.rpc(query);

        //The third argument of the query is the goal
        Object goal = result.arg(3);
        String string = goal.toString();
        int k=0;
        StringTokenizer tokenizer = new StringTokenizer(string,"[.=] ",true);
        while (tokenizer.hasMoreTokens()) {
            String tmp = tokenizer.nextToken();
            if(tmp.startsWith("com")) {
                for(int i=0;i<5;i++)
                    tokenizer.nextElement();
            }
            /** if function has parameter add the key word #param#
             * followed by the number of parameters **/
            else if(tmp.equals("arity")){
```

```
                    tmp = tokenizer.nextToken();
                    tmp = tokenizer.nextToken();
                    plan.add("#param#");
                    plan.add(tmp);
                }
                /** these are all terms to be filtered **/
                else if(
                    tmp.startsWith("Compound")||
                    tmp.equals("functor")||
                    tmp.equals("with")||
                    tmp.startsWith("arg")||
                    tmp.startsWith("Atom")||
                    tmp.equals(".")||
                    tmp.equals("=")||
                    tmp.equals(" ")
                ) continue;
                /** add all the other terms to the plan **/
                else if(tmp.equals("[")){
                    plan.add(tmp);
                }
                else if(tmp.equals("]")){
                    plan.add(tmp);
                }
                else{
                    plan.add(tmp);
                }
            }
        }catch(Exception e){System.out.println("Exception "+e);}
        return plan;
    }
}
```

5.2. NodeAction

```
package CSAI.unipa.SGolog;
import EDU.gatech.cc.is.clay.*;
import CSAI.unipa.clay.*;

/**
 *
 * @author Marco Di Stefano & Carmelo Scozzola
 */
public class NodeAction extends Node {

    public static final boolean DEBUG = Node.DEBUG;

    private String                      name;
    private NodeVec2                    steering;
    private NodeVec2                    turret;
    private NodeBoolean                 trigger;
    private int                         gripper;
    private int[]                       sonar;
    private int                         laser;

    public static final NodeVec2        NO_STEERING = null;
    public static final NodeVec2        NO_TURRET = null;
    public static final NodeBooleanDinamic     IMMEDIATE_TRIGGER = new NodeBooleanDinamic(true);
    public static final int             NO_GRIPPER = 1000;
    public static final int[]           NO_SONAR = new int[]{1000};
    public static final int             NO_LASER = -1;
    public static final int             LASER_ON = 1;
```

```
public static final int             LASER_OFF = 0;


/** Creates a new instance of a_NodeAction */
public NodeAction(String n, NodeVec2 steer, NodeVec2 turr, NodeBoolean tr, int grip, int[]
                  son_conf, int las) {
    name = n;
    steering = steer;
    turret = turr;
    trigger = tr;
    gripper = grip;
    sonar = son_conf;
    laser = las;
}

/** Creates a new instance of a_NodeAction */
public NodeAction(NodeAction act) {
    name = act.stringValue();
    steering = act.steeringValue();
    turret = act.turretValue();
    trigger = act.triggerValue();
    gripper =  act.gripperValue();
    sonar = act.sonarValue();
    laser = act.laserValue();
}

/** Creates a new instance of a_NodeAction */
public NodeAction() {
    name = "";
    steering = NodeAction.NO_STEERING;
    turret = NodeAction.NO_TURRET;
    trigger = NodeAction.IMMEDIATE_TRIGGER;
    gripper = NodeAction.NO_GRIPPER;
    sonar = NodeAction.NO_SONAR;
    laser = NodeAction.NO_LASER;
}

/**
 * Get the NodeVec2 value.
 *
 * @return the NodeVec2 steering
 */
public NodeVec2 steeringValue() {
    return steering;
}

/**
 * Get the NodeVec2 value.
 *
 * @return the NodeVec2 turret
 */
public NodeVec2 turretValue() {
    return turret;
}

/**
 * Get the String value.
 *
 * @return the String value
 */
public String stringValue() {
    return name;
}

/**
 * Get the NodeBoolean value.
 *
 * @return the NodeBoolean trigger
 */
public NodeBoolean triggerValue() {
    return trigger;
}

/**
 * Get the int[] value.
 *
 * @return the int[] sonar
 */
```

```
    public int[] sonarValue() {
        return sonar;
    }

    /**
     * Get the int value.
     *
     * @return the int laser
     */
    public int laserValue() {
        return laser;
    }

    /**
     * Get the int value.
     *
     * @return the int gripper
     */
    public int gripperValue() {
        return gripper;
    }
}
```

5.3. ActionList

```
package CSAI.unipa.SGolog;
import EDU.gatech.cc.is.clay.*;
import CSAI.unipa.clay.*;
import java.util.*;

/**
 *
 * @author Marco Di Stefano & Carmelo Scozzola
 */
public class ActionList{


    public static final boolean DEBUG =     Node.DEBUG;
    private Vector                          embedded;

    public ActionList() {
        embedded = new Vector();
    }

    /**
     * Get the NodeVec2 value.
     *
     * @param timestamp long indicates time of the request
     * @param String int indicates the name of the action
     * @return the NodeVec2 action
     */
    public NodeVec2 steeringValue(String item) {
        try{
            NodeAction act = (NodeAction)embedded.elementAt(indexOfString(item));
            return act.steeringValue();
        }
        catch(ArrayIndexOutOfBoundsException ie){
            System.out.println("Error "+ie);
            return(new v_Noise_(0));
        }
        catch(NullPointerException ne){
            System.out.println("Error "+ne);
            return(new v_Noise_(0));
        }
    }

    /**
     * Get the NodeVec2 value.
     *
     * @param timestamp long indicates time of the request
     * @param item int indicates the position of the action in the array
     * @return the NodeVec2 action
     */
    public NodeVec2 steeringValue(int item) {
        try{
            NodeAction act = (NodeAction)embedded.elementAt(item);
```

```
            return act.steeringValue();
        }
        catch(ArrayIndexOutOfBoundsException ie){
            System.out.println("Error "+ie);
            return(new v_Noise_(0));
        }
        catch(NullPointerException ne){
            System.out.println("Error "+ne);
            return(new v_Noise_(0));
        }
    }

    /**
     * Get the NodeBoolean value.
     *
     * @param timestamp long indicates time of the request
     * @param item int indicates the position of the trigger in the array
     * @return the NodeBoolean trigger
     */
    public NodeBoolean triggerValue(int item){
        try{
            NodeAction act = (NodeAction)embedded.elementAt(item);
            return act.triggerValue();
        }
        catch(ArrayIndexOutOfBoundsException ie){
            System.out.println("Error b "+ie);
            return(new NodeBooleanDinamic(false));
        }
        catch(NullPointerException ne){
            System.out.println("Error b "+ne);
            return(new NodeBooleanDinamic(false));
        }
    }

    /**
     * Get the NodeBoolean value.
     *
     * @param timestamp long indicates time of the request
     * @param String int indicates the name of the trigger
     * @return the trigger value
     */
    public NodeBoolean triggerValue(String item){
        try{
            NodeAction act = (NodeAction)embedded.elementAt(indexOfString(item));
            return act.triggerValue();
        }
        catch(ArrayIndexOutOfBoundsException ie){
            System.out.println("Error b "+ie);
            return(new NodeBooleanDinamic(false));
        }
        catch(NullPointerException ne){
            System.out.println("Error b "+ne);
            return(new NodeBooleanDinamic(false));
        }
    }

    /**
     * Get the position of the action named item.
     *
     * @param timestamp long indicates time of the request
     * @param item int indicates the name of the action
     * @return the int position of the action named item
     */
    public int indexOfString(String item) {
        for(int i=0; i<embedded.size();i++){
            NodeAction act = (NodeAction)embedded.elementAt(i);
            if(act.stringValue().equals(item)){
                return(i);
            }
        }
        return(-1);
    }

    /**
     * Get the String value.
     *
     * @param timestamp long indicates time of the request
     * @param item int indicates the position in the array
```

```
 * @return the String value
 */
public String stringValue(int item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(item);
        return act.stringValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return("");
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return("");
    }
}

/**
 * Get the NodeBoolean value.
 *
 * @param timestamp long indicates time of the request
 * @param String int indicates the name of the trigger
 * @return the trigger value
 */
public NodeVec2 turretValue(String item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(indexOfString(item));
        return act.turretValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(new v_Noise_(0));
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(new v_Noise_(0));
    }
}

/**
 * Get the NodeBoolean value.
 *
 * @param timestamp long indicates time of the request
 * @param item int indicates the position of the trigger in the array
 * @return the NodeBoolean trigger
 */
public NodeVec2 turretValue(int item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(item);
        return act.turretValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(new v_Noise_(0));
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(new v_Noise_(0));
    }
}

/**
 * Get the dimension of the array
 *
 *@return the ine dimension
 */
public int getLength() {
    return embedded.size();
}

/**
 * Get the action of position item in the array
 *
 * @param item int the position of the action
 * @return the NodeAction
 */
public NodeAction getNodeAction(int item) {
    try{
```

```
        NodeAction act = (NodeAction)embedded.elementAt(item);
        return act;
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(new NodeAction());
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(new NodeAction());
    }
}

/**
 * Get the int value.
 *
 * @param timestamp long indicates time of the request
 * @param String int indicates the name of the trigger
 * @return the int gripper value
 */
public int laserValue(String item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(indexOfString(item));
        return act.laserValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(-1);
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(-1);
    }
}

/**
 * Get the int value.
 *
 * @param timestamp long indicates time of the request
 * @param String int indicates the name of the gripper configuration
 * @return the int gripper value
 */
public int laserValue(int item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(item);
        return act.laserValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(-1);
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(-1);
    }
}

/**
 * Get the int value.
 *
 * @param timestamp long indicates time of the request
 * @param String int indicates the position of the gripper configuration
 * @return the int gripper value
 */
public int gripperValue(int item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(item);
        return act.gripperValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(0);
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(0);
    }
}
```

```
/**
 * Get the int value.
 *
 * @param timestamp long indicates time of the request
 * @param String int indicates the name of the gripper configuration
 * @return the int gripper value
 */
public int gripperValue(String item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(indexOfString(item));
        return act.gripperValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(0);
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(0);
    }
}

/**
 * Get the int[] value.
 *
 * @param timestamp long indicates time of the request
 * @param item String indicates the name of the gripper configuration
 * @return the int gripper value
 */
public int[] sonarValue(String item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(indexOfString(item));
        return act.sonarValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(new int[]{});
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(new int[]{});
    }
}

/**
 * Get the int[] value.
 *
 * @param timestamp long indicates time of the request
 * @param item int indicates the position of the sonar configuration
 * @return the int gripper value
 */
public int[] sonarValue(int item) {
    try{
        NodeAction act = (NodeAction)embedded.elementAt(item);
        return act.sonarValue();
    }
    catch(ArrayIndexOutOfBoundsException ie){
        System.out.println("Error "+ie);
        return(new int[]{});
    }
    catch(NullPointerException ne){
        System.out.println("Error "+ne);
        return(new int[]{});
    }
}

/**
 * Adds the specified item to the end of scrolling list.
 * @param item the item to be added
 * @since JDK1.1
 */
public void add(NodeAction item) {
    add(item,-1);
}

/**
 * Adds the specified item to the the scrolling list
```

```
 * at the position indicated by the index.  The index is
 * zero-based.  If the value of the index is less than zero,
 * or if the value of the index is greater than or equal to
 * the number of embedded in the list, then the item is added
 * to the end of the list.
 * @param       item   the item to be added;
 *              if this parameter is <code>null</code> then the item is
 *              treated as an empty string, <code>""</code>
 * @param       index  the position at which to add the item
 * @since       JDK1.1
 */
public void add(NodeAction item, int index) {
    if (index < -1 || index >= embedded.size()) {
        index = -1;
    }

    if (item == null) {
        item = new NodeAction();
    }

    if (index == -1) {
        embedded.addElement(item);
    } else {
        embedded.insertElementAt(item, index);
    }

}

/**
 * @deprecated     replaced by <code>remove(String)</code>
 *                         and <code>remove(int)</code>.
 */
private void delItem(int position) {
    delembedded(position, position);
}

/**
 * @deprecated As of JDK version 1.1,
 * Not for public use in the future.
 * This method is expected to be retained only as a package
 * private method.
 */
private synchronized void delembedded(int start, int end) {
    for (int i = end; i >= start; i--) {
        embedded.removeElementAt(i);
    }
}

/**
 * Gets the item associated with the specified index.
 * @return       an item that is associated with
 *                   the specified index
 * @param        index the position of the item
 * @see          #getItemCount
 */
public NodeAction getItem(int index) {
    return getItemImpl(index);
}

/**
 * Gets the number of embedded in the list.
 * @return     the number of embedded in the list
 * @see        #getItem
 * @since      JDK1.1
 */
public int getItemCount() {
    return embedded.size();
}

/**
 * Gets the embedded in the list.
 * @return       a string array containing embedded of the list
 * @see          #select
 * @see          #deselect
 * @see          #isIndexSelected
 * @since        JDK1.1
 */
public synchronized String[] getembedded() {
```

```
    String itemCopies[] = new String[embedded.size()];
    embedded.copyInto(itemCopies);
    return itemCopies;
}

/**
 * Removes the first occurrence of an item from the list.
 * @param        item  the item to remove from the list
 * @exception    IllegalArgumentException
 *                   if the item doesn't exist in the list
 * @since        JDK1.1
 */
public synchronized void remove(NodeAction item) {
    int index = embedded.indexOf(item);
    if (index < 0) {
        throw new IllegalArgumentException("item " + item +
        " not found in list");
    } else {
        remove(index);
    }
}

/**
 * Remove the item at the specified position
 * from this scrolling list.
 * @param      position   the index of the item to delete
 * @see        #add(String, int)
 * @since      JDK1.1
 * @exception    ArrayIndexOutOfBoundsException
 *               if the <code>position</code> is less than 0 or
 *               greater than <code>getItemCount()-1</code>
 */
public void remove(int position) {
    delItem(position);
}

/**
 * Removes all embedded from this list.
 * @see #remove
 * @see #delembedded
 * @since JDK1.1
 */
public void removeAll() {
    clear();
}

/**
 * Replaces the item at the specified index in the scrolling list
 * with the new string.
 * @param       newValue   a new string to replace an existing item
 * @param       index      the position of the item to replace
 * @exception ArrayIndexOutOfBoundsException if <code>index</code>
 *         is out of range
 */
public synchronized void replaceItem(NodeAction newValue, int index) {
    remove(index);
    add(newValue, index);
}

final NodeAction getItemImpl(int index) {
    return (NodeAction)embedded.elementAt(index);
}

/**
 * @deprecated As of JDK version 1.1,
 * replaced by <code>removeAll()</code>.
 */
private synchronized void clear() {
    embedded = new Vector();
}

public boolean isInList(String item){
    int index = indexOfString(item);
    if(index==-1){
        //System.out.println("azione "+item+" non in lista");
        return(false);
    }
    else{
```

```
            //System.out.println("azione "+item+" in lista");
            return(true);
        }
    }
}
```

5.4. SGologPlan

```
package CSAI.unipa.SGolog;
import EDU.gatech.cc.is.abstractrobot.*;
import EDU.gatech.cc.is.util.*;
import EDU.gatech.cc.is.clay.*;

import com.parctechnologies.eclipse.*;
import java.io.*;
import java.awt.event.*;
import java.awt.*;
import java.util.StringTokenizer;
import javax.swing.text.*;

import CSAI.unipa.clay.*;
import CSAI.unipa.abstractrobot.*;
import CSAI.unipa.SGolog.*;
import EDU.gatech.cc.is.abstractrobot.*;


/**
 *
 * @author Marco Di Stefano & Carmelo Scozzola
 */
public class SGologPlan {

    /** the result of the invocation **/
    private List                        goal;
    /** the plan numerically formatted **/
    private List                        numberGoal;
    /** a copy of the plan used to reset **/
    private List                        bufferGoal;
    /** the name of the current action **/
    private String                      action;
    /** the parameters of the current action **/
    private String[]                    param;

    /** the robot state **/
    private NodeVec2                    steering;
    private NodeVec2                    turret;
    private NodeBoolean                 trigger;
    private int                         gripper_fingers;
    private int[]                       sonar;
    private int                         laser;

    /** the matched actions **/
    private ActionList                  actions;
    /** an instance of the robot **/
    private SimpleInterface             abstract_robot;
    /** the trigger to control the golog state **/
    private NodeBooleanDinamic          finished;
    /** the number of golog state in FSA **/
    private int                         gologState;
    /** the trigger sensed **/
    private List                        sensedTrigger;
    /** if true debugging is actived **/
    private boolean                     debug = false;
    private boolean                     isChangedState = false;


    /** Creates a new instance of SGologPlane */
    public SGologPlan(SimpleInterface ar) {
        finished = new NodeBooleanDinamic(false);
        actions = new ActionList();
        param = new String[0];
        abstract_robot = ar;
        trigger = NodeAction.IMMEDIATE_TRIGGER;
        sensedTrigger = new List();
        sonar = new int[0];
    }
```

```
long            lasttime = 0;
private int     count = 0;

/** each timestep makes golog state go on **/
public int update(long timestamp, int state){

    isChangedState = false;
    /** begins the sgolog state **/
    if(count == 0)
        finished.setValue(false);

    /** ends the sgolog state, reset and exit **/
    if(count==(goal.getItemCount()-1)){
        reset();
        return(0);
    }

    /** if sgolog state is not finished, and need to check the next action **/
    if ((!finished.Value(timestamp))&&
        (state==gologState)&&
        (trigger.Value(timestamp))&&
        ((timestamp > lasttime)||(timestamp == -1)))
    {
        /*--- reset the timestamp ---*/
        if (timestamp > 0) lasttime = timestamp;

        /** repeat until an action il performed **/
        do{
            action = goal.getItem(count++);
            /** do the sense at the trigger **/
            if(action.equals("sense")){
                count = count + 3;
                action = goal.getItem(count++);
                performSense(timestamp, action);
                action = goal.getItem(count);
            }
            else
            /** there is a sensing and a branch to cut **/
            if(isSensedTrigger(action)){
                cutBranch(timestamp, count++);
                action = goal.getItem(count);
            }
            else
            /** is an action to perform **/
            if(actions.isInList(action)){
                readParameter();
                performAction(timestamp, action);
                isChangedState = true;
            }
            /** plan is finished, reset and exit **/
            if(count==(goal.getItemCount()-1)){
                reset();
                return(0);
            }
        }
        while((action.equals("["))||(action.equals("]"))||(action.equals(",")));
    }
    /** configure the robot if is in the golog state **/
    if(state==gologState)
        setRobotConfiguration(timestamp);
    return(0);
}


public void setPlan(List plan, int state) {
    sensedTrigger = new List();
    param = new String[0];
    count = 0;
    /** the goal **/
    goal = new List();
    trigger = NodeAction.IMMEDIATE_TRIGGER;
    for(int i=0;i<plan.getItemCount();i++)
        goal.add(plan.getItem(i));
    /** the copy of goal **/
    bufferGoal = new List();
    for(int i=0;i<goal.getItemCount();i++)
        bufferGoal.add(goal.getItem(i));
```

```
    gologState = state;
    /** creates the numerically formatted goal **/
    formatQuery(goal);
    if(debug){
        System.out.println("\n");
        for(int i=0;i<goal.getItemCount();i++)
            System.out.print(goal.getItem(i));
        System.out.println("\n");
    }
}

/** return true if plan is finished **/
public NodeBoolean isFinished(long timestamp) {
    return finished;
}

/** return the nodevec2 of steering **/
public NodeVec2 getSteering(long timestamp) {
    if(steering == null)
        return(new v_GlobalPosition_r(abstract_robot));
    else
        return steering;
}

/** return the nodevec2 of turret **/
public NodeVec2 getTurret(long timestamp) {
    if(turret == null)
        return(new v_GlobalPosition_r(abstract_robot));
    else
        return turret;
}

/** return the int of gripper **/
public int getGripperFingers(long timestamp) {
    return gripper_fingers;
}

/** return the int[] of sonar **/
public int[] getSonar(long timestamp){
    return sonar;
}

/** return the boolean of laser **/
public boolean getLaser(long timestamp){
    if(laser==1)
        return true;
    else
        return false;
}

/** get the parameters of a function **/
private void readParameter(){
    param = new String[0];
    List tmp = new List();
    /** if #param# there are parameters **/
    if(goal.getItem(count).equals("#param#")){
        count++;
        /** read the number of parameters **/
        int param_number = Integer.parseInt(goal.getItem(count++));
        if(debug)
            System.out.println("Funzione con parametri: "+param_number);
        int num = 0;
        /** repeat until read all the parameters **/
        do{
            String temp = goal.getItem(count++);
            /** if is a parameter add it to tmp, and increase num **/
            if((!temp.equals("["))&&(!temp.equals("]"))){
                tmp.add(temp);
                num++;
            }
        }while(num<param_number);
    }
    /** copy tmp into param **/
    param = new String[tmp.getItemCount()];
    for(int i=0;i<tmp.getItemCount();i++){
        param[i] = tmp.getItem(i);
    }
}
```

```
/** perform an action **/
private void performAction(long timestamp, String action) {
    /** create the string to be displayed by the robot **/
    String p = "(";
    for(int i=0;i<param.length;i++){
        if(i!=0){
            p = p+", ";
        }
        p = p+param[i];
    }
    p = p+")";
    if(debug)
        System.out.println("Eseguo l'azione :"+action+p+"\n");
    /** assign the state of the robot **/
    abstract_robot.setDisplayString("SGolog: "+action+p);
    steering = actions.steeringValue(action);
    turret = actions.turretValue(action);
    sonar = actions.sonarValue(action);
    gripper_fingers = actions.gripperValue(action);
    laser = actions.laserValue(action);
    trigger = actions.triggerValue(action);
}

/** make the sense of a trigger **/
private void performSense(long timestamp, String action) {
    try{
        sensedTrigger.add(action);
        if(debug)
            System.out.println("Effettuo il sense sul trigger :"+action+"\n");
        trigger = NodeAction.IMMEDIATE_TRIGGER;
    }catch(ArrayIndexOutOfBoundsException ie){}
}

/** set the configuration of the robot dependign by the performing action **/
private void setRobotConfiguration(long timestamp) {
    // STEERING
    if(steering!=null){
        Vec2 result = steering.Value(timestamp);
        abstract_robot.setSteerHeading(timestamp, result.t);
        abstract_robot.setSpeed(timestamp, result.r);
    }
    // TURRET
    if(turret!=null){
        Vec2 result = turret.Value(timestamp);
        try{
            SimpleN150ExploreSim simple =(SimpleN150ExploreSim)abstract_robot;
            simple.setTurretHeading(timestamp, result.t);
        }
        catch(Exception e){};
        try{
            SimpleN150Sim simple =(SimpleN150Sim)abstract_robot;
            simple.setTurretHeading(timestamp, result.t);
        }
        catch(Exception e){};
    }
    //SONAR
    if(sonar!=NodeAction.NO_SONAR){
        try{
            SonarObjectSensor sensor =(SonarObjectSensor)abstract_robot;
            sensor.turnOffAllSonar();
            for(int i=0;i<sonar.length;i++)
                sensor.turnOnSonar(sonar[i]);
        }
        catch(Exception e){};
    }
    //LASER
    if(laser!=NodeAction.NO_LASER){
        try{
            LaserFinderObjectSensor sensor =(LaserFinderObjectSensor)abstract_robot;
            if(laser == 1)
                sensor.turnOnLaser();
            else
                sensor.turnOffLaser();
        }
        catch(Exception e){};
    }
    //GRIPPER FINGERS
```

```
    if(gripper_fingers!=NodeAction.NO_GRIPPER){
        try{
            SimpleN150ExploreSim simple =(SimpleN150ExploreSim)abstract_robot;
            simple.setGripperFingers(timestamp, gripper_fingers);
        }
        catch(Exception e){};
        try{
            SimpleN150Sim simple =(SimpleN150Sim)abstract_robot;
            simple.setGripperFingers(timestamp, gripper_fingers);
        }
        catch(Exception e){};
    }
}

/** control if is a sensed trigger **/
private boolean isSensedTrigger(String action) {
    try{
        for(int i=0;i<sensedTrigger.getItemCount();i++)
            if(sensedTrigger.getItem(i).equals(action))
                return(true);
    }catch(ArrayIndexOutOfBoundsException ie){}
    return(false);
}

private void cutBranch(long timestamp, int count){
    /** control the trigger **/
    param=new String[0];
    boolean bresult = actions.triggerValue(action).Value(timestamp);
    trigger = NodeAction.IMMEDIATE_TRIGGER;
    if(debug)
        System.out.println("Il trigger "+goal.getItem(count-1)+" ha valore "+bresult);

    int i = count;
    /** if is trigger false **/
    if(!bresult){
        if(debug)
            System.out.println("\nEseguo un branch cut:");
        String level = numberGoal.getItem(i);
        String act = null;
        do{
            act = numberGoal.getItem(i+1);
            numberGoal.remove(i+1);
            goal.remove(i+1);
        }
        while(!act.equals(level));
    }
    /** if is trigger true **/
    else{
        if(debug)
            System.out.println("\nEseguo un branch cut:");
        String level = numberGoal.getItem(i);
        String act = null;
        int cont = i+1;
        do{
            act = numberGoal.getItem(cont++);
        }
        while(!act.equals(level));
        do{
            act = numberGoal.getItem(cont);
            numberGoal.remove(cont);
            goal.remove(cont);
        }
        while(!act.equals(level));
    }
    if(debug){
        for(int j=0;j<goal.getItemCount();j++)
            System.out.print(goal.getItem(j));
        System.out.println("\n");
    }
}

/** create the numerically formatted plan **/
private void formatQuery(List query) {
    int cont = 0;
    numberGoal = new List();
    for(int i=0;i<query.getItemCount();i++){
        numberGoal.add(query.getItem(i));
        if(query.getItem(i).equals("[")){
```

```
                cont++;
                numberGoal.replaceItem("@"+cont,i);
            }
            else
            if(query.getItem(i).equals("]")){
                cont--;
                numberGoal.replaceItem("@"+cont,i);
            }
        }
    }

    /** reset the plan **/
    private void reset(){
        if(debug)
            System.out.println("resetto");
        sensedTrigger = new List();
        param = new String[0];
        count = 0;
        goal = new List();
        for(int i=0;i<bufferGoal.getItemCount();i++)
            goal.add(bufferGoal.getItem(i));
        formatQuery(goal);
        if(debug){
            System.out.println("\n");
            for(int i=0;i<goal.getItemCount();i++)
                System.out.print(goal.getItem(i));
            System.out.println("\n");
        }
        finished.setValue(true);
    }

    public void addAction(String n, NodeVec2 steer, NodeVec2 turr, NodeBoolean tr, int grip, int[]
                            son_conf, int las){
        actions.add(new NodeAction(n,steer,turr,tr,grip,son_conf,las));
    }

    public void addAction(String n, NodeBoolean tr){
        actions.add(new NodeAction(n,NodeAction.NO_STEERING,NodeAction.NO_TURRET,tr,
                                   NodeAction.NO_GRIPPER,NodeAction.NO_SONAR,NodeAction.NO_LASER));
    }

    public void addAction(String n, NodeVec2 steer, NodeVec2 turr, NodeBoolean tr, int grip){
        actions.add(new NodeAction(n,steer,turr,tr,grip,NodeAction.NO_SONAR, NodeAction.NO_LASER));
    }

    public void debugging(boolean value){
        debug = value;
    }

    public String[] getParameter() {
        return param;
    }

    public String getActionName(){
        String p = action+"(";
        for(int i=0;i<param.length;i++){
            if(i!=0){
                p = p+", ";
            }
            p = p+param[i];
        }
        p = p+")";
        return p;
    }

    public boolean isChangedState(){
        return isChangedState;
    }
}
```