Università degli studi di Brescia - 1998-99

Controllo di un robot mobile tramite la tecnologia ad agenti cooperanti

DelBono Emanuele - 22288

Draghin Daniele - 22952

Lorini Valerio - 22766

Indice

Introduzione

Obbiettivo di questo lavoro è la realizzazione di un sistema software per il controllo di robot mobili attraverso la tecnologia ad agenti cooperanti.

Il termine agente e' usato in vari contesti con differenti significati. Tuttavia, l'orientamento della ricerca moderna ha focalizzato l'attenzione su agenti concepiti come entita' che sono i soggetti di intenzioni e persuasioni. In altre parole, attitudini come intenzioni, persuasioni e desideri sono associate all'agente al fine di ottenere un oggetto astratto in grado di fornire un modo conveniente e familiare per descrivere, spiegare e predire il comportamento di sistemi complessi.

Il generico agente deve risolvere i problemi di sua competenza e comunicare con gli altri agenti. Per questo motivo la sua struttura rispecchia questa sua duplice funzione. Al suo interno si sono introdotti due moduli separati e interagenti che svolgono le due funzioni specificate:

1. Kernel: si occupa della risoluzione vera e propria dei problemi di sua competenza specifica.

2. Shell: si occupa della comunicazione con gli altri agenti e della raccolta di dati statistici sull'agente con finalità di controllo.

La separazione dell'agente in due moduli interagenti, consente di attribuire, ad ognuno di essi, lo status di modulo autonomo e dotato di risorse proprie, autonome, indipendenti. In questo modo, l'intera architettura risulta costituita da un insieme di moduli interagenti (due moduli per agente). Il vantaggio offerto da questa scelta è notevole. Le due parti che costituiscono un agente non sono indipendenti da un punto di vista funzionale, quindi, presi singolarmente, non possono essere considerati degli agenti. Tuttavia attraverso la cooperazione essi sono in grado di realizzare l'agente, poiché possiedono capacità complementari. I moduli, però, sono indipendenti sotto il profilo delle risorse a disposizione (capacità di elaborazione e conoscenza), perciò il loro funzionamento può avvenire in modo indipendente e concorrenziale, poiché le attività che svolgono non sottostanno a vincoli di sequenzialità. Ad esempio, mentre il kernel sta risolvendo un problema, la shell può svolgere la propria attività di interfaccia e di controllo. Tutto ciò non sarebbe possibile se l'agente fosse costituito da un unico modulo, dato che, qualora una funzione dell'agente, ad esempio quella di elaborazione, avesse occupato le risorse, le altre funzioni dovrebbero attenderne la liberazione per eseguire il proprio compito. In questo modo si introdurrebbe, all'interno dell'agente, un vincolo di sequenzialità che non trova riscontro nel funzionamento reale. Affinché le azioni intraprese da un agente siano coerenti con lo stato dell'ambiente esterno e con le reali esigenze degli altri agenti dell'architettura, è necessario che, al manifestarsi di una richiesta di collaborazione di qualsiasi tipo, l'agente sia in grado di fornire aiuto nel più breve tempo possibile. Con il passare del tempo, infatti, lo stato dell'architettura evolve, ed analogamente evolve lo stato degli agenti che la compongono. Quando la risposta ad una richiesta di collaborazione arriva con molto ritardo, la situazione che ha prodotto la richiesta di aiuto può essere cambiata, ad esempio perché l'agente è riuscito a risolvere il problema in altro modo o tramite risultati provenienti da altri agenti, e la risposta può diventare inutile. Queste situazioni devono essere evitate, per quanto possibile, perché generano interazioni inefficaci e dannose a tutto il processo di problem-solving e

diminuiscono le prestazioni e l'efficienza del sistema. La progettazione dell'agente tramite moduli separati consente di ridurre le interazioni inutili aumentando la coerenza di comportamento dell'agente, inoltre consente di aumentare l'efficienza e le prestazioni di tutto il processo di risoluzione.
All'interno di un agente i due moduli possono funzionare in modo indipendente per quanto riguarda l'uso delle risorse di elaborazione, ma interagiscono in modo complesso per quanto riguarda lo scambio di informazioni. Durante la risoluzione di un problema, il kernel e la shell interagiscono scambiandosi informazioni e messaggi attraverso canali privati inaccessibili dall'esterno. Un osservatore esterno non può percepire in alcun modo la struttura interna dell'agente. L'immagine di un agente, visto dall'esterno, è caratterizzata esclusivamente dalla presenza del meccanismo per l'invio e la ricezione dei messaggi.

Il nostro lavoro è stato quello di interfacciare il programma AGENTS+ con il simulatore Saphira al fine di ottenere la simulazione di un robot che si muove autonomamente in un mondo, quale una serie di uffici collegati tramite corridoi.
Il programma è stato compilato su macchine SUN ULTRA 10 che utilizzano il sistema operativo Solaris 2.6 . La parte inerente gli agenti è stata scritta in C++ e compilata con il compilatore SUN SPARCOMPILER 4.2, per quanto riguarda invece la parte che si interfaccia direttamente con il simulatore Saphira, scritta in linguaggio C, abbiamo utlizzato il gcc.
Il programma fa uso della Coroutine Library [Stroustrup & Shopiro 87] che permette di implementare il multithread derivando le classi dei singoli thread dalla classe task.

Implementazione della struttura multi agente

Gli agenti che vengono creati non appena viene lanciato il programma sono 3 :

- Movement Manager : addetto alla movimentazione del robot;
- Integrity Preservation : garantisce l'integrità del robot;
- Sensor Manager : controlla i sensori;

Ad ogni agente è associata una conoscenza memorizzata in vari file di testo aventi un particolare formato che vedemo in seguito. Ogni agente è composto dai seguenti componenti:

- l'interfaccia
- i componenti operativi
- le intenzioni
- le persuasioni

Ognuno di questi componenti deve affrontare due problemi:

- completare il task che gli è stato assegnato in accordo con la sua natura
- comunicare con gli altri componenti dell'agente

Questi due compiti, come detto prima, sono eseguiti da due moduli separati : il **kernel** e la **shell**. La shell e il kernel sono implementati nel programma come due classi derivate dalla classe *task*. In altri termini il costruttore di un componente è strutturato come segue:

Component operation

Begin

Create the component shell;

Create the componet kernel;

Require component registration to the respective agent interface;

end task;

End

La shell gioca il ruolo di intermediario tra i componenti esterni e il kernel processando messaggi secondo lo schema:

Shell operation

Begin

Wait for a message

If message is coming from an external component

Then

Forwards the message to the kernel;

Else

Sends the message to the external component specified within the message

End

Per quanto concerne il kernel invece le principali attività sono:

1. ricevere richieste di risoluzione di problemi dagli altri agenti
2. ricevere richieste di risoluzione di problemi dai componenti del proprio agente

La conoscenza di ogni agente è memorizzata in file di testo ed è suddivisa in 5 categorie che vedremo ora nel dettaglio:

- Self knowledge
- Mutual knowledge
- Planning knowledge
- Problem solving knowledge
- Domain knowledge

Questi tipi di conoscenza sono memorizzate in file e sono codificate attraverso precise regole di sintassi, allo scopo di poter essere correttamente comprese e utilizzate. I file di testo devono avere un nome particolare che consenta all'agente di poterlo trovare facilmente.

Self knowledge

La self knowledge è una conoscenza che contiene la descrizione dei problemi che rientrano nelle capacità risolutive dell'agente. Questa conoscenza è divisa in :

- Statica: ciò che concerne i componenti operativi
- Dinamica: ciò che concerne le intenzioni

Static self knowledge

Contiene le informazioni per l'attivazione dei vari componenti operativi.

Il nome del file è costruito nel seguente modo:

nome_file = nome_agente + «SS»

e il contenuto del file ha il seguente formato:

```
Nome_Azione :: Nome_Funzione
```

dove

- `Nome_Azione` è l'azione che deve essere eseguita
- `Nome_Funzione` è il nome del componente operativo che esegue l'azione.

Dynamic self knowledge

Per semplicità è suddivisa in due file, uno che contiene le intenzioni primitive e il secondo che contiene le intenzioni generate durante le operazioni dell'agente.

Per quanto riguarda le intenzioni primitive il nome del file è cotruito come segue:

nome_file = nome_agente + «DSP»

e il suo contenuto:

```
Nome_Azione::Subject_Inention(Lock_Type)
```

dove:

- `Nome_Azione` è il nome dell'azione da eseguire
- `Subject_Intention` è il soggetto dell'intenzione che deve essere creato per eseguire la corrispondente azione.

- `Lock_Type` è un parametro usato per distribuire le risorse condivise.

Il nome del file che contiene le intenzioni generate durante le operazioni dell'agente è costruito nel seguente modo:

nome_file = nome_agente + «DS»

e il suo contenuto:

```
Nome_Azione::Subject_Intention(Lock_Type,Deadline)
```

dove

- `Nome_Azione`, `Subject_Intention` e `Lock_Type` hanno lo stesso significato di prima
- `Deadline` è il tempo massimo entro il quale la richiesta deve essere soddisfatta.

Mutual knowledge

La mutual knowledge è la conoscenza che l'agente ha del mondo esterno, cioè informazioni sulle entità presenti nell'architettura con le quali l'agente a cui si riferisce la conoscenza ha interagito. Nel nostro caso la struttura della mutual knowledge è abbastanza semplice: essa è contenuta in un file di testo che raccoglie le informazioni rilevate dalle interazioni con le altre entità. Ogni riga contiene informazioni circa una diversa entità. Le informazioni che possono essere raccolte riguardo ad un'entità sono al più quelle presenti nella sua self knowledge, poiché, dato che non esistono funzionalità di modeling che consentono all'entità di ricavare alcune caratteristiche del suo interlocutore attraverso l'analisi dei risultati delle interazioni, l'entità usa le informazioni che il suo interlocutore gli fornisce su sé stesso

nome_file = nome_agente + «MK»

Il formato del file :

```
Nome_Azione:: Nome_Agente_Esterno
```

dove

- `Nome_Azione` è il nome dell'azione
- `Nome_Agente_Esterno` è il nome dell'agente che è in grado di svolgere quell'azione.

Planning knowledge

In Agent+ si è assunto che le intenzioni non generino runtime dei piani ma che dei piani precompilati siano memorizzati all'interno di una base di conoscenza.

In particolare la planning knowledge contiene le associazioni tra i soggetti dell'intenzione e il piano per raggiungerla.

Nome_file = nome_agente + «PL»

Si fa differenza tra le intenzioni primitive e quelle generate, per le prime il formato del file è il seguente:

```
intention subject_intention (priority)

   plan_1

   plan_2

   …

   plan_n

end
```

dove

- `subject_intention` rappresenta il soggetto dell'intenzione
- `priority` è un etichetta che può assumere i valori very_low, low, medium, high, very_high.
- `Plan_i` è il piano necessario per raggiungere l'intenzione.

Per le intenzioni generate il formato ed il file è il seguente:

```
intention subject_intention (validity_condition)

   plan_1

   plan_2

   …

   plan_n

end
```

dove `validity_condition` è una proposizione che rappresenta la condizione per l'attivazione e la permanenza dell'intenzione.

Problem-solving knowledge

La problem solving knowledge contiene le associazioni tra i piani definiti nella planning knowledge e le azioni per portali a compimento. Il formato nome del file di testo è costruito come segue:

Nome_file = nome_agente + «PS»

Il formato del file è:

```
plan nome_piano(applicability_condition)

   action_1[T1]
```

```
    action_2[T2]
    …
    action_m[Tm]
end
```

dove

- `action_i` sono le azioni necessarie per completare il piano
- `T1,T2,…,Tm` sono i tempi stimati per eseguire l'azione
- `applicability_condition` è una condizione che indica l'applicabilità del piano.

Domain knowledge

E' la conoscenza che l'agente ha del dominio in cui opera.

E' composta da 3 parti :

- context nome_file = nome_agente + «CK»
- default nome_file = nome_agente + «DK»
- relation nome_file = nome_agente + «RK»

Le prime due (CK e DK) sono sequenze di fatti che devono essere veri nel contesto operativo considerato. Per ora sono condivisi tra tutti gli agenti.

La terza (RK) è memorizzata in un file con il seguente formato:

```
proposition justification_content(justification_type |
justification_type)
```

dove

- `proposition` è il soggetto della persuasione da giustificare
- `justification_content` è la proposizione che rappresenta il contenuto della giustificazione
- `justification_type` è il tipo della giustificazione, può assumere i valori related preposition, agent_knowledge, sensorial_data.

Struttura del programma AGENTS+

Il sistema di multi-agenti è creato nel main.cc nel seguente modo:

main program

begin
    Create agent1;
    Create agent2;
    …
    Create agentN;
end

più in dettaglio:(main.cc)

```
// file main.cc

#include <task.h>
#include <string.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>
#include "communic.h"
#include "agent.h"
#include "coda.h"
#include "sstring.h"
#include "global.h"
#include "responder.h"
#include "mirror.h"

#include "sfun.h"      //Prototipi per l'interfacciamento a Saphira

int num_agents = 3;    //Numero agenti utilizzati

int stop = 0;
int contai;
long global_code = 0;
long global_time = 0;     // tempo di rilevamento dati sensoriali
int obstacle_avoided = 0;
int wheels_blocked = 0;
int wheels_free = 0;

int wheels_locked = 0;
int failure = 0;
int Giro = 0;
int pointXY_is_reached=0; // modifica per la funzione go_XY

int count_agents = 0;
agent* db_agents = new agent[num_agents];
coda codaResp = new qhead;   // variabile globale nota a tutti...
coda codaMirror = new qhead;
```

```
void MyMoveRobot()
{
 int j;
 mysfSetVelocity(Saffy.Vel);    // SETTA LA VELOCITA'
 current_position.PX = mysfGetX();
 current_position.PY = mysfGetY();
}

void myAgents(void)
{
new Responder("Responder");
new Mirror("Mirror");

//-------- CREA GLI AGENTI DI BASE ------------
 Agent SM("Sensors_manager");
 Agent MM("Movement_manager");
 Agent IP("Integrity_preservation");

 thistask->resultis(0);
 exit(0);
}

void main(int argc, char **argv)
{
 // --- LANCIA SAPHIRA -----
 myStartSaphira();  // Inizializza l'ambiente di Saphira
 myAgents();        // Lancia il programma Agent+
}
```

La classe Agent

La classe agente ha la seguente struttura:

```
class Agent
{
private:
   String name;
public:
   Agent(const String name);
   ~Agent();
}
```

L'interfaccia, i componenti operativi e le intenzioni primitive vengono create nel costruttore dell'agente in questo modo:

Agent constructor

Begin
    Create the interface;

for each element belonging to the Static Self Knowledge of the agent
Create an operative component

for each element belonging to the dynamic Permanent Self Knowledge of the agent
Create a Primitive Intention
End

L'interfaccia

L'interfaccia è il primo componente creato ed è quello che mantiene tutte le informazioni globali sull'agente (indirizzo, componenti attivi,…). Per questo ogni attività mentale o opertiva manda un messaggio di registrazione all'interfaccia dopo che è stata creata.

```
class Interface : public task
{
private:
   Inter_Queues Qu;
public:
   Interface(const String name,queue AgentQ);
   ~Interface()
}
```

`AgentQ` denota l'indirizzo dell'agente e `name` è il nome dell'agente.

Componenti operativi

Sono creati recuperando le informazioni dalla static self knowledge

```
Class Op_Comp : public task
{
private:
   Op_Queues op_Qu;
public:
   Op_Comp(const String name_comp, const String name_action,…,queue
AgentQ);
   ~Op_Comp();
}
```

dove

- `name_comp` è il nome del componente
- `name_action` è il nome dell'azione che può eseguire
- `AgentQ` è l'indirizzo dell'interfaccia dell'agente Intenzioni primitive

Intenzioni

Sono create estraendo le informazioni contenute nella dynamic permanent self knowledge dell'agente.

```
Class intention : public task
{
private :
```

```
   Inten_Queues inten_Qu;
public:
   Intention(const String subject, const String name_action,…,queue
AgentQ);
   ~Intention();
}
```

dove
- `subject` è il soggetto dell'intenzione
- `name_action` è il nome dell'azione che l'intenzione può eseguire
- `AgentQ` è l'indirizzo dell'interfaccia

Struttura dei componenti e operazioni

Ognuno dei componenti visti deve affrontare due problemi:

1. Eseguire il compito per il quale è stato creato

2. Comunicare con gli altri componenti dell'agente.

Queste due operazioni sono eseguite ripettivamente dal Kernel e dalla Shell.
La shell ha il ruolo di intermediario tra i componenti esterni e il kernel.
La comunicazione tra i vari componenti avviene tramite lo scambio di messaggi.

Struttura dei messaggi

Il meccanismo di comunicazione riveste un ruolo fondamentale all'interno dell'architettura, poiché permette l'interazione tra le entità che vi sono contenute. Questo è importante perché la risoluzione dei problemi è basata sulla cooperazione tra entità, attraverso la comunicazione. Il meccanismo prescelto per Agents+ è di semplice utilizzo e si basa su due concetti fondamentali:

- la coda dei messaggi,

- il messaggio.

La comunicazione tra entità avviene attraverso un corretto utilizzo di questi due oggetti. Di seguito si descriveranno le caratteristiche principali di questi oggetti.

La coda dei messaggi

Le entità dell'architettura comunicano fra loro attraverso lo scambio di messaggi. Ciascuna possiede un indirizzo tramite il quale è raggiungibile dalle altre. Per spedire un messaggio ad una entità è sufficiente conoscerne l'indirizzo. In effetti, ognuna possiede una coda FIFO nella quale vengono accumulati, in ordine di arrivo, i messaggi che giungono, in attesa di essere elaborati. L'indirizzo di una entità non è altro che l'indirizzo della sua coda dei messaggi. Le principali operazioni sulla coda dei messaggi sono:

- prendere un messaggio dalla coda;
- spostare un messaggio all'inizio della coda;

- fissare il numero massimo di messaggi che possono essere contenuti nella coda;
- richiedere il numero massimo di messaggi che possono essere contenuti nella coda;
- richiedere il numero di messaggi che si trovano in coda;
- richiedere il numero di messaggi che possono essere ulteriormente contenuti dalla coda.

La coda dei messaggi, come è stato anticipato, è di tipo FIFO. Questo significa che i messaggi vengono inseriti nella coda, e quindi diventano disponibili all'entità, nell'ordine di arrivo. In alcuni casi può essere necessario disporre di tipi di code diverse, oppure esaminare i messaggi in coda senza rimuoverli. Queste funzionalità sono facilmente realizzabili utilizzando una funzione che consente di spostare un messaggio all'inizio della coda, oppure utilizzando altre code ausiliarie per immagazzinare temporaneamente i messaggi.
L'unica operazione che una entità può eseguire sulla coda di un'altra entità (a patto di conoscerne l'indirizzo) è:

- inserire un messaggio in quella coda.

Questa è l'operazione che realizza la comunicazione, poiché consente ad una entità di inserire, nella coda di un'altra entità, un messaggio.
Ogni entità possiede le proprie code di messaggi necessarie per la comunicazione, tuttavia, se necessario, essa è in grado di crearne delle nuove per far fronte a particolari esigenze di elaborazione. Ad esempio, si supponga che un'entità debba attendere un messaggio particolare. Dato che le code FIFO permettono all'entità di ottenere i messaggi esclusivamente in ordine di arrivo, diventa necessario alterare questo ordine.
Tutte le funzionalità per la gestione delle code è messa a disposizione dalla Task Library, di seguito vedremo quali entità sono coinvolte e come devono essere utilizzate.

I messaggi

I messaggi sono gli oggetti che le entità del sistema possono scambiarsi attraverso il meccanismo di comunicazione. Questi oggetti sono composti da un insieme di campi che possono contenere informazioni di tipo prestabilito. La struttura del messaggio è fissa, cioè il tipo dei campi deve essere predefinito e non può essere mutato. Nonostante sia possibile introdurre diversi tipi di messaggio, uno per ogni tipo di interazione, nel prototipo AGENTS+ si è preferito utilizzare un unico tipo di messaggio per tutti i tipi di interazione. Questa semplificazione ha consentito di ridurre la conoscenza di cui le entità devono disporre per la costruzione e l'interpretazione dei messaggi e di semplificare il meccanismo di comunicazione. Tuttavia, questa scelta ha portato a dover introdurre nel messaggio alcuni campi che servono solo per alcuni tipi di messaggi. Inoltre, il messaggio è dotato di campi di tipo dinamico, cioè di dimensioni non fissate. La struttura dei messaggi definita per il prototipo AGENTS+ è costituita dai seguenti campi, il cui significato verrà brevemente presentato di seguito:

- `id` Identificatore del messaggio
- `type` Tipo del messaggio
- `sender_type` Tipo del messaggio di ritorno
- `sender_name` Nome del mittente
- `content` Descrizione del problema da risolvere
- `wait_name` Elenco delle informazioni da attendere
- `result_address` Indirizzo dell'entità a cui spedire il risultato
- `result_string` Esito della risoluzione

Identificatore del messaggio. È un codice formato dal nome del mittente e da un numero progressivo, che identifica in modo univoco il messaggio.

Tipo del messaggio. È un nome che identifica il contenuto del messaggio. L'agente è composto da due moduli indipendenti e interagenti. Ciascuno di questi moduli è in grado di spedire e ricevere messaggi. La shell, che riceve tutti i messaggi diretti all'agente, deve essere in grado di individuare a quale modulo deve essere girato il messaggio. Si sono individuati tre tipi di messaggio che consentono alla shell di individuare il modulo a cui sono diretti. Questi tipi di messaggio sono: QUESTION, ANSWERDEC e ANSWERSOLV.

- QUESTION: contengono un problema da risolvere o una richiesta di informazioni. La shell inserisce questi messaggi nella coda dell'agente interessato.

- ANSWERDEC: sono i messaggi provenienti da aree di competenza o da agenti specialisti, contenenti informazioni riguardanti l'allocazione di sottoproblemi.

- ANSWERSOLV: contiene informazioni dirette al kernel. Si tratta di messaggi che contengono soluzioni di sottoproblemi necessari al nucleo di risoluzione per svolgere la sua attività risolutiva.

Tipo del messaggio di ritorno. Un modulo che spedisce un messaggio di tipo QUESTION ad un'altra entità, può indicare, tramite questo campo del messaggio, il tipo del messaggio che dovrà contenere la risposta. Questo consente ad un modulo di richiedere una informazione e contemporaneamente specificare a quale modulo debba essere spedita la risposta.

Nome del mittente. Il nome del mittente consente, a chi riceve il messaggio, di sapere il nome dell'entità da cui proviene. Questa informazione non è scontata poiché, ai fini dell'attività di comunicazione, essa non è necessaria e può essere omessa. Per comunicare con un'altra entità, infatti, è sufficiente conoscere l'indirizzo della sua coda dei messaggi. Tuttavia, questo indirizzo non può essere usato come riferimento assoluto all'entità poiché esso viene riassegnato ogni volta che l'architettura viene inizializzata. Il nome dell'entità, invece, rimane sempre costante poiché è una informazione presente nella sua conoscenza, e che non è modificabile, se non tramite un intervento dall'esterno. Il nome dell'entità, in questo prototipo, consente l'aggiornamento della mutual knowledge.

Descrizione del problema da risolvere. Questo campo contiene, espressa attraverso il linguaggio specifico del dominio di competenza dell'architettura, la descrizione del problema da risolvere. Le entità dell'architettura, in questo prototipo, non possiedono conoscenze per effettuare i ragionamenti necessari ad interpretare la descrizione del problema da risolvere per scoprire se esso ricade tra le loro competenze. Ad un livello minimale, si richiede, quindi, che ad ogni problema che può essere risolto dalle entità dell'architettura corrisponda un'unica descrizione (che a questo punto si riduce al nome del problema).

Elenco delle informazioni da attendere. È possibile specificare, all'interno del messaggio, le informazioni che l'entità che lo riceve deve attendere, prima di iniziare la risoluzione del problema descritto nel messaggio. Questo meccanismo dà la possibilità, all'entità che spedisce il messaggio, di specificare il contesto entro il quale il problema va risolto.

Indirizzo dell'entità a cui spedire il risultato. Questo campo specifica a quale entità dovrà essere spedita la risposta al messaggio stesso. L'entità che ha ricevuto il messaggio, perciò, costruirà e spedirà un messaggio di risposta all'indirizzo contenuto in questo campo.

Esito della soluzione. Questo campo contiene l'esito della risoluzione del problema. In generale, l'esito è "positivo", se il problema è stato risolto, e "negativo" se il problema non è stato risolto, tuttavia è possibile aggiungere altre informazioni domain dependent, ad esempio, la descrizione della causa del fallimento del tentativo di risoluzione.

La descrizione proposta per la struttura del messaggio fa riferimento ad un uso di tipo generale dei messaggi stessi.
Le operazioni che possono essere effettuate da una entità su un messaggio sono le seguenti:

- creare un messaggio;
- acquisire il contenuto dei campi del messaggio;
- modificare il contenuto dei campi del messaggio;
- duplicare un messaggio;
- stampare il contenuto del messaggio.

Il meccanismo di comunicazione

Il meccanismo di comunicazione fornito dalla libreria Task non possiede un oggetto che realizza una coda di messaggi, ma fornisce due classi, tra loro collegate, la classe `qhead` e la classe `qtail`, che insieme realizzano la vera coda di messaggi. La gestione di una coda con queste caratteristiche, che richiede l'utilizzo di due puntatori, uno per inserire messaggi nella coda (nella qtail) e l'altro per estrarre messaggi dalla coda (dalla qhead), può risultare eccessivamente macchinoso e, oltre a rendere meno leggibile il codice prodotto, può portare il programmatore, a commettere errori. Allo scopo di rendere più semplice le funzionalità di comunicazione AGENTS+ utilizza una sola classe, la classe qhead, definendo, per essa, il nuovo nome coda, e si sono nascosti i meccanismi di conversione dei puntatori, le operazioni di cast dei messaggi e le chiamate alle funzioni proprie delle classi qhead e qtail, all'interno di due funzioni che consentono di eseguire, in modo semplice e chiaro, le operazioni necessarie per la comunicazione:

```
Message* getFrom( coda )
void putIn( Message* , coda )
```

Dopo queste modifiche, per creare una coda chamata Agent_queue si procede nel seguente modo:

```
coda Agent_queue = new coda;
```

Per estrarre un messaggio da questa coda si utilizza la funzione getFrom:

```
Message* mess = getFrom( Agent_queue );
```

Mentre per mettere un messaggio, chiamato mess, in questa coda si utilizza la funzione putIn:

```
putIn( mess, Agent_queue );
```

In questo modo, il meccanismo di comunicazione risulta più semplice da utilizzare.

Al fine di comprendere meglio come vengono utilizzati i messaggi riportiamo una funzione del programma in cui si utilizzano messaggi.

```
Message* change_dir_right(Op_Queues& Q, Message* m)
{
 Message* mess = new Message();
 extern long global_code;
 extern int Giro;
 extern int energy;
 int i;

 mess->id = ++global_code;
 mess->type = ANSWER;
 mess->sender_name = Q.name_comp;
 mess->sender_type = OPERATIVE;
 mess->sender_address = Q.codaComp;
 mess->content = m->content;
 mess->num_param = m->num_param;
 if (m->num_param > 0)
 {
  mess->param = new String_A[mess->num_param];
  for (i=0;i<mess->num_param;i++)
      mess->param[i] = m->param[i];
 }

 mess->result_string = "Gira";
 mess->wait_name = m->wait_name;
 mess->wait_address = m->wait_address;

 mysfSetRVelocity(-TURN_SPEED); //Gira a destra

 energy--;
 Giro = 1;
 return mess;
}
```

Come si può intuire questa funzione comanda al robot di ruotare verso destra. Da notare che la funzione riceve in ingresso il messaggio `m` e al suo interno crea un nuovo messaggio `mess` che ritorna alla funzione che l'ha chiamata. In questo messaggio copia alcuni parametri dal messaggio `m` mentre altri li aggiorna nel modo opportuno.

L'interfacciamento con Saphira

L'interfacciamento con Saphira è implementato tramite un client in linguaggio C (file sfun.c e sfun.h); questo a differenza dei sorgenti di Agents+ devono essere compilati con il gcc tramite il comando

```
make –f make_sfun
```

il file oggetto ottenuto viene poi linkato insieme al resto del programma al fine di ottenere il file eseguibile agents.

In questi file è stata implementata una semplice interfaccia alle API di Saphira prestando particolare attenzione alla sincronizzazione del programma principale con il server di Saphira. A tal fine abbiamo dovuto introdurre degli stati di attesa tramite la funzione

```
MySfPause(PAUSA);
```

Queste pause si sono rese necessarie quando ci siamo accorti che al server arrivavano svariate richieste che non riusciva a soddisfare in tempo causando il crash del programma.

Riportiamo di seguito i due file dedicati all'interfacciamento opportunamente commentati:

```
/* sfun.c */
#include "saphira.h"

void myStartupFn(void)
{
   sfSetDisplayState(sfDISPLAY,1);
   sfConnectToRobot(sfLOCALPORT,sfCOMLOCAL);
}

void myStartSaphira(void)
{
   sfOnStartupFn(myStartupFn);
   printf("Starting...\n");
   sfStartup(1);
   while (! sfIsConnected) sfPause(0); /* aspetta la connessione */
}

void mysfSetVelocity(int param)
{
   sfSetRVelocity(0);       /* VELOCITA' DI ROTAZIONE NULLA*/
   sfPause(PAUSA);
   sfSetVelocity(param);  /* AVANTI TUTTA! */
   sfPause(PAUSA);
}

void mysfSetRVelocity(int param)
{
   if (param) sfSetVelocity(0); /* MOVIMENTO SOLO ROTAZIONALE */
   sfSetRVelocity(param);
   sfPause(PAUSA);
}


float mysfGetX(void)
{
   sfPause(PAUSA);
   return sfRobot.ax;
}

float mysfGetY(void)
{
   sfPause(PAUSA);
   return sfRobot.ay;
```

```
}

float mysfGetAth(void)
{
   sfPause(PAUSA);
   return sfRobot.ath;
}

int* mysfSonarRange(int* s)
{
   int j;
   if (s ==NULL) s=malloc(7*sizeof(int));  /* RIEMPIE L'ARRAY CON I
VALORI*/
   for (j=0;j<7;j++)              /* DI LETTURA DEI SONAR */
      s[j]=sfSonarRange(j);
   sfPause(PAUSA);
   return s;
}

void mysfPause(long t)
{                    /* PAUSA NECESSARIA PER LA TEMPORIZZAZIONE */
   sfPause(t);
}


//////////////////////////////////////////////////////////////////

// sfun.h

/* VELOCITA' TRASLAZIONALE */
#ifndef FORWARD_SPEED
#define FORWARD_SPEED 300
#endif


/* A QUESTA DISTANZA INIZIA A RALLENTARE */
#ifndef COLLISION_LIMIT
#define COLLISION_LIMIT 2500
#endif

/* VELOCITA' TRASLAZIONALE IN VICINANZA DI UN OSTACOLO */
#ifndef FORWARD_SLOW
#define FORWARD_SLOW 10
#endif


/* VELOCITA' ROTAZIONALE */
#ifndef TURN_SPEED
#define TURN_SPEED 45
#endif

#define PAUSA 200   // Millisecondi

/* SOGLIA DI COLLISIONE */
#ifndef COLLISION_TH
#define COLLISION_TH 1000
#endif
```

```
// prototipi per lanciare saphira

static int *sonar=NULL;

typedef struct {
   int Vel;
   }sSaffy;

sSaffy Saffy;

#ifndef __SFUN_H

#define __SFUN_H

#ifdef __cplusplus
extern "C"
{
#endif

   void myStartSaphira(void);
   void myAgents(void);
   void mysfSetVelocity(int);
   void mysfSetRVelocity(int);
   float mysfGetX(void);
   float mysfGetY(void);
   float mysfGetAth(void);
   int* mysfSonarRange(int *);
   void mysfPause(long);

#ifdef __cplusplus
}
#endif

#endif
```

Da segnalare le due funzioni addette alla movimentazione: `mysfSetVelocity` e `mysfSetRVelocity`. Queste prima di impostare la nuova velocità di traslazione e rotazione rispettivamente azzerano la velocità di rotazione e traslazione (rispettivamente), questo fa si che il robot possa o andare dritto o girare ma non entrambe le cose contemporaneamente. Abbiamo utilizzato questo meccanismo ereditandolo da Khepera che veniva movimentato controllando i due motori (destro e sinistro) in modo indipendente.

Sviluppo del progetto e problematiche incontrate

Il punto di partenza del progetto è stato quello di trovare una nuova versione della libreria, e relativa documentazione, della classe task per la programmazione multithread. Il nostro obbiettivo era quello di trovarne una versione per il sistema operativo Linux in modo da poter portare tutto il prototipo già esistente sulla piattaforma PC. Tale libreria si è poi rivelata inesistente e per questo abbiamo ripiegato su sistemi SUN e utilizzato il compilatore incluso nel pacchetto SparcWorks 4.2.

Successivamente abbiamo individuato il meccanismo col quale Agent+ e il simulatore Khepera interagiscono, abbiamo studiato il metodo di comunicazione tra i vari agenti, le funzioni di visualizzazione, i file della conoscenza, etc. Isolate le chiamate a khepera, necessarie per la rappresentazione grafica del robot all'interno di un mondo virtuale, abbiamo dovuto modificare i sorgenti relativi alla movimentazione e alla percezione sensoriale, adattandoli al simulatore Saphira che non disponeva di tutti i sonar del robot Khepera, abbiamo dovuto quindi rimappare e modificare i piani per adattarli alla struttura del Pioneer. Una notevole quantità di tempo e energie è stata spesa nell'ottimizzare il codice per rendere più stabile il programma. In particolare la sincronizzazione tra Agent+ e saphira, ci ha costretti a introdurre degli stati di attesa dopo l'esecuzione di alcune funzioni del simulatore. Uno dei problemi è stato, ad esempio, l'errata lettura dei sonar, che ovviamente pregiudicava irrimediabilmente il comportamento del robot. Succesivamente abbiamo corretto alcuni errori nella gestione della classe `String`, nella deallocazione dei puntatori alla classe task della precedente versione di agent+. Inizialmente il lavoro e' stato effettuato su una SparcStation con sistema operativo Solaris 2.3. Su questa piattaforma non esisteva però una versione di Saphira. Quindi siamo passati ad un sistema più recente, ossia una SUN ULTRA 10 con sistema operativo Solaris 2.6. Dopo aver riadattato nuovamente il codice, siamo riusciti a creare l'eseguibile. E' iniziata poi la lunga fase di debugging, ottimizzazione e testing del comportamento del robot. In questa fase abbiamo incontrato svariati problemi dovuti ai tempi di risposta del server, capitava spesso che il robot riceveva il comando di evitare l'ostacolo troppo tardi e inevitabilemte urtava quest'ultimo. Abbiamo allora lavorato sulle soglie di collisione e sulla velocità di traslazione del robot, aumentando le prime e riducendo la seconda con risultati non del tutto soddisfacenti. Per migliorare le prestazioni abbiamo introdotto una funzione di rallentamento lineare nel caso di rilevamento di un ostacolo, diversificando il comportamento nel caso in cui l'ostacolo si presenti lateralmente o frontalmente. Grazie a questa funzionalità il robot può raggiungere velocità più elevate quando non vengono rilevati ostacoli e rallentare quando vi si avvicina.

Conclusioni e sviluppi futuri

A questo punto il robot riesce a muoversi autonomamente evitando gli ostacoli che gli si presentano. Grazie all'architettura ad agenti che lo rende "robusto" non è vincolato a particolari ambienti (non avendo memorizzato al suo interno una mappa del mondo). Può quindi agire in qualsiasi tipologia di corridoi e uffici. Non ha ancora uno scopo e sarebbe interessante darglielo: gli si potrebbe dire di raggiungere un certo punto al fine di ottenere un robot postino del tutto autonomo.

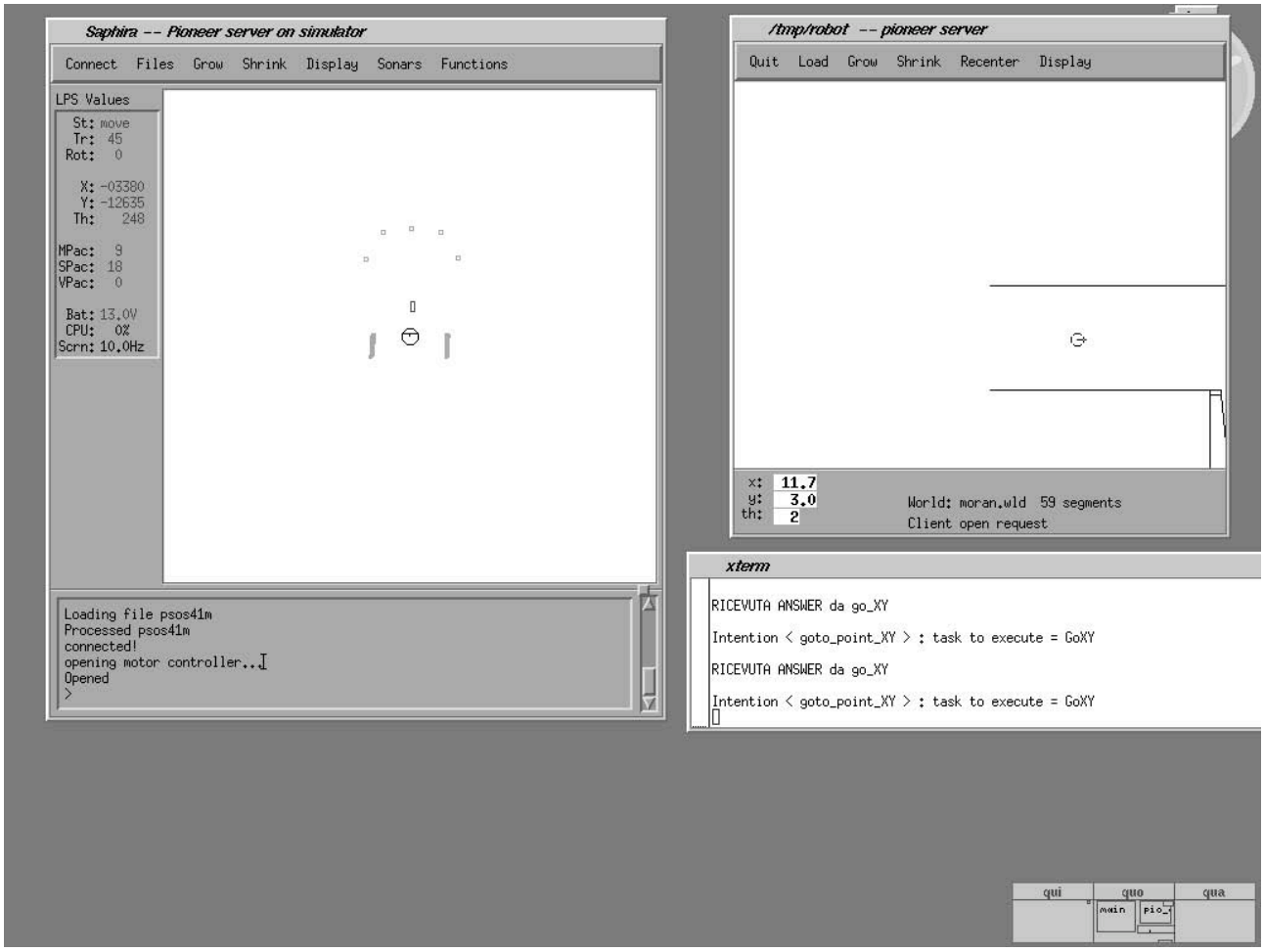

Fig. 1: Esecuzione di AGENTS+

Appendice : esempio.

Di seguito riportiamo i messaggi a schermo, visualizzati durante una sessione di AGENTS+. Si possono individuare le fasi di inizializzazione, di movimentazione in assenza di ostacoli, individuazione ostacoli e conseguente modifica dei piani. (nota: le righe che iniziano per # sono commenti introdotti nella relazione per facilitare la comprensione)

```
#inizializzazione di Saphira
Starting...
Display visual class: 3
Occ grid color unavailable
Allocated color cells for occ grid
Screen depth: 8

#creazione dei componenti operativi
sono il componente operativo check_obs
sono il componente operativo check_obs_right
sono il componente operativo check_obs_left
sono il componente operativo check_obs_upright
sono il componente operativo check_obs_upleft
sono il componente operativo check_light
sono il componente operativo go_forward
sono il componente operativo change_dir_left
sono il componente operativo change_dir_right
sono il componente operativo check_turn
sono il componente operativo turn_towards_light
sono il componente operativo check_if_RP_reached
sono il componente operativo check_if_light_reached
sono il componente operativo turn_towards_RP
sono il componente operativo hold_position
sono il componente operativo turn_back
sono il componente operativo check_if_reached_XY
sono il componente operativo go_XY
sono il componente operativo check_if_pointXY_reachable
sono il componente operativo lock_wheels
sono il componente operativo eliminate_lock_on_wheels
sono il componente operativo check_if_wheels_free
sono il componente operativo check_if_obstacle_avoided
sono il componente operativo check_energy
sono il componente operativo check_if_recharged


#compito da eseguire : se non ci sono ostacoli va avanti dritto
Intention < goto_point_XY > : task to execute = GoXY

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Persuasion < obs >: the truth value "true" has been determined

Persuasion < obs >: notification sent to avoid_collision

RICEVUTA ANSWER da go_XY


Intention < goto_point_XY > : task to execute = GoXY

Intention < avoid_collision > : task to execute = LeaveObstacle

Persuasion < obs >: the truth value "false" has been determined

Persuasion < obs >: notification sent to avoid_collision
```

```
Intention < avoid_collision > : plan revision needed

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

#Individuato ostacolo, viene selezionato il piano per evitarlo.

Intention < avoid_collision > : the new plan is
plan_avoid_collision1

Intention < leave_obstacle > created with deadline 0

Persuasion < leave_obstacleDone > created by leave_obstacle

Persuasion < obs_left > created by leave_obstacle

Persuasion < obs_right > created by leave_obstacle

Persuasion < obs_up_right > created by leave_obstacle

Persuasion < obs_up_left > created by leave_obstacle

Intention < leave_obstacle > : the chosen plan is
plan_leave_obstacle3

Intention < leave_obstacle > : the new plan is
plan_leave_obstacle4

Intention < leave_obstacle > : the new plan is
plan_leave_obstacle3

Intention < avoid_collision > : the chosen plan is empty

Intention < avoid_collision > : the new plan is
plan_avoid_collision1

Persuasion < leave_obstacleDone > eliminated

Persuasion < obs_left > eliminated

Persuasion < obs_right > eliminated

Persuasion < obs_up_right > eliminated

Persuasion < obs_up_left > eliminated

Intention < leave_obstacle > eliminated

Intention < leave_obstacle > created with deadline 0

Persuasion < leave_obstacleDone > created by leave_obstacle
```

```
Persuasion < obs_left > created by leave_obstacle

Persuasion < obs_right > created by leave_obstacle

Persuasion < obs_up_right > created by leave_obstacle

Persuasion < obs_up_left > created by leave_obstacle
ANSWER da go_XY

#ostacolo evitato
Intention < goto_point_XY > : task to execute = GoXY

Intention < goto_point_XY > : task to execute = GoXY

RICEVUTA ANSWER da go_XY

[…]

Intention < goto_point_XY > : task to execute = GoXY

Persuasion < obs_left >: the truth value "false" has been
determined

Persuasion < obs_left >: notification sent to leave_obstacle

Persuasion < obs_right >: the truth value "false" has been
determined

Persuasion < obs_right >: notification sent to leave_obstacle

Persuasion < obs_up_right >: the truth value "true" has been
determined

Persuasion < obs_up_right >: notification sent to leave_obstacle

Persuasion < obs_up_left >: the truth value "false" has been
determined

Persuasion < obs_up_left >: notification sent to leave_obstacle

applicability condition < obs_up_right > is true

piano corrispondente: plan_leave_obstacle3

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Persuasion < obs_up_right >: the truth value "false" has been
determined

Persuasion < obs_up_right >: notification sent to leave_obstacle

Persuasion < obs_up_left >: the truth value "true" has been
determined

Persuasion < obs_up_left >: notification sent to leave_obstacle
```

```
#trovato un ostacolo sulla destra
Intention < leave_obstacle > : task to execute = ChangeDirLeft

Persuasion < obs >: the truth value "false" has been determined

Persuasion < obs >: notification sent to avoid_collision

Persuasion < obs_up_right >: the truth value "false" has been
determined

Persuasion < obs_up_right >: notification sent to leave_obstacle

Intention < avoid_collision > : plan revision needed

Intention < leave_obstacle > : plan revision needed

new state INT_WAITING_OTHER_NOTIFICATION

Persuasion < leave_obstacleDone >: the truth value "true" has
been determined

#ostacolo evitato
Persuasion < leave_obstacleDone >: notification sent to
leave_obstacle

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Intention < leave_obstacle > signals "Intention terminated" to <
avoid_collision >


RICEVUTA ANSWER da leave_obstacle

Persuasion < obs >: the truth value "true" has been determined

Persuasion < obs >: notification sent to avoid_collision

 intenzione leave_obstacle 6

 intenzione leave_obstacle 5

 intenzione leave_obstacle 4

Persuasion < leave_obstacleDone >: the truth value "false" has
been determined

#ostacolo evitato
Persuasion < leave_obstacleDone >: notification sent to
leave_obstacle

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY
```

```
[…]

#individuato ostacolo, necessaria la revisione dei piani
Intention < avoid_collision > : plan revision needed

#il nuovo piano è evitare l'ostacolo
Intention < avoid_collision > : task to execute = LeaveObstacle

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

 intenzione leave_obstacle 3

 intenzione leave_obstacle 2

 intenzione leave_obstacle 1

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Persuasion < leave_obstacleDone >: the truth value "false" has
been determined

Persuasion < leave_obstacleDone >: notification sent to
leave_obstacle

#ostacolo evitato

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Persuasion < obs_left >: the truth value "false" has been
determined

Persuasion < obs_left >: notification sent to leave_obstacle

Persuasion < obs_right >: the truth value "true" has been
determined

Persuasion < obs_right >: notification sent to leave_obstacle
```

```
Persuasion < obs_up_right >: the truth value "false" has been
determined

Persuasion < obs_up_right >: notification sent to leave_obstacle

Persuasion < obs_up_left >: the truth value "false" has been
determined

Persuasion < obs_up_left >: notification sent to leave_obstacle

applicability condition < obs_right > is true

Intention < leave_obstacle > : the chosen plan is
plan_leave_obstacle2

Intention < avoid_collision > : the chosen plan is empty

Persuasion < leave_obstacleDone > eliminated

Persuasion < obs_left > eliminated

Persuasion < obs_right > eliminated

Persuasion < obs_up_right > eliminated

Persuasion < obs_up_left > eliminated

Intention < leave_obstacle > eliminated
an_leave_obstacle2

#una volta evitato l'ostacolo procede dritto

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Intention < leave_obstacle > : task to execute = GoForward

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Intention < leave_obstacle > : task to execute = GoForward

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY
[…]

Intention < leave_obstacle > : task to execute = GoForward

RICEVUTA ANSWER da go_XY
```

```
Intention < goto_point_XY > : task to execute = GoXY

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Persuasion < obs_right >: the truth value "false" has been
determined

Persuasion < obs_right >: notification sent to leave_obstacle

Persuasion < obs >: the truth value "false" has been determined

Persuasion < obs >: notification sent to avoid_collision

Intention < leave_obstacle > : plan revision needed

new state INT_WAITING_OTHER_NOTIFICATION

Intention < avoid_collision > : plan revision needed

Persuasion < leave_obstacleDone >: the truth value "true" has
been determined

Persuasion < leave_obstacleDone >: notification sent to
leave_obstacle

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

Intention < leave_obstacle > signals "Intention terminated" to <
avoid_collision >


RICEVUTA ANSWER da leave_obstacle

Persuasion < obs_right >: the truth value "true" has been
determined

Persuasion < obs_right >: notification sent to leave_obstacle

 intenzione leave_obstacle 6

 intenzione leave_obstacle 5

 intenzione leave_obstacle 4

 intenzione leave_obstacle 3

RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY

 intenzione leave_obstacle 2

 intenzione leave_obstacle 1
```

```
RICEVUTA ANSWER da go_XY

Intention < goto_point_XY > : task to execute = GoXY
```