Universidad Politécnica de Cartagena

TRABAJO FIN DE ESTUDIOS

Implementación en Erlang de máquina virtual para la ejecución de programas Teleo-Reactivos en dispositivos para Internet de las Cosas

Erlang Implementation of a virtual machine for executing Teleo-Reactive Programmes in the Internet of Things

Date:	July 21, 2017
Author:	Elias Antolinos García
Director:	D. Pedro Sánchez Palma
Co-director:	D. Diego Fernández Álvarez

Contents

List of Figures

Abbreviations

TR	Teleo-Reactive
BS	BeliefStore
PID	Process Identifier
OTP	Open Telecom Platform
GPIO	General Purpose Input/Output
IoT	Internet of Things

1. Overview

This chapter describes the background surrounding this thesis, and it provides a description of its objectives.

1.1 Introduction

"By 2020 there will be 50 billion of "things" connected to the Internet" (*The Internet of Things [INFOGRAPHIC]*, 2011). This requires that a significant proportion of these devices have to be able to operate in changing environments. As a consequence, they need to react according to these dynamic changes, and this fact is not always easily solved with the current artificial intelligence developed.

As a result of this problem, Nils J. Nilsson introduces the Teleo-Reactive paradigm (Nilsson, 1994). This approach allows agents to work in dynamic environments and react to different environment's changes while they are following a certain goal.

Furthermore, it is necessary to reproduce real-world parallelism and concurrency in the devices behavior. This implies that they need to perform several actions simultaneously. This is where concurrent programming languages are given center stage. The chosen language for this thesis is Erlang, since it allows to build scalable soft real-time systems with lightweight processes. Thus, it can be used in the Internet of Things without high requirements.

Therefore, and with the aim of exploiting the advantages of TeleoR, which is an extension of the Teleo-Reactive language, in everyday devices, this thesis gathers an Erlang implementation of a interpreter of Teleo-Reactive programs.

1.2 Objectives

In this project an architecture for the interpretation of TeleoR programs in Erlang is presented. To be able to undertake this work, the following objectives have been set:

- Studying the Teleo-Reactive approach: The functionality of the Teleo-Reactive paradigm is being studied briefly, in order to know its main characteristics, focusing in its extension: TeleoR.

- Studying the programming language Erlang: An introduction to functional programming and the Erlang language are being studied. It covers the Erlang syntaxes and the Open Telecom Platform framework (OTP).
- Implementing the interpretation architecture: It is going to be design a general procedure to translate a TeleoR program into Erlang, using three versions of a concrete TeleoR program.
- Testing the implementation in a Raspberry PI: The implementation of an example of TeleoR program is going to be tested in a Raspberry PI, so as to check its proper working.

2. Teleo-Reactive Paradigm

This chapter describes the main fundamentals and basis of Teleo-Reactive programs. Besides, it exposes the current state of Teleo-Reactive programming.

2.1 Introduction to Teleo-Reactive Paradigm

The Teleo-Reactive (TR) approach was introduced by Nilsson (Nilsson, 1994) for systematizing actions for autonomous agents in an environment which is susceptible to changes. Thus, a TR program directs agents towards achieving a specific goal, which can change depending on the environment circumstances.

A TR program can be defined as a set of ordered rules:

$$\begin{array}{lcl} K_1 & \rightarrow & a_1 \\ K_2 & \rightarrow & a_2 \\ & \vdots & \\ K_i & \rightarrow & a_i \\ & \vdots & \\ K_m & \rightarrow & a_m \end{array}$$

The K_i are conditions on perceptual inputs and on a model of the environment, while the a_i are actions on the environment or which change the environment. The list of rules is scanned from the top, where K_1 is the first rule, in other words, the rule with the highest priority. The first action executed belongs to the first rule whose condition part is satisfied. This action a_i can be a single action or a TR program.

With the aim of being able to react to changes in the environment, the list of rules that compose the TR program is constantly evaluated. Therefore, the system is able to react to these changes and execute other actions (considering that the first condition that is true has changed).

Moreover, Nilsson states that the actions of TR programs can be durative rather than discrete, and this differs from conventional systems. A discrete action is executed once, so it can not be interrupted, for example, "open gripper" or "move forward two meters". On the other hand, durative actions are executed while the condition which has led to its execution is true, for instance, "move" or "rotate". Durative actions can be stopped and modified with the execution of other rule.

Other authors, such as Gubisch (Gubisch, Steinbauer, Weiglhofer, & Wotawa, 2008), extends the TR paradigm by considering the simultaneous invocation of actions for a single condition:

$$K_i \quad \rightarrow \quad \begin{matrix} a_i, \\ a_j \end{matrix}$$

When the K_i condition is satisfied, a_i and a_j are activated.

2.2 TeleoR and QuLog

Currently, there is an extension of the original TR language developed by Nilsson, called TeleoR, and a logic programming language to implement TR programs derived from QuProlog and Prolog, called QuLog. This extension have been developed by Keith L. Clark and Peter J. Robinson (Clark & Robinson, 2015).

TeleoR

TeleoR programs are executed in softwares that can execute multiple processes or threads concurrently. This software architecture can be divided in four blocks: a BeliefStore, a Message Handler, a Percepts Handler and a TeleoR Evaluator, as can be appreciated on Figure 2.1.

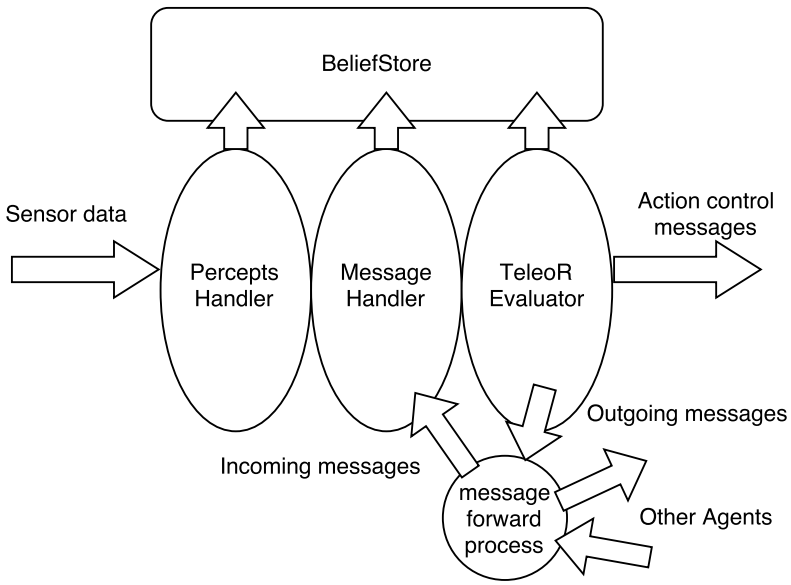

Figure 2.1: Multi-Threaded TeleoR Architecture (Clark & Robinson, 2015)

The BeliefStore contains the agent's knowledge, which is composed of facts and percepts taken from the environment (through sensors). The Percepts Handler thread receives data from the sensors and, with these information, it modifies the percepts in the BeliefStore. However, it is not the unique form to modify the BeliefStore. Because of the possibility of agents to communicate with other agents, the BeliefStore can be updated with the new information received. In addition, it can also be changed by the TeleoR Evaluator thread, even though it is no the most common way. The main task of the TeleoR Evaluator thread is to evaluate the rules that form part of the TeleoR program. This thread query the BeliefStore to decide which condition is the first that is satisfied and to trigger the execution of its associated action.

QuLog

QuLog is a high-level logic language used for functional programming and pattern match string processing. QuLog was developed with the purpose of managing the BeliefStore: defining the percepts, beliefs and relations. Moreover, with the imperative part of QuLog it is possible to define action procedures to send messages to other agents. Thus, these basic concepts are used in writing of the TR program. Further information can be found at the QuLog website: `http://staff.itee.uq.edu.au/pjr/HomePages/QulogHome.html`.

3. Erlang Programming Language

This chapter describes the functional programming language Erlang, giving an overview of the functional paradigm and the OTP framework.

3.1 Introduction to functional programming

Functional programming uses a different programming paradigm than imperative or object oriented languages. In this paradigm variables has no state, they do not change over time, and they are immutable, they do not change during the execution. Furthermore, there are no side effects, executing a function will not change anything outside its environment. Thus, it makes easier to predict and understand the behavior of a program. Other concepts to highlight in functional programming are that iteration (looping) is commonly achieved by recursion, there is no execution order (the order in which commands are carried out does not change the final result), and functional programming allows functions to be treated as values.

The main theoretical basis of the functional paradigm is the Lambda Calculus (Michaelson, 2011), which was developed in the 1930s by Alonzo Church as a computation model with the same computational power that a Turing machine. The Calculus Lambda provides a simple notation of definition of mathematic functions. It is only necessary three syntactic rules to define Lambda expressions:

- Lambda expression to define unnamed functions
- Abstraction to name a Lambda expression
- An application to evaluate a Lambda expression

The application of a Lambda expression to an argument is achieved replacing in the body of a Lambda expression its variable associated with the argument.

Within the functional programming languages, despite the fact that LISP, Clojure, Erlang, Haskell and F# are the most representative, programming in a functional style can also be applied to non-functional languages as PHP, Java 8, Perl or Scala.

3.2 Introduction to Erlang

Erlang is a concurrent, functional programming language. It was developed by Joe Armstrong, Robert Virding and Mike Williams in 1986 for the Swedish telecom Ericsson. However, it was released as open-source in 1998.

Erlang was developed from a telecommunication point of view: millions of parallel conversations happening at the same time. This led to companies as Amazon, Facebook, Yahoo!, Whatsapp, T-Mobile or Ericsson used Erlang in their production systems (*What is Erlang*, 2017).

The main reason for using Erlang instead of other functional languages is because of the ability of handle concurrency (Armstrong, 2013). By concurrency is meant that several processes can be handled at the same time. Concurrent programming can be used to improve the performance of systems, to make them scalable and fault-tolerant or to write more clear programs for real-world applications:

- Performance: The possibility of execute several threads at the same time has allowed that programs written in Erlang more than ten years ago for a sequential machine run faster with the technology improvement (multicores).
- Scalability: Due to the lightweight of processes, systems are easily scalable by increasing the number of processes and CPUs.
- Fault tolerance: Erlang was designed for building fault-tolerant telecommunications system, and it was possible because of the independence of processes, so a process failure can not provoke that accidentally other process crashes.
- Clarity: In Erlang is possible to reflect the real-world parallelism that in sequential programming is not easily attainable.

On the other hand, pattern matching is other of the big strengths of Erlang, where variables are bound to values through this mechanism. For example,

```
1> {X,Y}={1,2}.
{1,2}
2> X.
1
```

According to data representation, in Erlang there are only a few data types, but it is possible to achieve a lot using them. The most characteristics are:

- Atoms: An atom is a kind of string constant that is only identified with the characters in the string. They normally starts with a lower-case letter, underscore(_) or @. The use of atoms is similar to *enum* constants in Java or C, they are like labels.
- Tuples: A tuple is a ordered sequence of a fixed number of Erlang terms. They are written within curly brackets:

```
{one,"dog",{17,[]}}
```

- Lists: A list is a composed of a variable number of Erlang terms. They are written within square brackets:

```
[1,2,{cat,"Garfield"}]
```

- Pid: This data type is a process identifier, and it is unique for each process. The function *self()* returns the pid of the process that is currently running.

```
1> self().
<0.89.0>
```

- Funs: A fun is a functional object. In Erlang, due to being a functional programming language, functions are handle as data. It makes possible to create anonymous functions and pass them as arguments of other functions.

Further information in this programming language can be found at the Erlang Website (*What is Erlang*, 2017), and in the book *Programming Erlang: Software for a Concurrent World* (Armstrong, 2013).

3.3 Open Telecom Platform framework. OTP

OTP was originally an acronym for Open Telecom Platform. However, currently is not specific to telecom applications. It is a framework composed of a set of modules and standards designed to help the applications building.

The main advantages of OTP (Logan, Merritt, & Carlsson, 2011) are:

- Productivity: It makes possible to produce systems in a very short time.
- Stability: Code written on top of OTP can focus on the logic and avoid error-prone reimplementations of the typical things that every real-world system needs: servers, process management and state machines.
- Supervision: The application structure provided makes it simple to supervise and control the systems.
- Upgradability: It provides patterns for handling code upgrades.
- Reliable code base: The framework code is solid and has been thoroughly tested.

The central concept of program applications using OTP is the OTP *behaviour*. A behaviour is a formalization of a common pattern. The idea is to divide the code for a process in a generic part (a behaviour module) and a specific part (a callback module).

The most commonly abstraction used in the OTP system is the *gen server*, the following is a simplified template:

```
-module().
%gen_server_template
-behaviour(gen_server).
-export({start_link/0}).
%gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
 terminate/2, code_change/3]).

start_link() -> gen_server:start_link({local,?SERVER}}, ?MODULE, [],[]).
init([])->{ok,State}.

handle_call(_Request, _From, State) -> {reply, Reply, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

The variable *State* contains the global state of the server that gets passed around in the server. And the variable *Reply* contains the values that are going to be send back to the client as the return values of the calls.

According to the main functions, the *handle_call* and the *handle_cast* are the most important ones.The *handle_call/3* callback is used to handle synchronous messages, while the *handle_cast/2* function is used to work with asynchronous calls.

In addition, it is also important to highlight the *code_change/3* function. This callback lets a code upgrade, where *_OldVsn* is the version term in the case of an upgrade and *{down,Vsn}* in the case of reloading old code (downgrade).

Despite the *gen_server* behavior is the most used in OTP systems, it can be found other behaviors such as the *gen_event* that is an event manager. Anyway, one of the most useful part of OTP, besides the *gen_server* behavior, is the supervisor behavior. This behavior can be used to keep our software working in cause of errors. The supervisors trap exit signals of processes that are link to it and it can restart them in case of fault. Moreover, OTP provides libraries to work with supervisors and workers, so it allow to build *supervision trees*, as can be shown in the Figure 3.1.

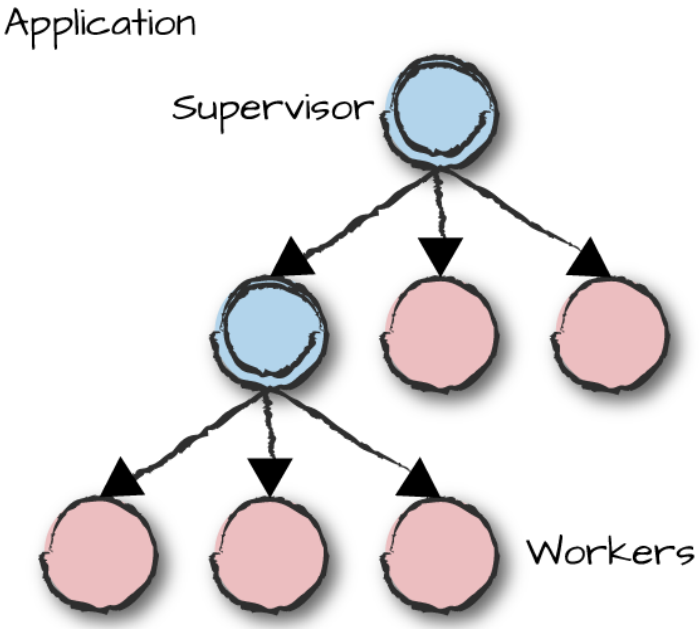

Figure 3.1: OTP supervisor tree (Vinoski & Cesarini, 2016)

4. Teleo-Reactive Specification

This chapter exposes the implementation in Erlang of three version of a Teleo-Reactive program, increasing in complexity progressively. These examples were extracted from a program developed by Keith L. Clark and it is based on collecting bottles with robots.

The Erlang implementation consists of four modules: one belong to the BeliefStore, other to the TeleoR program, other to the execution of actions and the last one to the writing of the actions in a log-file. The structure of these modules is thoroughly explained in chapter 6.

Moreover, these three version share the same core structure, which is based in the pattern matching mechanism. As explained in section 3.2, the pattern matching is one of the strong points of Erlang. In this implementation, the pattern matching has a key role. The evaluation of percepts and beliefs is implemented using this mechanism, and it follows this structure:

```
case {A_1,A_2,A_3,...,A_n} of
  {true,_,_,...}->
     action_1;
  {_,true,_,_,..}->
     action_2;
        .
        .
  {_,_,_,_,...}->
     action_n
end.
```

Where A_i represents the conditions that are evaluated. The agent executes the first action whose left statement is true. This process resembles to an AND statement. Thus, to focus in the satisfying of a specific condition , it takes advantage of using the "_" as variable. When the "_" is used, it ignores the value of this variable, so it always matches. Then, it allows to cover all the cases following a certain order.

Regarding the modules that compose these examples, the BeliefStore module and the executor module are the most important ones, besides the TR itself. The BeliefStore module was implemented in order to manage the system state and its environment. The BeliefStore has a *gen_server* behavior, since it is going to update the system state according to the agent's and environment's changes. It is comprised of percepts, which is information taken from the environment (through sensors), and beliefs (which is information inferred by the agent).

On the other hand, the executor module includes the set of functions to control the Teleo-Reactive program execution. It also includes functions to endure the execution of a function, obtain the priority of conditions and compare values from the BeliefStore.

4.1 Version 1

The TeleoR program of the first version implements the behavior of a single robot to collect bottles in a drop.

The TeleoR code is shown below:

```
%% Version 1
dir::= left | right
thing::= bottle | drop

percept
        gripper_open: (),
        holding : (),
        see : (thing),
        touching : (),
        over_drop : ()

durative
        move : (num),
        turn : (dir)

discrete
        open_gripper : (),
        close_gripper : ()

int _collected:=0

collect_bottles: ()
collect_bottles(){
        _collected >= 5 ~> ()
        holding & over_drop while 3 ~> drop_and_leave
        holding ~> get_to_drop
        true ~> get_bottle
}

drop_and_leave : () ~>
drop_and_leave(){
        gripper_open ~> leave_drop
        true ~> open_gripper ++ _collected := _collected + 1;
}

leave_drop : () ~>
leave_drop {
        not see(drop) & not see(robot) ~> move(1.0)
        see(drop) ~> turn(left, 0.8)
        see(robot)  ~> turn(right, 0.8)
}

get_to_drop:
get_to_drop(){
```

```
        over_drop  ~> ()
        see(drop) ~> move(1.5)
        true ~> turn(left,1.0) for 2; move(1.0) for 2
}

get_bottle:
get_bottle(){
        holding ~> ()
        touching & gripper_open ~> close_gripper
        touching ~> open_gripper
        see(bottle) ~> move(1.5)
        true ~> turn(left,1.0) for 2.8; move(1.0) for 2
}

go : ()
go() ~> collect_bottles()
```

In this example, it is possible to distinguish a preamble, where the percepts, actions, variables and enums are declared. Then there is a main function, which controls the main functionality of the TR. In this function there are goals that derivate in the execution of subgoals.

According to the performance of the TR, it has a function to get a bottle, other to take the bottle to the drop, other to leave the drop and other to drop the bottle. These functions are executed depending on the percepts and beliefs. After that, each function has internal actions/subgoals that are executed depending also in the beliefs and percepts. It highlights functions such as *move*, *turn*, *open_gripper*, *close_gripper* with different parameters (depending on the goal that they belong to). There are also actions to manage the BeliefStore.

4.1.1 BeliefStore module

The functions used to manage the BeliefStore are defined in the code below:

```
-module(bs).

-behaviour(gen_server).

%% ================================================
%% API FUNCTIONS
%% ================================================

-export([start_link/0,
    add_belief/1,
    update_belief/1,
    remove_belief/1,
    remove_one_belief/1,
    get_belief/1,
```

```
    get_bs/0,
    is_belief/1,
    is_belief/2,
    stop/0]).

%% ====================================================
%% gen_server FUNCTIONS
%% ====================================================
-export([init/1,
    handle_call/3,
    handle_cast/2,
    handle_info/2,
    terminate/2,
    code_change/3]).
```

Two parts can be distinguished in the BeliefStore implementation: one belonging to the API definition and other to the *gen_server* definition. The API definition includes the set of functions from which the agent can interface.
These functions are implemented in the following code:

```
%% ================================================
%% API FUNCTION DEFINITIONS
%% ================================================

start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

add_belief(Belief)->
    gen_server:cast(?MODULE,{add,Belief}).

update_belief(Belief)->
    gen_server:cast(?MODULE, {update, Belief}).

remove_belief(Belief)->
    gen_server:cast(?MODULE, {remove, Belief}).

remove_one_belief(Belief)->
    gen_server:cast(?MODULE, {remove_one, Belief}).

get_belief(Belief)->
    gen_server:call(?MODULE,{get,Belief}).

is_belief(Key)->
    gen_server:call(?MODULE,{is_belief,Key}).

is_belief(Key1,Key2)->
    gen_server:call(?MODULE,{is_belief,Key1,Key2}).

get_bs()->
```

```
    gen_server:call(?MODULE,{get_bs}).

stop()->
    gen_server:cast(?MODULE,stop).
```

The *start_link/0* function and the *stop/0* function let the start and stop of the server that is going to control the BeliefStore. Besides, the rest of functions are differentiate in two groups: functions that query the BeliefStore and functions that manage the BeliefStore.

The functions that control the BeliefStore are:

- *add_belief/1*: This function enables to insert new knowledge to the BeleifStore.
- *update_belief/1*: This function allows to update previous knowledge in the BeleifStore.
- *remove_belief/1*: This function enables to delete precepts from the BeleifStore.
- *remove_one_belief/1*: Since one belief can have several values, this function allows to delete only one value from the percept. For instance, the robot can see a bottle, the drop and other robot at the same time:

  ```
  see=>[bottle,drop,robot]
  ```

Furthermore, the functions that query the BeliefStore are:

- *get_belief/1*: This function returns the value of a certain belief.
- *is_belief/1*: This function returns true if a certain belief exists in the BeliefStore.
- *is_belief/2*: Since one belief can have several values (being a sequence of tuples), this function returns true if there is a belief fitting this structure:

  ```
  Key1=>[{Key2,Value}]
  ```

  For example,

  ```
  see=>[{robot,12},{bottle,69}]
  ```

- *get_bs/0*: This function returns the entire BeliefStore.

On the other side, the implementation of the *gen_server* functions is described in the following code:

```
init([])->
   {ok, #{see=>
            [],
         collected=>
            0,
         other_collected=>
            0,
         last_executed=>
            {[],[],[]},
         executing=>
            {[],[],[]},
         to_execute=>
            {[],[],[]},
         timer=>
            [],
         priority=>
            [{collect_bottles,[collected, holding_over_drop,
            holding, true,[]]},
            {drop_and_leave,[gripper_open,true,[]]},
            {leave_drop,[not_see,see_drop,see_robot,[]]},
            {get_to_drop,[over_drop, see_drop, true,[]]},
            {get_bottle,[holding, touching_gripper_open,
            touching, see_bottle, true,[]]},
            {[],[]}]}}.

handle_call({get_bs},_From,State)->
   {reply,State,State};

handle_call({get,Key},_From,State)->
   case maps:is_key(Key, State) of
         true->
            {reply,maps:get(Belief,State),State};
         _->
            {reply,false, State}
   end.

handle_call({is_belief,Key},_From,State)->
   {reply,maps:is_key(Key,State),State};

handle_call({is_belief,Key1,Key2},_From,State)->
   case maps:is_key(Key1, State) of
         true->
            List=maps:get(Key1, State),
            case proplists:lookup(Key2, List)
               none ->
                  {reply,false,State};
               _->
                  {reply,true,State}
             end;
          false->
             {reply,false,State}
```

```
   end.

handle_cast(stop,State)->
   {stop,normal,State};

handle_cast({add,{Key,Value}},State)->
   case maps:is_key(Key, State) of
        true->
           case maps:get(Key, State)=:=[] of
                true->
                   {noreply,State#{Key => [Value]}};
                false->
                   {noreply,State#{Key =>
                   ordsets:add_element(Value, maps:get(Key,
                    State))}}
           end;
         _->
            {noreply, maps:put(Key,Value,State)}
   end;

handle_cast({update,{Key,Value}},State)->
   case maps:is_key(Key, State) of
        true->
           case is_list(maps:get(Key, State)) of
                true->
                   io:format("Belief␣updated:␣~p.~n",[{Key,Value}]),
                   {noreply,State#{Key => [Value]}};
                _->
                   io:format("Belief␣updated:␣~p.~n",[{Key,Value}]),
                   {noreply,State#{Key => Value}}
           end;
        false->
           {noreply,State}
   end;

handle_cast({remove,Key},State)->
   case maps:is_key(Key, State) of
        true->
           io:format("Belief␣removed:␣~p.~n",[Key]),
           {noreply,maps:remove(Key, State)};
        false->
           {noreply, State}
   end;

handle_cast({remove_one,{Key,Subkey}},State)->
   case maps:is_key(Key, State) of
        true->
           List=maps:get(Key, State),
           case proplists:lookup(Subkey, List) of
                none->
                   {noreply,State};
```

```
                _->
                    io:format("Belief␣removed:␣~p.~n",[{Key,Subkey}]),
                    {noreply,State#{Key => proplists:delete(Subkey,
                    maps:get(Key, State))}}
            end;
        false->
            {noreply,State}
    end.

handle_info(Info, State) ->
    error_logger:info_msg("~p~n", [Info]),
    {noreply, State}.

terminate(_Reason, _State) ->
    error_logger:info_msg("terminating~n"),
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

The *init/0* function initializes the BeliefStore state, it includes some beliefs such as the number of bottles collected, the elements seen, and some beliefs related with the execution of the actions. These last beliefs are thoroughly explained in the Executor module.

This function also includes a list with the same structure than the TeleoR root, in order to get the priority of each TeleoR condition. In each list of subgoals that belongs to a main goal, there is an extra subgoal: *[]*. It allows to compare main goals.

The rest of functions are divided in two, *handle_call* & *handle_cast*, where only the input arguments vary. Due to pattern matching mechanism, it is possible to select the function that is going to be executed just varying the arguments. In these cases, the function that is executed is determined with an atom (which can be, e.g., stop, add, remove, remove_one or update, for the *handle_cast* function).

The different implementations of *handle_call*, according to its first atom are:

- get_bs: This implementation returns the entire BeliefStore, which is represented with the variable *State*.
- get: This implementation returns the value of the element *Belief*.
- is_belief: This implementation returns true if the element represented with *Key1* or *Key1* and *Key2* exists in the BeliefStore.

On the other hand, the different implementations of *handle_cast*, according to

its first atom are. These functions do not return any value, they only modify the BeliefStore, which is meant, the *State* variable:

- stop: This implementation stop the server.
- add: This implementation creates a new belief in the BeliefStore. If the belief is already in the BeliefStore, it adds a new value to this belief.
- update: This implementation modifies the value of a known belief.
- remove: This implementation deletes a known belief.
- remove_one: This implementation deletes a value from a belief.

The rest of functions has to be implemented due to the *gen_server* behavior, but they are not used in these examples.

4.1.2 Executor module

The functions that can be used from other modules are the following ones:

```
-module(executor).

%% ================================================
%% API FUNCTIONS
%% ================================================
-export([execute_while/1,
   execute/1,
   while_condition/1,
   remember/2,
   compare_value/4]).
```

The functions of this module can be blocked in three groups: functions that control the execution of an action, functions that control priority and inner functions.

Within the functions that control te execution of actions, discrete and durative actions are distinguished. The durative actions are implemented with two discrete actions, which represents the start and end of the action.

Each function is determined with three variables/parameters:*Rule*, which represents the goal and subgoal it belongs to; *Fun*, which represents the name of the function; and *Args*, which includes the arguments of this function. It is also important to point that the module, from which the function belongs, needs to be specified to execute it.

Therefore, the implementation of these functions is the following:

```
execute_while([Rule,Fun,Args,true])->
   case check_execution(Rule,Fun,Args) of
        no_action->
           no_action;
        _->
           bs:update_belief({to_execute,{Rule,Fun,Args}}),
           execute([Rule,Fun,Args])
   end;

execute_while([Rule,Fun,Args,Time])->
   case check_execution(Rule,Fun,Args) of
        no_action->
           no_action;
        _->
           bs:update_belief({to_execute,{Rule,Fun,Args}}),
           % Debugging lines
           writer:writeIt(debug,io_lib:fwrite("Instruction
            execute_while ~p.~n",[{Rule,Fun,Args}])),
           writer:writeIt(debug,io_lib:fwrite("Executing ~p.~n",
            [bs:get_belief(executing)])),
           writer:writeIt(debug,io_lib:fwrite("Last executed ~p.~n",
            [bs:get_belief(last_executed)])),
           writer:writeIt(debug,io_lib:fwrite("To execute ~p.~n",
            [bs:get_belief(to_execute)])),
           %
           execute([Rule,Fun,Args]),
           {ok,TRef}=timer:apply_after(Time, tr, Fun,
            [{finalize,Rule,Args}]),
           % Debugging lines
           writer:writeIt(debug,io_lib:fwrite("Timer ~p.~n",
            [{TRef,Rule,Fun,Args}])),
           %
           bs:update_belief({timer,TRef})
        end.

execute([Rule,Fun,Args])->
   case check_execution(Rule,Fun,Args) of

        execute->
           bs:update_belief({to_execute,{Rule,Fun,Args}}),
           % Debugging lines
           writer:writeIt(debug,io_lib:fwrite("Instruction
            execute_while ~p.~n",[{Rule,Fun,Args}])),
           writer:writeIt(debug,io_lib:fwrite("Executing ~p.~n",
            [bs:get_belief(executing)])),
           writer:writeIt(debug,io_lib:fwrite("Last executed ~p.~n",
            [bs:get_belief(last_executed)])),
           writer:writeIt(debug,io_lib:fwrite("To execute ~p.~n",
            [bs:get_belief(to_execute)])),
           %
```

```
        tr:Fun({start,Rule,Args}),
        bs:update_belief({to_execute,{[],[],[]}});

    update->
        bs:update_belief({to_execute,{Rule,Fun,Args}}),
        % Debugging lines
        writer:writeIt(debug,io_lib:fwrite("Instruction
         execute_while ~p.~n",[{Rule,Fun,Args}])),
        writer:writeIt(debug,io_lib:fwrite("Executing ~p.~n",
         [bs:get_belief(executing)])),
        writer:writeIt(debug,io_lib:fwrite("Last executed ~p.~n",
         [bs:get_belief(last_executed)])),
        writer:writeIt(debug,io_lib:fwrite("To execute ~p.~n",
         [bs:get_belief(to_execute)])),
        %
        {Rule2,Fun2,Args2}=bs:get_belief(executing),
        case bs:get_belief(timer)=:=[] of
            false->
                timer:cancel(bs:get_belief(timer)),
                % Debugging lines
                writer:writeIt(debug,io_lib:fwrite("Kill ~p.~n",
                 [bs:get_belief(timer)])),
                %
                bs:update_belief({timer,[]});
            _->
                no_action
        end,
        tr:Fun({update,Rule,Args}),
        bs:update_belief({to_execute,{[],[],[]}});

    execute_priority->
        bs:update_belief({to_execute,{Rule,Fun,Args}}),
        % Debugging lines
        writer:writeIt(debug,io_lib:fwrite("Instruction
         execute_while ~p.~n",[{Rule,Fun,Args}])),
        writer:writeIt(debug,io_lib:fwrite("Executing ~p.~n",
         [bs:get_belief(executing)])),
        writer:writeIt(debug,io_lib:fwrite("Last executed ~p.~n",
         [bs:get_belief(last_executed)])),
        writer:writeIt(debug,io_lib:fwrite("To execute ~p.~n",
         [bs:get_belief(to_execute)])),
        %
      {Rule2,Fun2,Args2}=bs:get_belief(executing),
      case bs:get_belief(timer)=:=[] of
            false->
                timer:cancel(bs:get_belief(timer)),
                % Debugging lines
                writer:writeIt(debug,io_lib:fwrite("Kill ~p.~n",
                 [bs:get_belief(timer)])),
                %
                bs:update_belief({timer,[]});
```

```
                _->
                    no_action
            end,
            tr:Fun2({finalize,Rule2,Args2}),
            tr:Fun({start,Rule,Args}),
            bs:update_belief({to_execute,{[],[],[]}});
        _->
            no_action
    end.
```

There are two ways to execute durative actions: one where the action is executed while the condition is true, and other that is only executed for a certain period of time. This last case implies that it is necessary to run a timer to finish the action after the set time.

Since the TeleoR conditions are constantly evaluated, it is necessary an implementation to control the execution of actions. Thus, it avoids the re-execution of actions, and it also can stop actions to execute other (e.g., when the condition of the action executed is not longer true). With the aim of carrying out this process, the last action executed, the action that is executing and the action that is going to be executed are stored in the BeliefStore. With this information is decided if an action is executed or not, through the calculation of its priority. The diagrams of the execution of a durative and a discrete action are displayed in the Figure 4.1 & Figure 4.2, respectively.

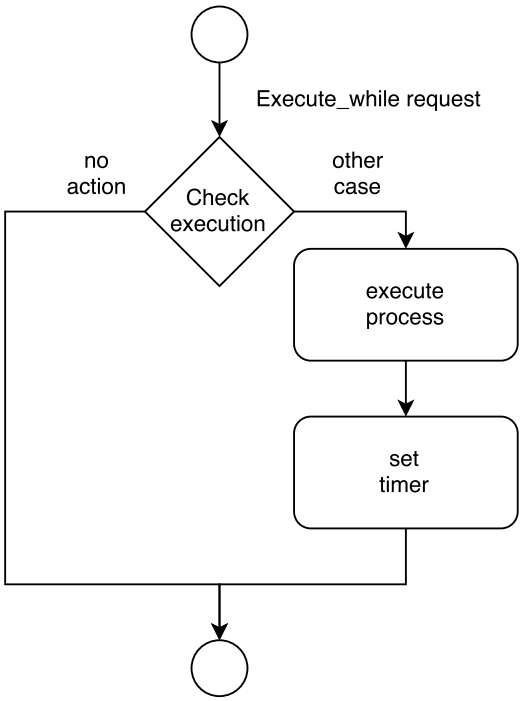

Figure 4.1: Durative action execution diagram

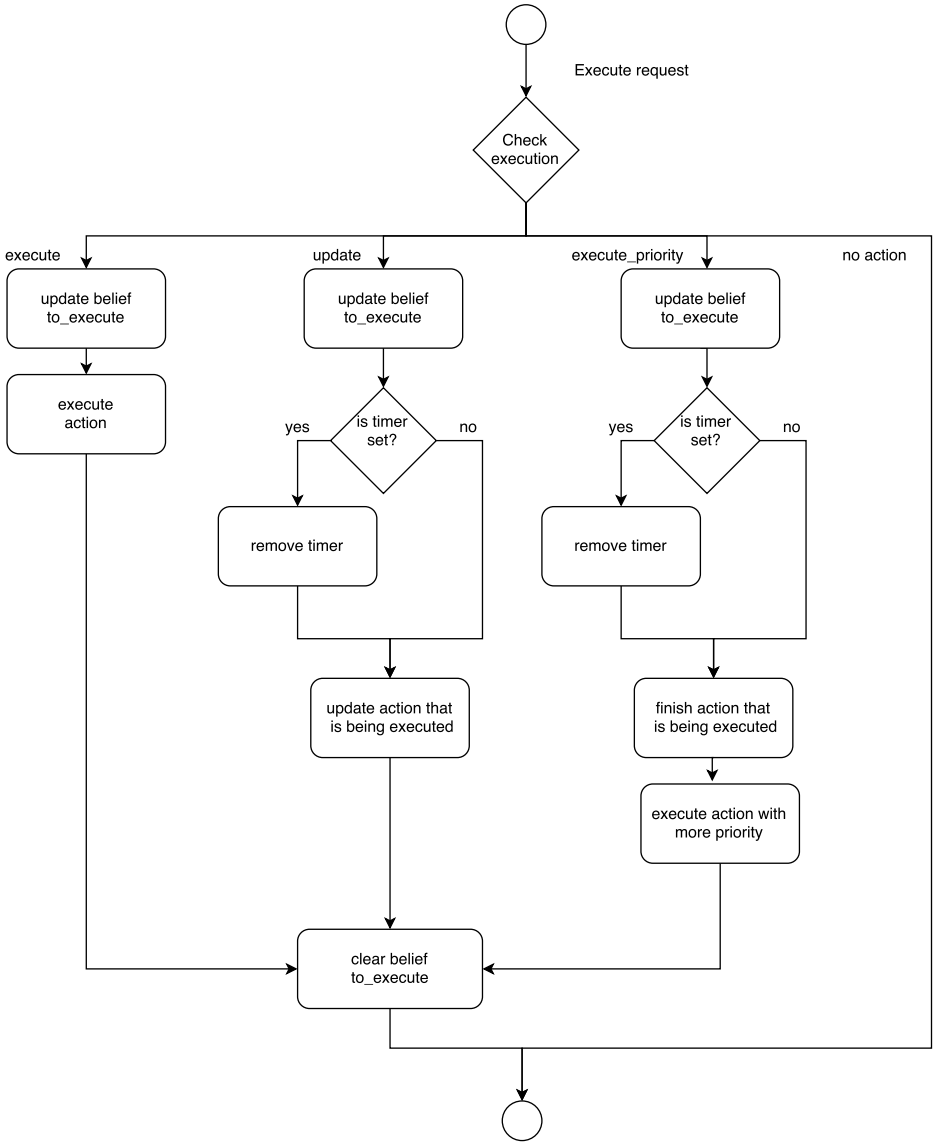

Figure 4.2: Discrete action execution diagram

The implementation of the functions to check the execution of an action and to calculate the priority is the following:

```
check_execution(Rule,Fun,Args)->
   {ExecRule,ExecFun,ExecArgs}=bs:get_belief(executing),
   {LastRule,LastFun,LastArgs}=bs:get_belief(last_executed),
   {NextRule,NextFun,NextArgs}=bs:get_belief(to_execute),

   kill_while(Rule),

   case ({ExecRule=:=[],priority(get_priority(Rule,
    bs:get_belief(priority)), get_priority(ExecRule,
    bs:get_belief(priority))),Fun=:=ExecFun,
    Args=:=ExecArgs,Fun=:=LastFun,Args=:=LastArgs}) of
        {true,_,_,_,true,true}->
           no_action;
        {true,_,_,_,_,_}->
           case ({NextRule=:=Rule andalso NextFun=:=Fun andalso
            NextArgs=:=Args, priority(get_priority(NextRule,
            bs:get_belief(priority)),get_priority(Rule,
            bs:get_belief(priority)))}) of
                {true,_}->
                   execute;
                {_, less_priority}->
                   execute;
                {_,_}->
                   no_action
           end;
        {_,same_priority,_,_,_,_}->
           no_action;
        {_,_,true,_,_,_}->
           case ({NextRule=:=Rule andalso NextFun=:=Fun andalso
            NextArgs=:=Args, priority(get_priority(NextRule,
            bs:get_belief(priority)),get_priority(Rule,
            bs:get_belief(priority)))}) of
                {true,_}->
                   update;
                {_, less_priority}->
                   update;
                {_,_}->
                   no_action
           end;
        {_,_,_,_,_,_}->
           case ({NextRule=:=Rule andalso NextFun=:=Fun andalso
            NextArgs=:=Args, priority(get_priority(NextRule,
            bs:get_belief(priority)),get_priority(Rule,
            bs:get_belief(priority)))}) of
                {true,_}->
                   execute_priority;
                {_, less_priority}->
                   execute_priority;
```

```
                {_,_}->
                    no_action
            end
    end.

%% ================================================
%% PRIORITY FUNCTIONS
%% ================================================

get_priority({MainFun,SubFun},ListOfLists)->
    get_max_priority({MainFun,SubFun},ListOfLists,1);

get_priority(_,ListOfLists)->
    get_max_priority({[],[]},ListOfLists,1).

get_max_priority(_,[],_)->
    not_found;

get_max_priority({MainFun,SubFun},[{MainFun,SubList}|_],
 MaxPriority)->
    get_sub_priority(SubFun,SubList,MaxPriority,1);

get_max_priority({MainFun,SubFun},[_|List],Priority)->
    get_max_priority({MainFun,SubFun},List,Priority+1).

get_sub_priority(_,[],MaxPriority,_)->
    {MaxPriority,not_found};

get_sub_priority(SubFun,[SubFun|_],MaxPriority,SubPriority)->
    {MaxPriority,SubPriority};

get_sub_priority(SubFun,[_|List],MaxPriority,SubPriority)->
    get_sub_priority(SubFun,List,MaxPriority,SubPriority+1).

priority({A,_B},{C,_D}) when A<C -> more_priority;
priority({A,B},{C,D}) when A=:=C, B<D -> more_priority;
priority({A,B},{C,D}) when A=:=C, B=:=D -> same_priority;
priority(_,_)-> less_priority.
```

The functions to calculate the priority can be divided into two groups: one belonging to get the *priority index* and other to compare two *priority indexes*. The first group, the functions related to get_priority, find the index that belongs to certain condition/action in the BeliefStore (in the priority belief) and returns a tuple with two indexes: first condition index, sub-condition index. While the second group, the functions related to priority, compare two tuples of indexes. The priority is determined knowing that 1 is the highest value. For instance, the priority of the condition set (that in the code implementation is named *Rule*) {*get_bottle,holding*}={5,1} has a lower priority than the condition set {*leave_drop, see_robot*}={3,3}.

On the other hand, in the *check_execution/3* function there is a function named *kill_while/1*. This function finishes the timer process arising from a while rule. In TeleoR programs these rules allow to keep one condition true for a certain time, despite the fact it is not true, while there is any true condition with more priority. The implementation of these functions can be found in the code below:

```
while_condition([Rule,Time])->
   {ok,TRef}=timer:apply_after(Time, bs, remove_belief, [while_timer]),
   bs:update_belief({while_timer,{Rule,TRef}}).

kill_while(Rule)->
   case bs:is_belief(while_timer) of
        true->
           {RuleWhile,TRef}=bs:get_belief(while_timer),
           case priority(get_priority(Rule,bs:get_belief(priority)),
            get_priority(RuleWhile,bs:get_belief(priority))) of
                more_priority->
                   %Debuggin line
                   writer:writeIt(debug,io_lib:fwrite("Kill while timer
                    ~p.~n", [TRef])),
                   %
                   bs:remove_belief(while_timer),
                   timer:cancel(TRef);
                _->
                   no_action
           end;
        _->
           no_action
   end.
```

Finally, the executor module also includes a function to remember a belief for a certain amount of time, and a function to compare a belief with a value. The comparison can be performed with <, > and/or =.
The implementation is the following:

```
remember({Belief,Args},Time)->
   bs:add_belief({Belief,Args}),
   timer:apply_after(Time, bs, remove_belief, [Belief]).

compare_value(Key,KeyList,Mode,Value)->
   case KeyList=:=all of
        true->
           List=bs:get_bs(),
           case bs:is_belief(Key) of
                true->
                   case Mode of
                        minor ->
                           proplists:get_value(Key, List) < Value;
```

```
                        minor_equal ->
                            proplists:get_value(Key, List) =< Value;
                        major ->
                            proplists:get_value(Key, List) > Value;
                        major_equal ->
                            proplists:get_value(Key, List) >= Value;
                        equal ->
                            proplists:get_value(Key, List) =:= Value
                    end;
                _->
                    false
            end;
        false->
            case bs:is_belief(KeyList,Key) of
                true->
                    List=bs:get_belief(KeyList),
                    case Mode of
                        minor ->
                            proplists:get_value(Key, List) < Value;
                        minor_equal ->
                            proplists:get_value(Key, List) =< Value;
                        major ->
                            proplists:get_value(Key, List) > Value;
                        major_equal ->
                            proplists:get_value(Key, List) >= Value;
                        equal ->
                            proplists:get_value(Key, List) =:= Value
                    end;
                _->
                    false
            end
    end.
```

4.1.3 Writer module

This module includes the functions to generate the debug files and the file with the commands to execute. This module controls the writing into files. It is represented in the following code:

```
-module(writer).

%% ====================================================
%% API FUNCTIONS
%% ====================================================
-export([writeIt/2]).
```

```
%% ================================================
%% INTERNAL FUNCTIONS
%% ================================================
writeIt(debug,Sequence)->
    {Mode,File}=bs:get_belief(writer),
    case Mode:=debug of
          true->
              write(Sequence,string:concat(File,"_debug.dat"));
          _->
              no_action
    end;

writeIt(normal,Sequence)->
    {Mode,File}=bs:get_belief(writer),
    case Mode:=debug of
          true->
              write(Sequence,string:concat(File,"_debug.dat"));
          _->
              write(Sequence,string:concat(File,".dat"))
    end.

write(Sequence,File)->
    {ok,IoDevice}=file:open(File,[append]),
    file:write(IoDevice, string:concat(format_utc_timestamp(),
     Sequence)).

format_utc_timestamp() ->
    TS = {_,_,Micro} = os:timestamp(),
    {{Year,Month,Day},{Hour,Minute,Second}} =
     calendar:now_to_universal_time(TS),
    Mstr = element(Month,{"Jan","Feb","Mar","Apr","May","Jun","Jul",
    "Aug","Sep","Oct","Nov","Dec"}),
    io_lib:format("~2w ~s ~4w ~2w:~2..0w:~2..0w.~6..0w ",
     [Day,Mstr,Year,Hour+2,Minute,Second,Micro]).
```

The *writeIt/2* function determines if the command line should be written in the file or not, depending on the type of the file: *debug* file or *normal* file. While the *write/2* function writes the sequence into the file, adding the current timestamp.

4.1.4 TR module

This module includes the Erlang implementation of the TeleoR program. The code of the first version is the following:

```
-module(tr).
%% ================================================
%% API FUNCTIONS
%% ================================================
-export([go/3,
```

```
move/1,
turn/1,
close_gripper/1,
open_gripper/1]).
%% ============================================
%% INTERNAL FUNCTIONS
%% ============================================
go(Mode, File ,Num)->
    bs:start_link(),
    bs:add_belief({writer,{Mode, File}}),
    bs:add_belief({goal,Num}),
    collect_bottles().

collect_bottles()->
    case {bs:get_belief(collected)>=bs:get_belief(goal),
     bs:is_belief(holding), bs:is_belief(while_timer),
     bs:is_belief(over_drop)} of
          {true,_,_,_}->
             writer:writeIt(normal, io_lib:fwrite("~p cans collected,
              task finished.~n", [bs:get_belief(collected)]));
          {_,true,_,true}->
             drop_and_leave();
          {_,_,true,_}->
             drop_and_leave();
          {_,true,_,_} ->
             get_to_drop();
          {_,_,_,_} ->
             get_bottle()
    end.

drop_and_leave()->
    case bs:is_belief(while_timer) of
          true->
             ok;
          _->
             bs:add_belief({while_timer,[]}),
             executor:while_condition([{collect_bottles, holding_over_drop},
              3000])
    end,

    case bs:is_belief(gripper_open) of
          true ->
             leave_drop();
          _ ->
             executor:execute([{drop_and_leave,true}, open_gripper,[]]),
             bs:update_belief({collected, bs:get_belief(collected)+1}),
             collect_bottles()
    end.

leave_drop()->
    case {bs:is_belief(see,drop),bs:is_belief(see,robot)} of
```

```
        {true,_} ->
            executor:execute_while([{leave_drop,see_drop},turn,
            {left,0.8},true]),
            collect_bottles();
        {_,true} ->
            executor:execute_while([{leave_drop,see_robot},turn,
            {right,0.8},true]),
            collect_bottles();
        {_,_} ->
            executor:execute_while([{leave_drop,not_see},move,{1.0},
            true]),
            collect_bottles()
    end.

get_to_drop()->
    case {bs:is_belief(over_drop),bs:is_belief(see,drop)} of
        {true,_} ->
            collect_bottles();
        {_,true} ->
            executor:execute_while([{get_to_drop,see_drop},move,{1.5},
             true]),
            collect_bottles();
        {_,_} ->
            executor:execute_while([{get_to_drop,true},turn,
             {left,1.0},2000]),
            executor:execute_while([{get_to_drop,true},move,{1.0},
             2000]),
            collect_bottles()
    end.

get_bottle()->
    case {bs:is_belief(holding),bs:is_belief(touching),
    bs:is_belief(gripper_open),bs:is_belief(see,bottle)} of
        {true,_,_,_} ->
            collect_bottles();
        {_,true,true,_} ->
            executor:execute([{get_bottle,touching_gripper_open},
             close_gripper,[]]),
             collect_bottles();
        {_,true,_,_} ->
            executor:execute([{get_bottle,touching},open_gripper,
            []]),
            collect_bottles();
        {_,_,_,true} ->
            executor:execute_while([{get_bottle,see_bottle},move,
            {1.5},true]),
            collect_bottles();
        {_,_,_,_} ->
            executor:execute_while([{get_bottle,true},turn,
            {left,1.0},2800]),
            executor:execute_while([{get_bottle,true},move,{1.0},
```

```
            2000]),
            collect_bottles()
    end.
```

In this implementation is possible to highlight some features:

- At the end of each condition, it starts to evaluate all the conditions from the beginning. It is represented with the *collect_bottles/0* call.
- The evaluation of the conditions is ordered, and only the first condition that is true is the one executed.
- The action that is executed is chosen by pattern matching mechanism. At the beginning of each function, all the percepts are checked, so it is possible to perform the pattern matching.

Moreover, there are functions that do not depend on the TeleoR program such as move or turn. These functions are implemented writing in the file the command to execute its action.

```
move({Atom,Rule,Args})->
    case Atom of
        start->
            bs:update_belief({executing,{Rule,move,Args}}),
            writer:writeIt(normal,io_lib:fwrite("Begin move ~p.~n",
             [Args]));
        finalize->
            bs:update_belief({last_executed,{Rule,move,Args}}),
            bs:update_belief({executing,{[],[],[]}}),
            writer:writeIt(normal,io_lib:fwrite("End move ~p.~n",
             [Args]));
        update->
            bs:update_belief({last_executed,
             bs:get_belief(executing)}),
            bs:update_belief({executing,{Rule,move,Args}}),
            writer:writeIt(normal,io_lib:fwrite("Move updated to ~p.~n",
             [Args]))
    end.

turn({Atom,Rule,Args})->
    case Atom of
        start->
            bs:update_belief({executing,{Rule,turn,Args}}),
            writer:writeIt(normal,io_lib:fwrite("Begin turn ~p.~n",
             [Args]));
        finalize->
            bs:update_belief({last_executed,{Rule,turn,Args}}),
            bs:update_belief({executing,{[],[],[]}}),
            writer:writeIt(normal,io_lib:fwrite("End turn ~p.~n",
             [Args]));
```

```
        update->
            bs:update_belief({last_executed,
            bs:get_belief(executing)}),
            bs:update_belief({executing,{Rule,turn,Args}}),
            writer:writeIt(normal,io_lib:fwrite("Turn updated to ~p.~n",
             [Args]))
    end.

close_gripper({_Atom,Rule,_Args})->
    bs:update_belief({last_executed,{Rule,close_gripper,[]}}),
    writer:writeIt(normal,io_lib:fwrite("Gripper closed.~n",[])).

open_gripper({_Atom,Rule,_Args})->
    bs:update_belief({last_executed,{Rule,open_gripper,[]}}),
    writer:writeIt(normal,io_lib:fwrite("Gripper opened.~n",[])).
```

4.2 Version 2

This version includes the communication between two agents. These agents exchange messages informing when they collect a bottle, so the collecting goal is mutual.

This version incorporates a modification in the BeliefStore module, but the Executor & Writer modules remain without changes. The TeleoR program of the second version is the following:

```
dir::= left | centre | right
thing::= bottle | drop | robot

percept
   gripper_open: (),
   holding : (),
   see : (thing, int),
   touching : (dir),
   over_drop : ()

durative
   move : (num),
   turn : (dir)

discrete
   open_gripper : (),
   close_gripper : ()

int _collected:=0
int _other_collected:=0

communicating_collect_bottles: (int, atom)
communicating_collect_bottles(Total, OthrAg){
   _collected + _other_collected >= Total ~> ()
   holding & goal(drop) while 3 ~> drop_and_leave(OthrAg)
   holding ~> get_to_drop
   true ~> get_bottle
}

drop_and_leave : (atom) ~>
drop_and_leave(OtherAg){
   gripper_open ~> leave_drop
   true ~> open_gripper ++
    update_and_communicate_count(OthrAg)
}

update_and_communicate_count: atom
update_and_communicate_count(OthrAg) ~>>
   _collected := _collected + 1;
   count(_collected) to OthrAg
```

```
leave_drop : () ~>
leave_drop {
   not see(drop,_) & not see(robot,_) ~> move(1.0)
   see(drop, _) ~> turn(left, 0.8)
   see(robot, _)  ~> turn(right, 0.8)
}

next_to : (thing)
next_to(Th) <= see(Th, Dist) & Dist < 15

get_to_drop:
get_to_drop(){
   over_drop  ~> ()
   next_to(drop) ~> move(1.0)
   see(drop, Dist) & Dist > 40 ~> move(2.0)
   true ~> turn(left, 1.0) for 2; move(1.0) for 2
}

get_bottle:
get_bottle(){
   holding ~> ()
   touching(centre) & gripper_open ~> close_gripper
   touching(_) & gripper_open ~> turn(left, 0.2)
   next_to(bottle) ~> open_gripper wait 2^3
   see(bottle, _) ~> move(1.5)
   true ~> turn(left,1.0) for 2.8; move(1.0) for 2
}

handle_message_(M, _):: M = count(Count) ~>>
 other_collected := Count;
handle_message_(_,_)  % Ignore any other message

go : (int, atom)
go(Num, OtherName) ~>> communicating_collect_bottles(Num,
 OtherAgent@localhost)
```

4.2.1 BeliefStore module

Since the TeleoR program varies from the first version, the *priority* belief also changes.

```
priority=>
   [{communicating_collect_bottles,[collected,
    holding_next_drop, holding, true,[]]},
   {drop_and_leave,[gripper_open,true,[]]},
   {leave_drop,[not_see,see_drop,see_robot,[]]},
   {get_to_drop,[over_drop, next_drop, see_drop, true,[]]},
   {get_bottle,[holding, touching_centre_gripper_open,
    touching_gripper_open, next_bottle,see_bottle, true,[]]},
```

```
    {[],[]}]}}.
```

4.2.2 TR module

According to the TeleoR module, it integrates new rules, procedures and the communication between agents.

The new implementation is the following:

```
-module(tr).

%% ============================================
%% API FUNCTIONS
%% ============================================
-export([init/4,
go/2,
communicating_collect_bottles/0,
move/1,
turn/1,
close_gripper/1,
open_gripper/1,
handle_message/0]).

%% ================================================
%% INTERNAL FUNCTIONS
%% ================================================

init(Num,OthrAg,Mode,File)->
    bs:start_link(),
    bs:add_belief({writer,{Mode,File}}),
    spawn(tr,go,[Num,OthrAg]).

go(Num,OthrAg)->
    bs:add_belief({total,Num}),
    bs:add_belief({othrAg,OthrAg}),
    communicating_collect_bottles().

communicating_collect_bottles()->
    executor:update_execution(communicating_collect_bottles),
    case {bs:get_belief(collected)+
     bs:get_belief(other_collected)>=bs:get_belief(total),
     bs:is_belief(holding) andalso next_to(drop),
     bs:is_belief(while_timer), bs:is_belief(holding)} of

         {true,_,_,_}->
            io:format("~p cans collected, task finished.~n",
              [bs:get_belief(total)]),
```

```
            writer:writeIt(normal, io_lib:fwrite("~p cans collected,
             task finished.~n", [bs:get_belief(total)]));
        {_,true,_,true}->
            drop_and_leave();
        {_,_,true,_}->
            drop_and_leave();
        {_,true,_,_} ->
            get_to_drop();
        {_,_,_,_} ->
            get_bottle()
    end.

drop_and_leave()->
    case bs:is_belief(while_timer) of
        true->
            ok;
        _->
            bs:add_belief({while_timer,[]}),
            executor:while_condition([{communicating_collect_bottles,
            holding_over_drop},3000])
    end,

    case bs:is_belief(gripper_open) of

        true ->
            leave_drop();
        _ ->
            executor:execute([{drop_and_leave,true},open_gripper,[]]),
            update_and_communicate_count(bs:get_belief(othrAg))
    end.

update_and_communicate_count(OthrAg)->
    bs:update_belief({collected,bs:get_belief(collected)+1}),
    OthrAg ! {count,bs:get_belief(collected)},
    communicating_collect_bottles().

leave_drop()->
    case  {bs:is_belief(see,drop),bs:is_belief(see,robot)} of

        {true,_} ->
            executor:execute_while([{leave_drop,see_drop},turn,
            {left,0.8}, true]),
            communicating_collect_bottles();
        {_,true} ->
            executor:execute_while([{leave_drop,see_robot},turn,
             {right,0.8},true]),
            communicating_collect_bottles();
        {_,_} ->
            executor:execute_while([{leave_drop,not_see},move,{1.0},
            true]),
            communicating_collect_bottles()
```

```
    end.

next_to(Th)->s
    case executor:compare_value(Th, bs:get_belief(see), minor, 15) of
        true->
            true;
        _->
            false
    end.

get_to_drop()->
    case {bs:is_belief(over_drop), next_to(drop), executor:compare_value(
     drop, bs:get_belief(see), major, 40)} of

        {true,_,_} ->
            communicating_collect_bottles();
        {_,true,_} ->
            executor:execute_while([{get_to_drop,next_drop},move,{1.0},
             true]),
            communicating_collect_bottles();
        {_,_,true}->
            executor:execute_while([{get_to_drop,see_drop},move,{2.0},
             true]),
            communicating_collect_bottles();
        {_,_,_} ->
            executor:execute_while([{get_to_drop,true},turn,{left,1.0},
             2000]),
            executor:execute_while([{get_to_drop,true},move,{1.0},
             2000]),
            communicating_collect_bottles()
    end.

get_bottle()->
    case {bs:is_belief(holding),executor:compare_value(touching,
     bs:get_bs(), equal, centre) andalsobs:is_belief(gripper_open),
     bs:is_belief(touching) andalso bs:is_belief(gripper_open),
     next_to(bottle), bs:is_belief(see,bottle)} of

        {true,_,_,_,_} ->
            communicating_collect_bottles();
        {_,true,_,_,_} ->
            executor:execute([{get_bottle,touching_centre_gripper_open},
             close_gripper,[]]),
            communicating_collect_bottles();
        {_,_,true,_,_} ->
            executor:execute_while([{get_bottle,touching_gripper_open},
             turn,{left,0.2},true]),
            communicating_collect_bottles();
        {_,_,_,true,_}->
            executor:execute([{get_bottle,next_bottle},open_gripper,
             []]),
```

```
            communicating_collect_bottles();
        {_,_,_,_,true} ->
            executor:execute_while([{get_bottle,see_bottle},move,{1.5},
             true]),
            communicating_collect_bottles();
        {_,_,_,_,_} ->
            executor:execute_while([{get_bottle,true},turn,{left,1.0},
             2800]),
            executor:execute_while([{get_bottle,true},move,{1.0},
             2000]),
            communicating_collect_bottles()
    end.

handle_message()->
    receive
        {count,Num}->
            bs:update_belief({other_collected,Num}),
            handle_message();
        {_,_}->
            handle_message()
    end.
```

This module includes a new function to initialize the system: *init/4*. This function set the number of bottles that needs to be collected to complete the task, the identifier of the other robot, the mode of writing (*normal* or *debug*) and the name of the file.

In addition, this version includes a function to determine the proximity to an object: *next_to/1*. Besides, it implements a function to catch the messages to update the number of bottles collect for the other robot in its BeliefStore. It ignores the rest of messages.

4.3 Version 3

This version adds the interaction between agents, with the aim of avoiding collisions. This process is carried out reducing the speed at which the robot moves, and sending a message to the other robot indicating that it has been seen.
The TeleoR program is shown below:

```
dir::= left | centre | right
thing::= bottle | drop | robot

percept
   gripper_open: (),
   holding : (),
   see : (thing, int),
   touching : (dir),
   over_drop : ()

durative
   move : (num),
   turn : (dir)

belief
   seen: ()

discrete
   open_gripper : (),
   close_gripper : ()

int _collected:=0
int _other_collected:=0

communicating_collect_bottles: (int, atom)
communicating_collect_bottles(Total, OthrAg){
   _collected + _other_collected >= Total ~> ()
   holding & next_to(drop) while 3 ~>
    drop_and_leave(OthrAg)
   holding ~> get_to_drop(OthrAg)
   true ~> get_bottle(OthrAg)
}

drop_and_leave : (atom) ~>
drop_and_leave(OtherAg){
   gripper_open ~> leave_drop
   true ~> open_gripper ++
    update_and_communicate_count(OthrAg)
}

update_and_communicate_count: atom
update_and_communicate_count(OthrAg) ~>>
   _collected := _collected + 1;
   count(_collected) to OthrAg
```

```
leave_drop : () ~>
leave_drop {
   not see(drop,_) & not see(robot,_) ~> move(1.0)
   see(drop, _) ~> turn(left, 0.8)
   see(robot, _)  ~> turn(right, 0.8)
}

next_to : (thing)
next_to(Th) <= see(Th, Dist) & Dist < 15

get_to_drop: atom
get_to_drop(OthrAg){
   over_drop  ~> ()
   seen ~> move(0.2)
   next_to(drop) ~> move(1.0)
   see(robot, Dist) & Dist < 30 ~> seen to OthrAg
   see(drop, Dist) & Dist > 40 ~> move(2.0)
   true ~> turn(left, 1.0) for 2; move(1.0) for 2
}

get_bottle: atom
get_bottle(OthrAg){
   holding ~> ()
   touching(centre) & gripper_open ~> close_gripper
   touching(_) & gripper_open ~> turn(left, 0.2)
   seen ~> move(0.2)
   see(robot, Dist) & Dist < 30 ~> seen to OthrAg
   next_to(bottle) ~> open_gripper
   see(bottle, _) ~> move(1.5)
   true ~> turn(left,1.0) for 2.8; move(1.0) for 2
}

handle_message_(M, _):: M = count(Count) ~>>
 other_collected := Count;
handle_message_(M, _):: M = seen ~>> remember seen for 5;
handle_message_(_,_)  % Ignore any other message

go : (int, atom)
go(Num, OtherName) ~>> communicating_collect_bottles(Num,
 OtherAgent@localhost)
```

4.3.1 BeliefStore module

In the same way that in the second version, the changes in the TR module imply a modification in the *priority* belief:

```
priority=>
   [{communicating_collect_bottles,[collected,holding_next_drop,holding,
    true,[]]},
```

```
    {drop_and_leave,[gripper_open,true,[]]},
    {leave_drop,[not_see,see_drop,see_robot,[]]},
    {get_to_drop,[over_drop,seen,next_drop,see_robot,see_drop,true,[]]},
    {get_bottle,[holding,touching_centre_gripper_open,
     touching_gripper_open,seen,see_robot,next_bottle,see_bottle,true,[]]},
    {[],[]}]}}.
```

4.3.2 TR module

This module adds, compared to the previous one, a new functionality to handle the messages related to when a robot is seen. It also includes the new protocol to control and avoid collisions between robots.

```
-module(tr).

%% ============================================
%% API FUNCTIONS
%% ============================================
-export([init/4,
go/2,
communicating_collect_bottles/0,
move/1,
turn/1,
leave_drop/0,
close_gripper/1,
open_gripper/1,
send_message/1,
handle_message/0]).

%% ============================================
%% INTERNAL FUNCTIONS
%% ============================================
init(Num,OthrAg,Mode,File)->
    bs:start_link(),
    bs:add_belief({writer,{Mode,File}}),
    spawn(tr,go,[Num,OthrAg]).

go(Num,OthrAg)->
    bs:add_belief({total,Num}),
    bs:add_belief({othrAg,OthrAg}),
    communicating_collect_bottles().

communicating_collect_bottles()->
    case {bs:get_belief(collected)+
     bs:get_belief(other_collected)>=bs:get_belief(total),
     bs:is_belief(holding) andalso next_to(drop),
     bs:is_belief(while_timer), bs:is_belief(holding)} of
```

```
        {true,_,_,_}->
           io:format("~p cans collected, task finished.~n",
           [bs:get_belief(total)]),
           writer:writeIt(normal, io_lib:fwrite("~p cans collected,
            task finished.~n", [bs:get_belief(total)]));
        {_,true,_,_}->
           drop_and_leave();
        {_,_,true,_}->
           drop_and_leave();
        {_,_,_,true} ->
           get_to_drop();
        {_,_,_,_} ->
           get_bottle()
   end.

drop_and_leave()->
   case bs:is_belief(while_timer) of

        true->
           ok;
        _->
           bs:add_belief({while_timer,[]}),
           executor:while_condition([{communicating_collect_bottles,
            holding_over_drop},3000])
   end,

   case bs:is_belief(gripper_open) of

        true ->
           leave_drop();
        _ ->
           executor:execute([{drop_and_leave,true},open_gripper,[]]),
           update_and_communicate_count({drop_and_leave,true},
           bs:get_belief(othrAg))
   end.

update_and_communicate_count(Rule,OthrAg)->
   bs:update_belief({collected,bs:get_belief(collected)+1}),
   executor:execute([Rule,send_message,{count,bs:get_belief(collected)}]),
   communicating_collect_bottles().

leave_drop()->
   case  {bs:is_belief(see,drop),bs:is_belief(see,robot)} of

        {true,_} ->
           executor:execute_while([{leave_drop,see_drop},turn,{left,
           0.8},true]),
           communicating_collect_bottles();
        {_,true} ->
           executor:execute_while([{leave_drop,see_robot},turn,{right,
           0.8},true]),
```

```
            communicating_collect_bottles();
        {_,_} ->
            executor:execute_while([{leave_drop,not_see},move,{1.0},
             true]),
            communicating_collect_bottles()
    end.

next_to(Th)->
    case executor:compare_value(Th, see, minor, 15) of
        true->
            true;
        _->
            false
    end.

get_to_drop()->
    case {bs:is_belief(over_drop), bs:is_belief(seen), next_to(drop),
     executor:compare_value(robot,see,minor,30),executor:compare_value(
     drop,see,major,40)} of

        {true,_,_,_,_} ->
            communicating_collect_bottles();
        {_,true,_,_,_} ->
            executor:execute_while([{get_to_drop,seen},move,{0.2},true]),
            communicating_collect_bottles();
        {_,_,true,_,_} ->
            executor:execute_while([{get_to_drop,next_drop},move,{1.0},
             true]),
            communicating_collect_bottles();
        {_,_,_,true,_} ->
            executor:execute([{get_to_drop,see_robot},send_message,
             {seen,self()}]),
            communicating_collect_bottles();
        {_,_,_,_,true}->
            executor:execute_while([{get_to_drop,see_drop},move,{2.0},
            true]),
            communicating_collect_bottles();
        {_,_,_,_,_} ->
            executor:execute_while([{get_to_drop,true},turn,{left,1.0},
             2000]),
            executor:execute_while([{get_to_drop,true},move,{1.0},
             2000]),
            communicating_collect_bottles()
    end.

get_bottle()->
    case {bs:is_belief(holding),executor:compare_value(touching,all,
     equal,centre) andalso bs:is_belief(gripper_open),
     bs:is_belief(touching) andalso bs:is_belief(gripper_open),
     bs:is_belief(seen),executor:compare_value(robot,see,minor,30),
     next_to(bottle), bs:is_belief(see,bottle)} of
```

```
        {true,_,_,_,_,_,_} ->
            communicating_collect_bottles();
        {_,true,_,_,_,_,_} ->
            executor:execute([{get_bottle,touching_centre_gripper_open},
             close_gripper,[]]),
            communicating_collect_bottles();
        {_,_,true,_,_,_,_} ->
            executor:execute_while([{get_bottle,touching_gripper_open},
            turn,{left,0.2},true]),
            communicating_collect_bottles();
        {_,_,_,true,_,_,_} ->
            executor:execute_while([{get_bottle,seen},move,{0.2},true]),
            communicating_collect_bottles();
        {_,_,_,_,true,_,_}->
            executor:execute([{get_bottle,see_robot},send_message,
             {seen,self()}]),
            communicating_collect_bottles();
        {_,_,_,_,_,true,_}->
            executor:execute([{get_bottle,next_bottle},open_gripper,[]]),
            communicating_collect_bottles();
        {_,_,_,_,_,_,true} ->
            executor:execute_while([{get_bottle,see_bottle},move,{1.5},
            true]),
            communicating_collect_bottles();
        {_,_,_,_,_,_,_} ->
            executor:execute_while([{get_bottle,true},turn,{left,1.0},
            2800]),
            executor:execute_while([{get_bottle,true},move,{1.0},2000]),
            communicating_collect_bottles()
    end.

handle_message()->
    register(robotPID,self()),
    receive
        {count,Num}->
            bs:update_belief({other_collected,Num}),
            handle_message();
        {seen,PID}->
            executor:remember({seen,PID},5000),
            handle_message();
        {_,_}->
            handle_message()
    end.

send_message({_,Rule,Msg})->
    bs:update_belief({last_executed,{Rule,send_message,Msg}}),
    bs:update_belief({executing,{[],[],[]}}),
    bs:get_belief(othrAg) ! Msg.
```

4.4 Overall performance

Taking the final version as starting point, the general functioning of the system can be represented with the next scheme: The first step is to spawn the process

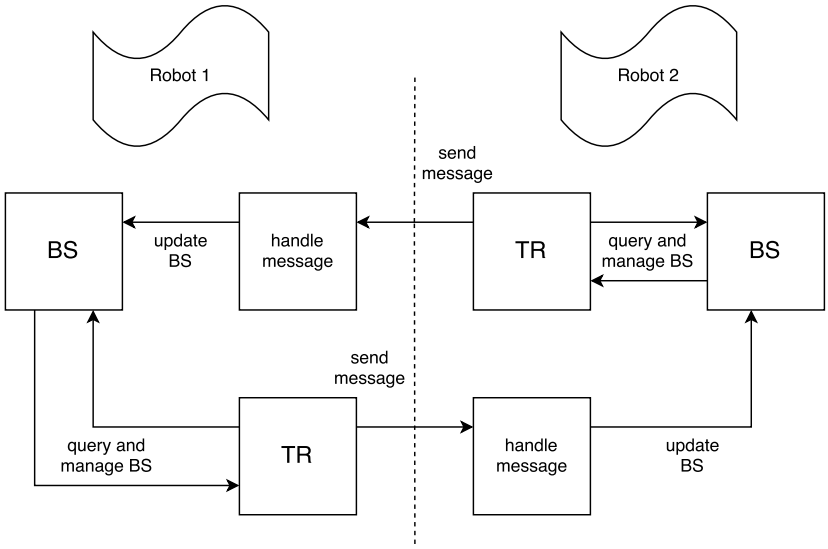

Figure 4.3: Overall performance diagram

to handle the messages, in order to store the PID (from the other robot) in the BeliefStore. Therefore, it allows the exchange of messages between the TR process and the handle message process of the other robot. Moreover, there is other process for the BS. In this diagram the processes spawned by the TR to execute the actions have been ignored. However, these processes have the same connection with the BS (they can query for information and update the BS with new data).

Furthermore, this is an example of the file obtained after running this implementation in *debug* mode:

```
15 Jul 2017 20:10:57.937000 Instruction execute_while
{{get_bottle,true},turn,{left,1.0}}.
15 Jul 2017 20:10:57.938000 Executing {[],[],[]}.
15 Jul 2017 20:10:57.939000 Last executed
{{get_bottle,true},move,{1.0}}.
15 Jul 2017 20:10:57.939000 To execute
{{get_bottle,true},turn,{left,1.0}}.

15 Jul 2017 20:10:57.940000 Instruction execute
```

execute{{get_bottle,true},turn,{left,1.0}}.
15 Jul 2017 20:10:57.941000 Executing {[],[],[]}.
15 Jul 2017 20:10:57.941000 Last executed
{{get_bottle,true},move,{1.0}}.
15 Jul 2017 20:10:57.942000 To execute
{{get_bottle,true},turn,{left,1.0}}.

15 Jul 2017 20:10:57.943000 Begin turn {left,1.0}.
15 Jul 2017 20:10:57.944000 Timer
{{44269000,#Ref<0.0.2883585.87739>},{get_bottle,true},turn,{left,1.0}}.

15 Jul 2017 20:10:58.893000 Instruction execute
priority{{get_bottle,see_bottle},move,{1.5}}.
15 Jul 2017 20:10:58.895000 Executing
{{get_bottle,true},turn,{left,1.0}}.
15 Jul 2017 20:10:58.896000 Last executed
{{get_bottle,true},move,{1.0}}.
15 Jul 2017 20:10:58.897000 To execute
{{get_bottle,see_bottle},move,{1.5}}.

15 Jul 2017 20:10:58.898000 Kill {44269000,#Ref<0.0.2883585.87739>}.
15 Jul 2017 20:10:58.899000 End turn {left,1.0}.
15 Jul 2017 20:10:58.901000 Begin move {1.5}.

15 Jul 2017 20:11:42.390000 Instruction execute update
{{get_bottle,seen},move,{0.2}}.
15 Jul 2017 20:11:42.391000 Executing
{{get_bottle,see_bottle},move,{1.5}}.
15 Jul 2017 20:11:42.392000 Last executed
{{get_bottle,true},turn,{left,1.0}}.
15 Jul 2017 20:11:42.393000 To execute
{{get_bottle,seen},move,{0.2}}.

15 Jul 2017 20:11:42.396000 Move updated to {0.2}.

15 Jul 2017 20:11:47.376000 Instruction execute update
{{get_bottle,see_bottle},move,{1.5}}.
15 Jul 2017 20:11:47.377000 Executing
{{get_bottle,seen},move,{0.2}}.
15 Jul 2017 20:11:47.377000 Last executed
{{get_bottle,seen},move,{0.2}}.
15 Jul 2017 20:11:47.378000 To execute
{{get_bottle,see_bottle},move,{1.5}}.

15 Jul 2017 20:11:47.379000 Move updated to {1.5}.

The robot starts turning left for 2 seconds, but while it is performing the action, it receives the request to {move,1.5}. This is because it has seen a bottle. Thus, it finishes the 2 seconds timer related with {turn,1.0} and executes the action {move,1.5}. Then, it updates its action to {move,0.2}, as a consequence of being seen by the other robot. Finally, after 5 seconds (that is the time the seen belief is remembered in the BS), it forgets the belief seen. Besides, it executes again the {move,1.5}, so it updates {move,0.2} to move,1.5}.

5. Case Study. Raspberry PI Implementation

This chapter describes the implementation of the TR program, which is explained in chapter 4, in a Raspberry PI.

5.1 Version 1

The first example, which is explained in section 4.1, includes the main functionalities of the robot to collect the bottles: get a bottle, go to the drop, drop the bottle, and leave the drop. The simulation of the agent's environment is carry out with the use of LEDs and switches, as it is explained in section 5.1.3. Thus, with this implementation it is possible to observe when the robot is turning, when it is moving, when it opens/closes the gripper and when it collects a bottle.

5.1.1 Erlang installation

The first step to be able to execute the first example is install Erlang in the Raspberry PI. This installation can take around an hour, the commands (*Elixir on the Raspberry Pi - Blinking an LED*, 2015) that need to be executed are:

```
# Download, compile, and install Erlang
$ apt-get install wget libssl-dev ncurses-dev m4 unixodbc-dev erlang-dev
$ wget http://www.erlang.org/download/otp_src_18.1.tar.gz
$ tar -xzvf otp_src_18.1.tar.gz
$ cd otp_src_18.1/
$ export ERL_TOP=`pwd`
$ ./configure
$ make
$ make install
```

Once the installation is finished, it is possible to execute the Erlang shell doing the following:

```
pi@raspberrypi ~ $ sudo erl
```

And in the Erlang shell the user only has to compile the files and run the example:

```
Erlang/OTP 19 [erts-8.3] [source] [smp:4:4] [async-threads:10]
 [kernel-poll:false]
Eshell V8.3  (abort with ^G)
1>c(bs).
{ok,bs}
2>c(executor).
{ok,executor}
3>c(writer).
{ok,writer}
4>c(gpio).
{ok,gpio}
5>c(tr).
{ok,tr}
6>tr:go(debug,"test_v1",5).
```

5.1.2 TR modification

With the aim of running the first version implementation in the Raspberry PI some changes are needed. It is necessary to modify the *move*, *turn*, *open_gripper*, *close_gripper*, and the *drop_and_leave* functions to manage some percepts and to write in the GPIO. In addition, it uses a new module to control de GPIO developed by Paolo Oliveira (Oliveira, 2015).

The functions updated are the followings:

```
move({Atom,Rule,Args})->

    case Atom of
        start->
            bs:update_belief({executing,{Rule,move,Args}}),
            gpio:write(bs:get_belief(move), 1);
        finalize->
            bs:update_belief({last_executed,{Rule,move,Args}}),
            bs:update_belief({executing,{[],[],[]}}),
            gpio:write(bs:get_belief(move), 0);
        update->
            bs:update_belief({last_executed,bs:get_belief(executing)}),
            bs:update_belief({executing,{Rule,move,Args}}),
            gpio:write(bs:get_belief(move), 1)
    end.


turn({Atom,Rule,Args})->

    case Atom of
        start->
            bs:update_belief({executing,{Rule,turn,Args}}),
            gpio:write(bs:get_belief(turn), 1);
        finalize->
            bs:update_belief({last_executed,{Rule,turn,Args}}),
            bs:update_belief({executing,{[],[],[]}}),
            gpio:write(bs:get_belief(turn), 0);
        update->
            bs:update_belief({last_executed,bs:get_belief(executing)}),
            bs:update_belief({executing,{Rule,turn,Args}}),
            gpio:write(bs:get_belief(turn), 1)
    end.

close_gripper({_Atom,Rule,_Args})->
    bs:update_belief({last_executed,{Rule,close_gripper,[]}}),
    bs:remove_belief(gripper_open),
    bs:add_belief({holding,[]}),
    gpio:write(bs:get_belief(gripper), 0),
    timer:sleep(1000).
```

```
open_gripper({_Atom,Rule,_Args})->
    bs:update_belief({last_executed,{Rule,open_gripper,[]}}),
    bs:add_belief({gripper_open,[]}),
    bs:remove_belief(holding),
    gpio:write(bs:get_belief(gripper), 1),
    timer:sleep(1000).

drop_and_leave()->
    case bs:is_belief(while_timer) of
          true->
             ok;
          _->
             bs:add_belief({while_timer,[]}),
             executor:while_condition([{collect_bottles,holding_over_drop},
              3000])
    end,

    case bs:is_belief(gripper_open) of

          true ->
             leave_drop();
          _ ->
             executor:execute([{drop_and_leave,true},open_gripper,[]]),
             bs:update_belief({collected,bs:get_belief(collected)+1}),
             gpio:write(bs:get_belief(buzzer), 1),
         timer:sleep(1500),
             gpio:write(bs:get_belief(buzzer), 0),
             collect_bottles()
    end.
```

Each durative function turn on the LED in its *start* and *update* and turn off the LED in its *finalize*. While the discrete actions only perform one action (turn on/off) and waits 1 second before executing the next action, in order to be able to appreciate the status changes of the LEDs. In addition, the *gripper_open* and *holding* percepts are managed in the *open_gripper* and *close_gripper* functions, in order to avoid errors, because they are percepts that are closely related.

To use the GPIO module, it is necessary to initialize the Raspberry PI pins at the beginning. Moreover, the reference of the output pins are stored in the BeliefStore. And the input pins are controlled with a handler function, so as to determine if there are changes of the percepts stored in the BeliefStore. Thus, the *go* function is also modified:

```
go(Mode,File,Num)->
    bs:start_link(),
    bs:add_belief({writer,{Mode,File}}),
    bs:add_belief({goal,Num}),

    L0 = gpio:init(17, out),
```

```
    bs:add_belief({move,L0}),
    L1 = gpio:init(18, out),
    bs:add_belief({turn,L1}),
    L2 = gpio:init(23, out),
    bs:add_belief({gripper,L2}),
    L3 = gpio:init(24, in),
    spawn(tr,handler,[L3,touching]),
    L4 = gpio:init(10, out),
    bs:add_belief({buzzer,L4}),
    L5 = gpio:init(9, in),
    spawn(tr,handler,[L5,see,drop]),
    L6 = gpio:init(25, in),
    spawn(tr,handler,[L6,see,bottle]),
    L7 = gpio:init(11, in),
    spawn(tr,handler,[L7,over_drop]),
    collect_bottles().

handler(Ref,Belief)->
    case gpio:read(Ref) of
        "0" ->
            case bs:is_belief(Belief) of
                true->
                    bs:remove_belief(Belief),
                    handler(Ref,Belief);
                false->
                    handler(Ref,Belief)
            end;
        "1" ->
            case bs:is_belief(Belief) of
                true->
                    handler(Ref,Belief);
                false->
                    bs:add_belief({Belief,[]}),
                    handler(Ref,Belief)
            end
    end.

handler(Ref,Key1,Key2)->
    case gpio:read(Ref) of
        "0" ->
            case bs:is_belief(Key1,Key2) of
                true->
                    bs:remove_one_belief({Key1,Key2}),
                    handler(Ref,Key1,Key2);
                false->
                    handler(Ref,Key1,Key2)
            end;
        "1" ->
            case bs:is_belief(Key1,Key2) of
                true->
                    handler(Ref,Key1,Key2);
```

```
                false->
                    bs:add_belief({Key1,{Key2,[]}}),
                    handler(Ref,Key1,Key2)
            end
    end.
```

Since there are percepts that can have several values, there are two types of *handler* function. The first one for percepts of a single value (e.g. touching=>[]), and the second for percepts of several values (e.g. see=>{bottle,drop,robot}).

Furthermore, the GPIO module used is the following:

```
%% @author Paolo Oliveira <paolo@fisica.ufc.br>
%% @copyright 2015-2016 Paolo Oliveira (license MIT)
%% @version 1.0.0
%% @doc
%% A simple, pure erlang implementation of a module for <b>Raspberry Pi's
%%General Purpose Input/Output</b> (GPIO), using the standard Linux kernel
%% interface for user-space, sysfs, available at <b>/sys/class/gpio/</b>.
%% @end

-module(gpio).
-export([init/1, init/2, handler/2, read/1, write/2, stop/1]).
-author('Paolo Oliveira <paolo@fisica.ufc.br>').

%% API

% @doc: Initialize a Pin as input or output.
init(Pin, Direction) ->
    Ref = configure(Pin, Direction),
    Pid = spawn(?MODULE, handler, [Ref, Pin]),
    Pid.

% @doc: A shortcut to initialize a Pin as output.
init(Pin) ->
    init(Pin, out).

% @doc: Stop using and release the Pin referenced as file descriptor Ref.
stop(Ref) ->
    Ref ! stop,
    ok.

% @doc: Read from an initialized Pin referenced as the file descriptor
%%Ref.
read(Ref) ->
    Ref ! {recv, self()},
    receive
        Msg ->
            Msg
    end.
```

```
%% @doc: Write value Val to an initialized Pin referenced as the file
%%descriptor Ref.
write(Ref, Val) ->
    Ref ! {send, Val},
    ok.

%%% Internals

configure(Pin, Direction) ->
    DirectionFile = "/sys/class/gpio/gpio" ++ integer_to_list(Pin) ++
     "/direction",

% Export the GPIO pin
    {ok, RefExport} = file:open("/sys/class/gpio/export", [write]),
    file:write(RefExport, integer_to_list(Pin)),
    file:close(RefExport),

% It can take a moment for the GPIO pin file to be created.
    case filelib:is_file(DirectionFile) of
         true -> ok;
         false -> receive after 1000 -> ok end
    end,

    {ok, RefDirection} = file:open(DirectionFile, [write]),
    case Direction of
         in -> file:write(RefDirection, "in");
         out -> file:write(RefDirection, "out")
    end,
    file:close(RefDirection),
    {ok, RefVal} = file:open("/sys/class/gpio/gpio" ++
     integer_to_list(Pin) ++ "/value", [read, write]),
    RefVal.

release(Pin) ->
    {ok, RefUnexport} = file:open("/sys/class/gpio/unexport", [write]),
    file:write(RefUnexport, integer_to_list(Pin)),
    file:close(RefUnexport).

% @doc: Message passing interface, should not be used directly, it is
%%present for debugging purpose.
handler(Ref, Pin) ->
    receive
         {send, Val} ->
            file:position(Ref, 0),
            file:write(Ref, integer_to_list(Val)),
            handler(Ref, Pin);
         {recv, From} ->
            file:position(Ref, 0),
            {ok, Data} = file:read(Ref, 1),
            From ! Data,
```

```
            handler(Ref, Pin);
        stop ->
            file:close(Ref),
            release(Pin),
            ok
    end.

%% End of Module.
```

5.1.3 Hardware implementation

As it has been explained in the previous sections, the environment changes are simulated with the use of switches and LEDs and the read/write with the GPIO module.

The Raspberry PI pins used are the represented in the Figure 5.1.

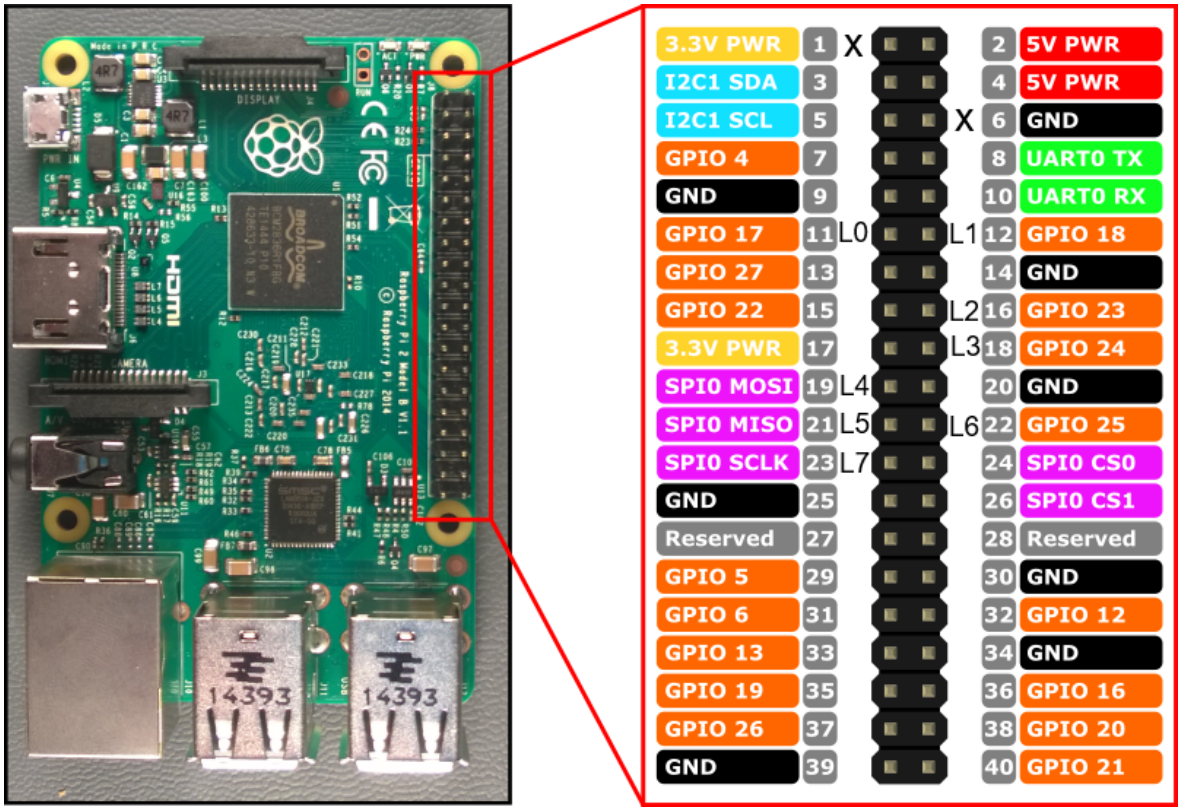

Figure 5.1: Raspberry PI 3 used pins scheme

Where:

- L0 represents the LED associated with the *move* action
- L1 represents the LED associated with the *turn* action
- L2 represents the LED associated with the gripper, which is turned on when the gripper is open
- L3 represents the switch associated with the *touching* percept
- L4 represents the LED associated with the collected bottle, which is turned on when a bottle is collected
- L5 represents the switch associated with the *{see,drop}* percept
- L6 represents the switch associated with the *{see,bottle}* percept
- L7 represents the switch associated with the *over_drop* percept

Therefore, the distribution of the LEDs and switches observed in the Figure 5.2 is described:

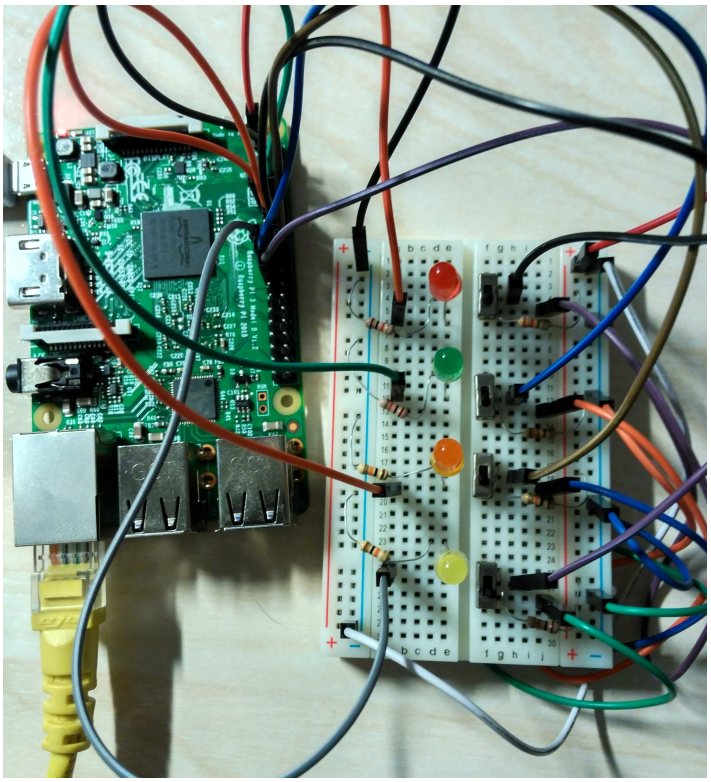

Figure 5.2: Example 1 implementation in Raspberry 3

The red LED represents the *move* action, the green LED represents the *turn* action, the orange LED represents the gripper and the yellow LED represents the collect bottle action. On the other side, the switches, from the top to the bottom, represents the *touching*, *{see,drop}*, *{see,bottle}*, and *over_drop* percepts. The default position (switch turned down), represents a '0' (switch off).

Some different configurations of the percepts are described here below:

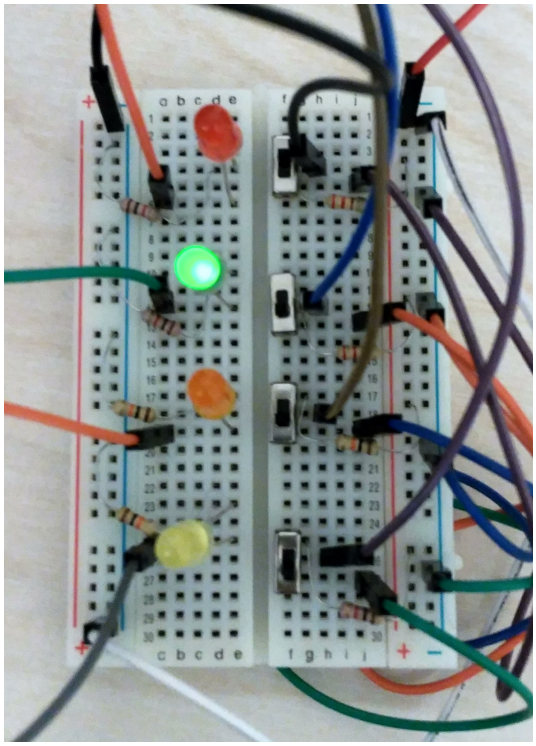

Figure 5.3: Get to bottle → Turning

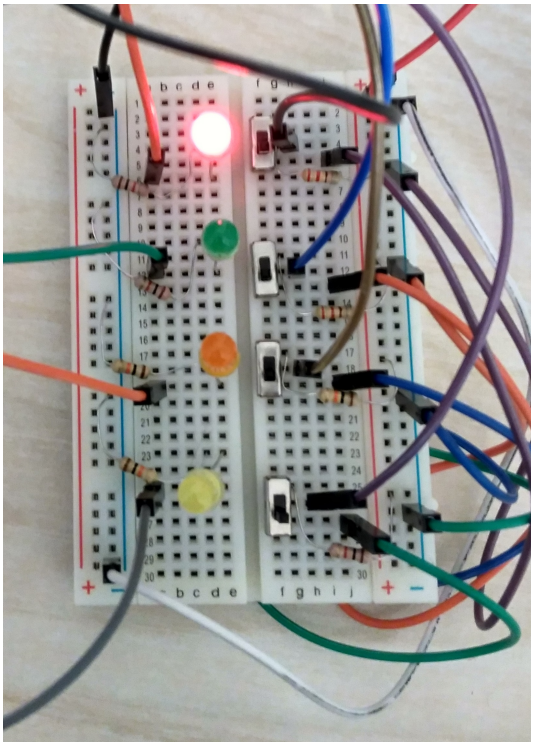

Figure 5.4: Get to bottle → Moving

These two figures, Figure 5.3 & Figure 5.4, represents the default status, where there are no percepts in the BeliefStore, and the robot is looking for a bottle.

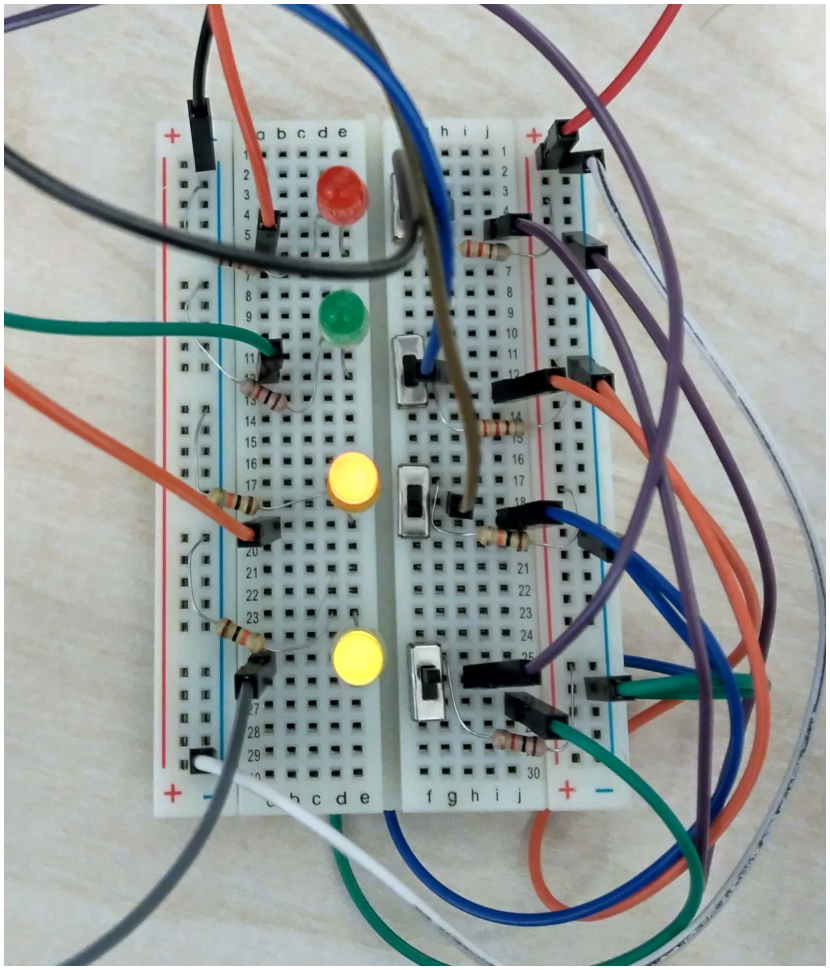

Figure 5.5: Drop and leave

On the other hand, the Figure 5.5, represent the moment when the robot opens the gripper to drop the bottle and the BeliefStore updates the *collected* belief. In this case, the *touching* percept had been switched on, and also the *over_drop* percept.

6. Interpretation Architecture

In this chapter the architecture of translation from TeleoR to Erlang is described. The main goal is to design a process where this translation can be performed systematically. First of all, the elements that remain unchanged are explained, and then the procedures to translate the TR expressions.

6.1 Permanent elements

During the translation procedure, there are elements that do not change from one interpretation to other. These elements are the executor module, explained in the section 4.1.2, and the writer module, explained in the section 4.1.3.

The structure of the BeliefStore module, which has been explained in the section 4.1.1, also remains invariable, only a modification in the *init/0* function is needed. The permanent part of the *init/0* function is the following:

```
init([])->
   {ok, #{last_executed=>
            {[],[],[]},
         executing=>
            {[],[],[]},
         to_execute=>
            {[],[],[]},
         timer=>
            [],
         priority=>
            [{[],[]}]}}.
```

The *priority* belief is fulfill with the goal/subgoal structure of the TR that is going to be translated. Each list of subgoals needs to finish with an empty subgoal ([]), in order to be able to compare goals with a higher level. Moreover, it has to finish with an empty {goal,subgoal}={[],[]}.

Regarding to the *priority/2* and the *get_priority/2* functions of the executor module, and the *is_belief/2* and the *remove_one/1* functions of the BeliefStore module, it is necessary an extrapolation when the TR structure is not composed of two levels: goal + subgoal. For instance, it could be composed of three levels: goal + subgoal + sub-subgoal.

6.2 Interpretation procedures

Firstly, focusing in the elements that can be found in the preamble of the TeleoR program, the translation concepts are the followings:

- Enums: The use of enums is going to be implemented with atoms inside the functions. A previous declaration is not needed.
- Percepts: Since the percepts are added to the BS through sensors, it is not necessary to declare them, so they can be ignored.
- Beliefs: The beliefs are manage in the BS through the TR functions.
- Variables: The variables are initialize in the *init/0* function from the BS. And they are managed from the TR functions.
- Durative actions: These actions are executed with the *execute_while/1* function from the Executor module.
- Discrete actions: These actions are executed with the *execute/1* function from the Executor module.

Going into detail, the interpretation procedures are described below.

Each function is implemented similarly to its TeleoR implementation. However, there are some differences:

- The evaluation of each condition is executed with the pattern matching mechanism. Within a *case...of structure* all the conditions are evaluated. Thus, it checks the first pattern which matches with the evaluation of conditions, and executes its actions.
- In the TeleoR functions there are **true** cases, that are executed when the rest of conditions are not true. This is implemented in Erlang with the "default" pattern: {_,_,_}, where all the conditions are caught with underscores (so its value is ignored, and it always matches).
- At the end of each final action, the main function is called, in order to re-evaluate all the conditions from the beginning. With the main functions is meant the function that encapsulates all the goals. Moreover, when a condition has not an action, it is represented with -> *()* in TeleoR, in Erlang only has to call the main function.

According to the evaluation of conditions, the existence of a percept/belief in the BS is checked with the *bs:is_belief*, which returns true if this is the case. There is also a function to check if a percept/belief has a certain value: *compare_value/4*,

which returns true if the condition is true. This last function allows to compare if the percept/belief is minor, minor or equal, equal, major or equal, and major of a certain value.

On the other hand, the management of variables is carried out with the *bs:get_belief/1* & *bs:update_belief/1* functions. Nevertheless, if the variable has not been initialize in the BS (in the *init/0* function) it has to be added with the *add_belief/1* function. But it is highly recommended to add the variables before running the TR program, with the aim of avoiding errors and exceptions.

The execution of actions is carried out with the *executor:execute_while/1* and the *executor:execute/1* functions, as it has been described at the beginning of this section. It is important to clarify the structure of the argument that these functions take:

```
execute_while([Rule,Fun,Args,true/Time])
execute([Rule,Fun,Args])
```

As already explained in the chapter 4, the *Rule* parameter represents the {goal,subgoal} of the action that is going to be executed, and the *Fun* and *Args* parameters represent the name and arguments of the function. In addition, the *execute_while* function includes other parameter, which can value *true* or *Time*, to determine the duration of the action. The *Time* parameter is expressed in milliseconds.

Limitation

This implementation presents a limitation, as it has been exposed, it only allows two levels of nesting. However, it can be scalable modifying the structure of the functions that calculate the priority and the functions which manage the beliefs in the BS. Therefore, when the level of nesting required is significant, this modification it is not easily affordable.

6.3 Interpretation algorithm

Recapitulating the translation concepts described in the section 6.1 and section 6.2, it is possible to define an algorithm to carry out the interpretation:

6.3.1 First step: Generate invariable modules

The first step is to generate the **Writer module**, the **Executor module** and the **BeliefStore module**. These modules does not change from an interpretation to other.

6.3.2 Second step: Analyze the TeleoR preamble

The second step is to analyze the preamble of the TR program, in order to **add** the **variables to the BeliefStore**.

For each variable, a new entrance is added to the *init* function of the BS:

```
init([])->
   {ok, #{...
          variable_A=>0,
          ...
          }}
```

From the preamble, it is also important to know if a function is durative or discrete, this information is used in further steps.

6.3.3 Third step: Create main structure

The third step consists in create the main structure of the TR program, using the goals.

For each task, it generates a new function:

```
Task1 : () ~>
Task1{
        Rule 1
          .
          .
          .
        Rule N
}
```

Figure 6.1: TeleoR: task

→

```
task1()->

        Rule 1
          .
          .
          .
        Rule N.
```

Figure 6.2: Erlang: task

For each rule, it generates a new entrance in a **case...of** structure:

```
Task1 : () ~>
Task1{
   Condition 1 ~> Action 1
             .
             .
             .
   Condition N-1 ~> Action N-1
   true ~> Action N
}
```

→

```
task1()->
 case {Cond 1,..., Cond N-1} of
   {true,_,...,_}-> Action 1;
                  .
                  .
                  .
   {_,_,...,true}-> Action N-1;
   {_,_,...,_}-> Action N
 end.
```

Figure 6.3: TeleoR: rule

Figure 6.4: Erlang: rule

For each condition, if it is a:

- Percept or belief: The Erlang condition check if the percept/belief exists in the BS, using the function *bs:is_belief*.

```
Task1 : () ~>
Task1{

   Percept 1 ~> Action 1
             .
             .
             .
   Belief N-1 ~> Action N-1
   true ~> Action N
}
```

→

```
task1()->
 case {bs:is_belief(Percp 1),...,
  bs:is_belief(Belief N-1} of
   {true,_,...,_}-> Action 1;
                  .
                  .
                  .
   {_,_,...,true}-> Action N-1;
   {_,_,...,_}-> Action N
 end.
```

Figure 6.5: TeleoR: percept/belief

Figure 6.6: Erlang: percept/belief

- Condition which compares a variable/belief/percept with a certain value: This is implemented with the use of the function *executor:compare_value(Key, KeyList, Mode, Value)*, where *Key* is the belief/percept that is going to be compared, *KeyList* is the list where the belief/percept is stored. It can value "all" if it is a general belief/percept, or other value, such as *see*, if it belongs to other belief/percept. For variables, the function used is *bs:get_belief*.

```
variable 1 + variable 2 < A ~>
       Action 1

percept 1 (C,D) & D < F ~>
       Action 2

true ~> Action 3
```

Figure 6.7: TeleoR: condition

```
case {bs:get_belief(var 1)+
     bs:get_belief(var 2) < A,
     executor:compare_value(C,
     percept 1,minor,F)} of
     {true,_}-> Action 1;
     {_,true}-> Action 2;
     {_,_}-> Action 3
end.
```

Figure 6.8: Erlang: condition

6.3.4 Fourth step: Define actions

The fourth steps is to define the calls for the discrete and durative actions.

For each discrete action:

```
executor:execute([Rule,Fun,Args]),
```

For each durative action:

```
executor:execute_while([Rule,Fun,Args,Time]),
```

The value of *Time* can be set to 'true' if it the action is executed while its left condition is true.

Moreover, at the end of the final action of each condition, the main task is called. This allows to be able to execute again all the conditions, to decide which action should be fired.

```
executor:execute_while([Rule,Fun,Args,Time]),
maintask().
```

6.3.5 Fifth step: Data management

The fifth step consists in the update of variables/beliefs of the BeliefStore. This process is carry out with the function *bs:update_belief*.

```
variable 1 := variable 1 + 1
```

Figure 6.9: TeleoR: variable update

```
%bs:update_belief({Belief,Value})
bs:update_belief({variable 1,
   variable 1 + 1})
```

Figure 6.10: Erlang: variable update

6.3.6 Sixth step: While conditions

Due to the existence of while structures in TeleoR, the evaluation of conditions experiences some changes. For each condition which includes a while structure, there is a new condition that checks if the *while_timer* exists in the BS. As a consequence, there are two possibilities for executing the action:

- The condition that derives the action is true.
- The timer associated to the while structure is active in the BS.

To implement this procedure, this code is implemented at the beginning of the action, in order to start the timer if it is not already in the BS:

```
case bs:is_belief(while_timer) of
  true->
       ok;
  _->
       bs:add_belief({while_timer,[]}),
       executor:while_condition([Rule,Time])
end,
```

Furthermore, it is also necessary to add a new condition in the evaluation of conditions in the execution of goals. The code is shown below:

```
function{
  A_1->
       action_1

  A_2 while X->
       action_2

  A_3 ->
       action_3

  true ->
       action_4
}
```

Figure 6.11: TeleoR: while condition

→

```
case {A_1,A_2,bs:is_belief(while_timer),
 A_3} of
  {true,_,_,_}->
       action_1;
  {_,true,_,_}->
       action_2;
  {_,_,true,_}->
       action_2;
  {_,_,_,true} ->
       action_3;
  {_,_,_,_} ->
       action_4
end.
```

Figure 6.12: Erlang: while condition

6.3.7 Seventh step: Remembers & Forgets

The seventh step consists in remember and forget beliefs. These processes are carried out with the *executor:remember(Belief,Time)* function, and the *bs:remove_belief*, which removes the whole belief, and *bs:remove_one_belief*, which removes a value from a multi-value belief, functions.

7. Conclusions

This chapter exposes the conclusions obtained during the development of this thesis. In addition, it defines the possible lines for future work.

7.1 Conclusions

After studying the Erlang syntaxes and OTP, we discovered the real potential of its mechanism for concurrent programming. The execution of several processes simultaneously enables the agent to react to environmental changes in a more efficient way. This fact was corroborated in the implementation of the first example in the Raspberry PI, which is defined in section 5.1.

Although the immutable data characteristic of Erlang was a big headache during the implementation of the data management, specially in the BeliefStore, the use of a functional language, such as Erlang, is more considerable than other object oriented languages. Because of its scalability and fault-tolerance. Moreover, in Erlang the incremental cost per-process is quite low, so it is possible to spawn hundreds of thousands of processes within a single application. However, it is also remarkable that the studying of a functional programming language like Erlang is not trivial, mainly because the way of work with these kind of languages differs significantly of the most common languages, which are principally imperative. Nevertheless, once the basics of these languages are known, the programs are easily followed, since its syntax is straightforward.

7.2 Future work

There are different ways to continue this work and carry out a deeper study of this area.

Firstly, the natural continuation is the development of a software that generate the Erlang code from a TeleoR program, using a software framework such as Xtext, further information about this framework can be found in `https://eclipse.org/Xtext/`.

Regarding the case study, one possible extension would be the implementation of the third version of the examples, which is explained in section 4.3, in two Raspberry PI. The communication between the Raspberry PI can be implemented with Kaa, which is a platform for the Internet of Things that allows connecting application, among other things: `https://www.kaaproject.org`. Thus, it allows to study the performance of the communication between two agents with Erlang.

Moreover, it is also pretended that a user, with an application such as GameSalad, `http://gamesalad.com/`, could specify the Teleo-Reactive system. Thus, the web environment generates the specification and with the translation procedure being able to generate the Erlang code that "interprets" the TR.

Bibliography

Armstrong, J. (2013). *Programming erlang: Software for a concurrent world* (Second ed.). The Pragmatic Bookshelf.

Clark, K. L., & Robinson, P. J. (2015). Robotic agent programming in teleor. *IEEE International Conference on Robotics and Automation (ICRA)*.

Elixir on the raspberry pi - blinking an led. (2015). Retrieved 2017-06-27, from `https://wtfleming.github.io/2015/12/10/embedded-elixir-raspberry-pi/`

Gubisch, G., Steinbauer, G., Weiglhofer, M., & Wotawa, F. (2008). A teleo-reactive architecture for fast, reactive and robust control of mobile robots. *21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, volume 5027 of Lecture Notes in Artificial Intelligence*.

The internet of things [infographic]. (2011). Retrieved 2017-06-27, from `https://blogs.cisco.com/diversity/the-internet-of-things-infographic`

Logan, M., Merritt, E., & Carlsson, R. (2011). *Erlang and otp in action.* Manning Publications Co.

Michaelson, G. (2011). *An introduction to functional programming through lambda calculus.* Dover publications.

Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 139-158.

Oliveira, P. (2015). *A simple, pure erlang, raspberry pi user-space gpio library.* Retrieved from `https://github.com/paoloo/gpio`

Vinoski, S., & Cesarini, F. (2016). *Designing for scalability with erlang/otp.* O'Reilly Media, Inc.

What is erlang. (2017). Retrieved 2017-06-25, from `http://erlang.org/faq/introduction.html`