Development and Testing of a TeleoR agent

Keith L. Clark and Peter J. Robinson

May 19, 2015

This guide assumes you have installed the QuLog+TeleoR system in a qulog directory. You need to have pre-installed the most recent Qu-Prolog (staff.itee.uq.edu.au/pjr/HomePages/QuPrologHome.html). You also need to install Pedro (staff.itee.uq.edu.au/pjr/HomePages/PedroHome.html), an inter-process communications server that supports addressed communication using email style addresses for processes, as well as publish/subscribe communication. Make sure the bin directories of the three installations are in your path.

The bin directory of qulog directory has three Python programs to facilitate the development and debugging of a TeleoR agent application: a robot shell, an agent shell, an agent/robot interaction logger.

The logger, logger.py, can be used independently of the two shells to remotely monitor the *BeliefStore* states and rule firings of a running agent controlling a robot or full robot/environment simulation.

The robot shell, robot_shell.py, is an interactive program that enables you to test TeleoR procedures one at a time by taking on the role of a robot and its sensors. The TeleoR procedures to be tested are each called as tasks of a configurable agent, provided in the teleor extension of the QuLog interpreter. The bin directory includes a small example TeleoR program for you to use while familiarising yourself with the robot shell. To write your own TeleoR programs you need to consult the tutorial book: *Programming Robotic Agents: A Multi-Tasking Teleo-Reactive Approach*, the first six chapters of which are down-loadable from www.teleoreactiveprograms.net.

The Python agent shell, agent_shell.py enables you to take on the role of a simple agent interacting with a robot simulation or the interface process of a robot. It allows you to send control actions to the interface, similar to those that would be generated from a TeleoR agent, and to see what percepts are sent back. It can be used to test the interface percept construction and its supported actions, and to calibrate the robot. For example, you can use it to determine how long it takes to turn through 90 degrees at a certain

speed.

You can also use the QuLog+TeleoR interpreter for this same purpose. This can be the next step after using the Python agent shell. You can then also consult a file of QuLog relation definitions, start up a TeleoR configurable agent, and deductively query the percepts that are returned from the robot or simulation before deciding on your action response. That way you may be able to induce some *guard*~>*actions* TeleoR rules that you can later embed in TeleoR procedures. We shall describe this interactive use of the QuLog interpreter before we describe the use of the more restricted Python agent shell tool.

The configurable TeleoR agent makes use of default percept and message handlers, which can be changed by adding certain QuLog action definitions to the consulted TeleoR agent program. The configuration options are described in Sections 6 and 7, and summarised in Section 12. These three sections assume familiarity with QuLog type declarations. To configure the agent you need to be familiar with QuLog action programming.

There is a summary of the recommended use of the various tools in Section 11

1 Example use of the logger and robot shell

In qulog\examples\introduction is a very simple TeleoR program tr_eg.qlg. We shall use this to describe how to use the robot shell. Look at the percept and action declarations of this program.

To start an agent executing a task using the files tr procedure, using both the logging tool and the interactive robot shell, you need 3 terminal windows.

In terminal 1, start the Pedro inter-process comms server and the logger process, naming it logger (or another name of your choice), with the two OS commands:

```
pedro                    % Has no effect if Pedro already running on this host
logger.py logger
```

The name logger will be registered with pedro as the name of the Python process allowing messages to be sent from the TeleoR agent.

In terminal 2, start the robot shell giving it a name robo (or some other name) using:

```
robot_shell.py robo
```

The name `robo` will be registered with `pedro` for this process. There is an optional `-w` flag that can be given in the command line. Its role is explained in the next section.

In terminal 3, in the `qulog/examples/introduction` directory, execute the following six commands:

```
qulog                                        % Start the QuLog interpreter
| ?? teleor.                          % Enter the TeleoR extension, prompt changes
| ?~> consult tr_eg.                          % Consult the example program
success                                                    % System response
| ?~> start_agent ag robo all.    % Start system provided TeleoR agent shell
success
| ?~> log logger.                              % Turn logging of task behaviour on
success
| ?~> start_task tsk tr.          % Start the agent's task executing call tr
success
```

The `start_agent` command starts a two thread agent process comprising a percepts handler and a message handler. `robotAg` is the name of this agent process and it will be registered with the `pedro` server. The second argument, `robo`, links the agent to the robot shell process as the destination for actions from its yet to be launched task thread. More usually this will be the name of a full simulation process or an interface process to an actual robot (more on this later). The last argument, `all`, is the convention used for sending new precepts. The other convention is `updates`. Both are explained below.

The two procedure `TeleoR` program `tr_eg.qlg` has a mobile robot look for, approach and get hold of an unspecified 'thing'. It is a simplification of the bottle collector program of Chapter 3 of the `TeleoR` book.

An agent task is started as a third thread, giving an architecture as depicted in Figure 1, by the last command `start_task tsk tr`. For us the source of the percepts and the destination for the control actions is the robot shell process. The `TeleoR` agent assumes the percepts source and action controls destination is always the one process named in the second argument of the `start_agent` command.

`tsk` is the name of the started task and `tr` the `TeleoR` procedure call it will execute. The procedure is in the consulted file `tr_eg`. You can use the task name to later terminate the task using a `kill_task tsk` command. If testing a `TeleoR` program with several procedures you can now start another task executing a different top level procedure call. A good strategy is to start with procedure calls that are at the bottom of the call chain and to work up testing procedures that call already tested procedures one at a time - bottom up testing of your control program. If your consulted `TeleoR` program

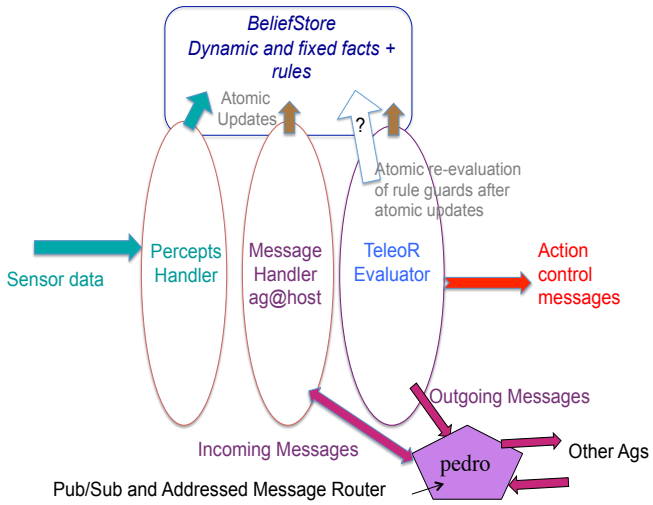

Figure 1: Three Thread Communicating TeleoR Agent Architecture

allowed for multiple robotic resource use, then you can have more than one task running at a time.

Immediately after executing `start_agent`, you will see an `initialise_` message in the robot shell window. The message is sent by the percepts handler thread. It is to tell a robot to ready itself to receive action controls and to send back to the percepts handler a list containing a set of percept facts, which are the application specific interpretation of sensor readings and any camera image, at that time. If you do not respond to the `initialise_` message within 5 seconds it will be sent again to remind you (or a robot interface process) that the agent is waiting for its initial set of percepts.

The interaction between the agent, the robot shell and the logger is depicted in the upper configuration of Figure 2.

2 The percept refresh conventions `all` and `updates`

`all` means that the robot interface will send all the percepts that hold at the time that they are sent as a list of percept terms, repeating a percept fact that might have been sent last time, if it still holds. The generic agent's percepts handler first removes from the *BeliefStore* any current percept belief that is not on the received list of new percepts. It then adds each precept on the received list that is not already in the *BeliefStore*.

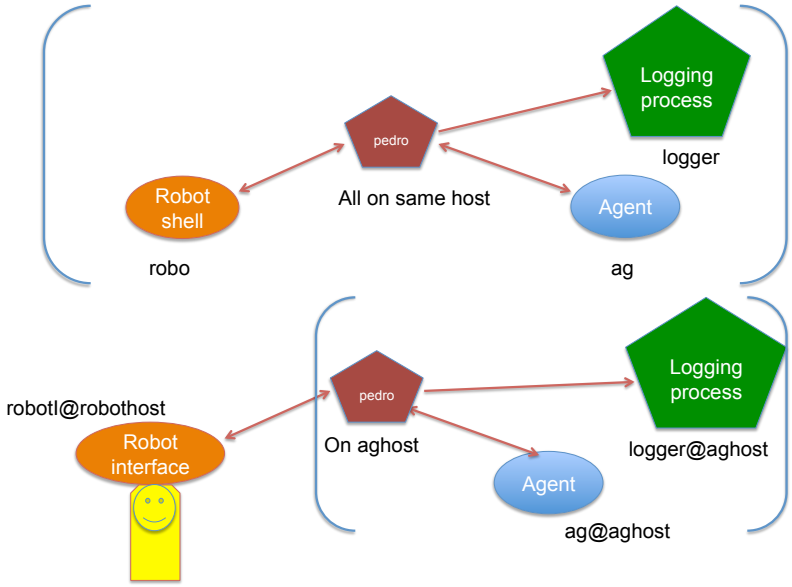

Figure 2: Emulating a robot and remote robot control

This corresponds to a simple sensor interface that does not remember previously sent percepts and sends *all* the robot's current perceptions of the environment state each time new percepts need to be sent to its controlling agent. If you have launched the robot shell *without* the `-w` flag this is what you must do when entering percepts. You must enter all the percept facts that you believe hold at the time you enter them.

`updates` corresponds to a sensor interface that remembers all the percepts that should be in the agent's *BeliefStore* as a result of percept information it has previously sent. It only sends the *updates* that must be made to these current *BeliefStore* percepts to record the latest perceptions. You use `updates` as third argument when you have launched the robot shell with the `-w` flag, or this is what your robot interface process does.

Using the robot shell with the `-w` flag you enter into the edit window only new percept facts, *not those that continue to hold*, and explicitly indicate, by prefixing with a - minus sign, those that *must be deleted*.

Suppose initially

```
see(left,3.1), holding
```

was the percept interpretation of sensor readings, and you have entered these in the edit window. They are sent to the agent shell as the list

```
[see(left,3.1), holding]
```

or as a list of wrapped percepts

```
[r_(see(left,3.1)), r_(holding)]
```

if you used the `-w` (for wrap) flag.

Suppose you next wanted to indicate that

```
see(left,2.9), holding
```

is now the case. Using the `all` option on agent launch and no `-w` flag on launch of the robot shell, you enter these two comma separated percepts in the robot shell's edit window. They will be sent to the agent as the list

```
[see(left,2.9), holding]
```

However, if the `updates` and `-w` flag combination, you record the the second set of percepts by entering

```
-see(left,3.1), see(centre,2.9)
```

They will be sent as the list of wrapped percept facts

```
[f_(see(left,3.1)), r_(see(centre,2.9))]
```

You do not have to resend `holding` as that will continue to be believed. However, you must now indicate the explicit forgetting of the previous `see(left,3.1)` percept belief.

A list of `f_` (for forget) and `r_` (for remember) wrapped percepts is easier to process in the agent's percepts handler by a single pass over the list. Also, where most sensor readings will stay the same it is a more efficient way to inform the agent of what it should now believe. The cost is that the robot interface has to keep track of which of the previously sent percepts will still be in the agent's *BeliefStore*, i.e. those sent in a previous `r_` wrapper that have not been rescinded by being sent in a `f_` wrapper.

3 Interacting with a `TeleoR` agent using the Python robot shell

After you see the `initialise_` message displayed in the robot shell window respond to it by entering some initial sequence of percepts, say just `see(4.7,left)`. To see what percept facts you can enter look at the **percepts** declaration in the example program file.

You can see what the TeleoR agent believes after you have sent this single percept by entering the command bs followed by $< fullstop >< return >$ or $< return >< return >$ in the agent terminal 3. At this point you should also see in the logger window that the task has fired the 3rd rule of the tr call, which has called get_to. This call has then fired its 4th rule resulting in the concurrent actions move(4),turn(left). The log will also display the percepts and dynamic beliefs of the agent when the rules were fired, and the time that these were last updated. There will be no beliefs displayed unless you have sent the agent a message (see below).

You will get the control message

```
controls_([start_(move(4)),start_(turn(left))])
```

displayed in the robot shell window, indicating that these two durative actions should be started. These are what would be sent to a robot interface process. After a short while you can enter something like see(4.1,centre). Try to get the agent to steer you (the robot) towards an unspecified seen thing until you are next to its centre at 0 distance. A history of entries into the robot shell that will achieve this is given in the history_all.txt file of the tools directory. You should then get back a

```
controls_([stop_(move(4)),exec_(grab)])
```

control messages response. In the logger window you will see that tr has fired its second rule.

After this you can either respond with holding, see(0,centre), as in the history, recording that the grab has been successful, or you can respond with see(0.3,centre), recording that the thing that you were next to has been pushed away by the grab action. You can also respond with no percepts at all (just hit return), corresponding to the thing having been pushed out of view of the camera.

The TeleoR procedure's response to an empty list of new percepts, entered at any time, will be action control messages to stop any forward move and start or modify a turn action to turn(left), because it will fire the default rule of the get_to procedure.

You might now like to start again, launching the agent with updates as the last argument. First do a

```
| ?~> kill_agent.
```

In the robot shell you will see a finalise_ message. The finalise_ is always sent when you kill an agent.

You do not need to restart the logger, but before restarting the agent you must relaunch the robot shell giving it a -w flag

```
robot_shell.py -w robo
```

This flag indicates that percept facts must be wrapped in `r_` and `f_` terms and that you will indicate that a `f_` wrapper should be used by prefixing the percept with a `-` (minus) sign.

Then execute

```
?~> start_agent ag robo updates.
```

in the agent terminal.

In the `tools` directory the `history_updates.txt` file gives a possible sequence of entered percepts for the `updates` percept sending convention.

4 Use of a `TeleoR` agent with a robot interface process

The first things you must do is decide on the percepts update convention you will use, and on the action repertoire and percept fact generation you will support in the robot interface process or the simulator. If the `TeleoR` program already exists, the actions and percepts must be those declared in that program. If it does not yet exist, these choices will depend upon the action and sensor capabilities of the robot. The chosen updates convention determines whether or not you interface always sends all the new sensor readings as unwrapped percept facts, or whether it remembers sent percepts not yet rescinded and just sends the percept updates as `r_`, `f_` wrapped percepts.

You can write the interface/simulator in Python, Java or C/C++. All have Pedro interfaces. The Python simulation programs in the `towers`, `bottle_collector` sub-directories of the `examples` directory exemplify how to do this in Python. Both use the API in `pedroclient.py`. The Pedro documentation may also be consulted. Your interface or simulator must register a name with the same Pedro server with which the agent and logger process (if used) will register their names. This name, say `robotI`, must be given to the agent as its second argument when launched. If the interface is running on a different host `Host` than the `TeleoR` control agent, the full handle *robotI@Host* must be given as second argument to the `start_agent` command.

The interface program you write must first wait for an `initialise_` message to be sent. This is the signal to gather sensor readings application specific interpretations of which are marshalled into a string representation of a `QuLog` list to be sent to the agent's percept thread via Pedro. The Pedro

handle identifying the sender of the `initialise_` message should be used. The percepts should all be wrapped as `r_(percept)` terms if your agent will be launched using the `updates` convention.

Thereafter, at application specific intervals, the interface collects new interpretations of sensor readings sending the required string to the agent's percept handling thread.

The control actions that the interface process will receive from a `TeleoR` task will be of the form

```
start_(durative), stop_(durative), mod_(durative), exec_(discrete)
```

where `durative` and `discrete` are ground durative or discrete actions declared in your `TeleoR` program. `mod` will only be received if the robot is already executing that durative action but with different parameters. For example, if it had received `start_(move(3.1)`, it might then get `mod_(move(2.4)` indicating a slowing down.

The list of controls is sent wrapped in a `controls_([..,`c_i`,..)` message, where each c_i is one of the above action control terms. This is what the robot interface must be able to process and map them into whatever low level behaviour calls needed to conform to the controls. The interface must also be able to handle the `initialise_` and `finalise_` messages sent when an agent shell is started and terminated.

The Python code in the `robot_shell.py` file will help regarding the writing of such an interface. It contains the code to register a name with a Pedro server, to get messages sent to it by an agent or any other process via Pedro, and for marshalling percepts into a string representing lists of percept facts, wrapped or unwrapped.

A possible configuration is shown in the lower part of Figure 2 in which the agent, Pedro and the logger process are on one host, `aghost`, and the interface is running on a host `robot host` in the robot. The interface process must register its name `robotI` with the Pedro server on `aghost`.

5 Using an on-board `TeleoR` agent and a remote logger

When you have a finished `TeleoR` control program, you might want to run a Pedro server, the agent process and the robot interface process on a robot. The agent and interface process then use the on-board Pedro server to communicate with each other. This configuration is depicted in Figure 3 where Pedro is running on `robot host`.

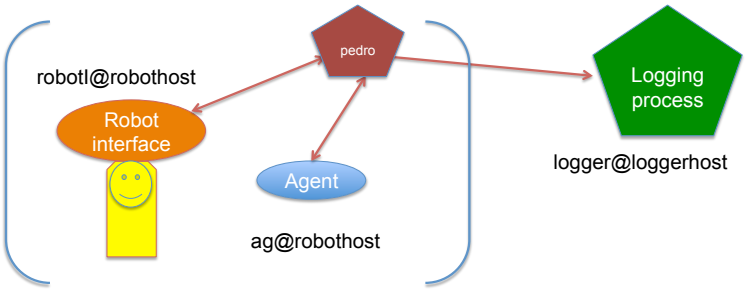

Figure 3: Using a remote logger and an on-board agent

Typically the logger is located on another host, `loggerhost`. In that case we must connect the logger with the Pedro server running on the robot's processor `robothost` by launching it with

```
logger.py logger -N robothost
```

`robothost` is the name or IP address of the robot's on-board processor. The -N switch causes the `logger` process to register with the Pedro server on this host.

In addition, the `log` command executed in the agent process, must be

```
| ?~> log logger@loggerhost.
```

in order to have log information dispatched to the remote logger. The advantage of this configuration is that we can have several loggers, running on the same remote host, each connected with a Pedro server on a different robot, telling us what each robot control agent is doing and the state of its *BeliefStore*.

6 Tailoring a TeleoR agent's percept and action handling

Sometimes, after all the new percepts have been added, and before any control message response is decided, it is useful to check the *BeliefStore* for consistency regarding all its dynamic beliefs, perhaps deleting, adding or modifying some of them. You can have this done by the shell agent's percept handling thread by defining a no argument QuLog action procedure procedure

```
checkBS_.
```

This can be used to ensure, for example, that there is only one see belief when using the example program. This should, of course, always be the case, but you can also check if you want. If this procedure is defined it will be called by the percepts handler as the last thing it does during its atomic transaction on the agent's *BeliefStore*.

Finally, you can use your own format for sending new percept facts, and your own convention indicating how the percepts in the agent's *BeliefStore* should be updated, by defining one or both of two QuLog action procedures

```
get_percepts_:([term]?)
handle_percepts_:([term])
```

If handle_percepts_ is defined in your TeleoR program, it will be called each time a list or percepts has been received by the percepts hander. Neither default handler for the all and updates conventions will not be used. If you define handle_percepts_:([term]) in your TeleoR program file you must launch the agent using the user flag instead of an all or updates flag. There is an example handle_percepts_ action procedure in the towers.qlg TeleoR program file in the qulog/examples/towers directory.

If get_percepts_ is defined in your TeleoR program, it will be called immediately after a list of percepts has been processed. It could poll for a new batch of percepts sent in any string format you like, and suspend until they are received. It should then use QuLog string processing to extract sub-strings representing percept facts, or to extract sensor data and build the percept facts using the QuLog @.. term builder and the primitive string_to_term converter relation.

If your robot uses ROS, and you configure it so that there is a percepts publisher that sends all the interpreted sensor readings as a string of the same format as the strings you must enter into the robot_shell.py tool, i.e. a string of the form "[P1,P2,..,Pk]", where the Pi are percept facts as declared in your TeleoR control program, you can use the ros_pedro.py example interface between ROS and Pedro to route this percept list to the

TeleoR agent. In that case you do not have to define get_percepts_, all you need do is launch your agent using the name ros_pedro for the robot interface process. This interface handles the initialise_ message sent when you start the agent and does not forward it as a control message to the robot.

The default of the TeleoR agent's task evaluator is to send a

```
controls_([control_action_])
```

message in string form via Pedro to the robot interface process. The type control_action_ has the definition

```
control_action_ ::= start_(durative) | stop_(durative) |
                    mod_(durative) | exec_(discrete)
```

where durative and discrete are the action types defined in the TeleoR program.

You can change this default behaviour by defining a QuLog action procedure

```
send_robot_message_: (robot_message_)
```

where

```
robot_message_ ::= hand_shake_ || controls_message_

hand_shake_ ::= initialise_ | finalise_

controls_message_ ::= controls_([control_action_])
```

in your TeleoR program file. This is then called by the task evaluator, passing in the list of control actions that need to be converted into control messages sent to the robot interface.

If your interface is the ros_pedro.py example interface and you have a ROS process that has subscribed for controls publications and can handle a string payload which represents a list of control actions, you need not define send_robot_message_. All the robot messages sent by the agent will be handled by this interface. However, if you do not want to have the controls subscriber handle start, stop etc messages, you can can modify the ros_pedro.py to convert to the controls publications you want to handle, or you do this conversion in a send_robot_message_ action procedure which ends with a message send of the form

```
YourControls to ros_pedro@localhost
```

If ROS is used on the robot and the Ros/Pedro Python example interface is used for the agent/robot communication the appropriate configuration is that of Figure 4.

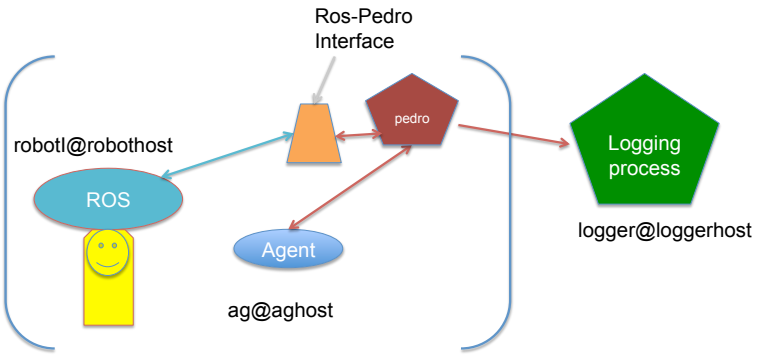

Figure 4: Using a ROS robot with a ROS/Pedro Interface Process

7 Receiving and handling messages

A `TeleoR` agent can be sent messages from other `TeleoR` agents, or processes with a Pedro registered name, using the Pedro handle `Agent@Host`, where `Agent` is the name given to the agent in the `start_agent` command, and *Host* is the singly quoted name (e.g. `'zeus.doc.ic.ac.uk'`) or IP address (e.g. `'192.168.1.72'`) of the host on which the agent was launched. Such messages are handled by the agent's message handling thread. A message `Mess` is sent to the agent by execution of the `QuLog` action

```
Mess to Agent@Host
```

The default behaviour of the `messages` thread is to check that the message `Mess` is ground, and if so to record its receipt, from its sender process (or agent) with Pedro handle `Nm@HostNm`, by remembering a fact

```
message_(Mess,Nm@HostNm,Time)
```

Here, `Time` is the time of receipt of the message. The `QuLog` type of the `message_` relation is

```
belief message_:(term,agent_handle,num)
```

where an agent handle is a term of the form `atom@atom`.

You can directly query these `message_` fact beliefs in your `TeleoR` rule guards to find out what information has come from other sources. Or, these beliefs can be processed by launching another thread inside the agent executing a `QuLog` action procedure which converts the `message_` beliefs into beliefs using application specific predicates that are declared as **belief** predicates in the consulted `TeleoR` program. An example of such a conversion might be the replacement of a `message_(tell(B),Ag,Time)` belief, where `Ag` is believed to be reliable, by a belief `B`, providing `B` is a ground fact for a belief predicate tested by `type(B,!belief)`. The `!` signals that `B` must be ground.

As an alternative, you can define a `QuLog` action procedure

```
handle_message_:(term?,!agent_handle)
```

in your `TeleoR` program file. If present, this will be called by the message handler of the `TeleoR` agent shell each time a message is received, with first argument the message term and second argument the Pedro handle of the sender. The `?` postfix indicates that the message term may contain variables - e,g. queries from other agents - and need not be ground by the call to `handle_message_`.

The program file `bottle.qlg`, in the `bottle_collector` sub-directory of the `examples` directory, has such an application specific `handle_message_` action procedure.

8 Driving a robot as with a remote control

You actually do not need to start an agent task thread executing a `TeleoR` procedure call to have action control messages sent to the interface process of a robot. You can act as the control program, just as you can act as the robot using the robot shell tool. You should however have declarations of the **durative** and **discrete** actions that the robot interface supports in a file that you consult.

You do not even have to the robot interface support the sending of percepts in order to drive a robot. You are then driving the robot as though you were using a remote control to get it to perform different actions. You are then the robot's sensors.

To do this, using a control action supporting interface process on the robot, execute a

```
| ?~> start_agent driver robotI@robothost none.
```

command. The `none` indicates that no percepts will be processed by the agent and converted into percept facts even if sent from the robot interface. So none need to be sent.

This will still send `initialise_` to the interface process `robotI@robothost`, which should be already running. You should also have consulted a file that at least declares the `durative` and `discrete` actions, and their argument types.

You can now send appropriate action control messages using an `actions` command such as

```
| ?~> actions [move(2.5), turn(left,0,3)].
```

These will be mapped into a list of control actions, wrapped as a

```
controls_([start_(move(2.5)), start_(turn(left,0,3))])
```

term dispatched to the robot's interface process for execution by the robot, unless you have defined your own `send_robot_message` as described in Section 6.

You watch what the robot does, just as you would a remote control device, and at the appropriate moment you execute another `actions` command, for example

```
| ?~> actions [move(3.0)].
```

mapped into the control message

```
controls_([mod_(move(3.0)), stop_(turn(left,0.3))])
```

You can use the up cursor key to display and then edit previous `actions` inputs to speed up your responses.

Entering

```
| ?~> actions [].
```

will stop all executing durative actions.

You are step by step controlling the robot - real or simulated. This user control is particularly useful when you are testing the interface and the repertoire of actions it supports. You can also use it to calibrate actions. For example, to find how long it takes for a robot to turn through 180 degrees when instructed to turn at a certain speed.

The advantage of driving without percepts feedback is that you can first implement the robot interface so that it just supports action controls. When you are happy with that, you can add the percept gathering and dispatch. You can use this minimal mode even if the interface does dispatch percepts. The `start_drive` command does launch a thread that sends the `initialise_` message, but it discards any percepts it receives.

When you want to stop driving the robot you can do a

```
kill_agent.
```

to send a `finalise_` message to the robot interface process.

9 Driving a robot while getting and querying its percepts

Suppose you do have an interface that will send percepts and accept action controls. You can launch an agent shell in order to process the percepts that will be returned from the interface. You should consult a file that minimally declares the percept predicates and their argument types as well as the durative and discrete actions. If you have tailored the shell agent this file might also contain `QuLog` action procedures for `get_percepts_`, `handle_percepts_`, `checkBS_` and `handle_message_`.

You can still drive the robot yourself using the `actions` command. It is probably best to use a logger process as that will automatically keep you up to date as to the agent's changing *BeliefStore*, and keep a log of the actions you will send.

After launching the agent shell, when you see in the logger window that percepts have been sent to the agent in response to the `initialise_` message, you can decide what an appropriate action response(s) should be to the dynamic beliefs now in the agent's *BeliefStore*. These will have been displayed in the logger window. You can also see them in the agent terminal window using the command `bs`. If you have defined relations in the consulted file, you can also query these relations to get inferred interpretations of the current percept beliefs.

Each time you execute an `actions` command the interface process will respond by executing the received controls. Independently, and probably quite frequently, it will send percepts, which will be immediately displayed in the logger window together with the new state of the *BeliefStore*. At any time, you can respond to these changing beliefs, just as a `TeleoR` program would respond, by sending back control actions by entering an `actions` command.

BeliefStore monitoring combined with action by action driving of a robot is one way of finding out what an appropriate action response might be to particular states of the agent's *BeliefStore*, enabling you to induce some of the `TeleoR` rules to be used to achieve certain goals.

10 Using the Python agent shell

The minimal way of driving a robot and testing its interface, when all that you have implemented is an action repertoire, is to use the Python agent shell `agent_shell.py` in the `qulog/bin` directory. You launch this with a terminal command

```
agent_shell.py ag "robotI@robothost"
```

You will get a GUI similar to the robot shell window but in this case what you will enter into the edit window are control actions for the robot interface process `robotI@robothost`, not percepts.

The agent shell will have sent an `initialise_` message to this process and as you should have programmed it to respond to this message by sending back a list of percepts these will be displayed in the GUI's main window.

If you are using this Python tool preparatory to eventually using the `QuLog`+`TeleoR`system, you should send the control actions that will be generated from a running `TeleoR` agent. That is you enter into the edit window a comma separated sequence of control actions of the form

```
start_(durative), stop_(durative), mod_(durative), exec_(discrete)
```

preceded by the word `controls`. As an example,

```
controls  start_(move(4.2)), stop_(turn(left,1.2)), exec_(beep)
```

might be entered into the edit field. With this entry in the edit field, the term

```
controls_([start_(move(4.2)), stop_(turn(left,1.2)), exec_(beep)])
```

is actually sent, exactly as would be the case from a `TeleoR` agent that had been turning on the spot but now had fired a rule with the parallel actions `move(4.2),beep` when the previous action had been `turn(left,1.2)`.

11 Summary of the tools

- The logger is a Python process that can be used to remotely spy on the sequence of *BeliefStore* states of an agent, and to see the rule firings and primitive actions response whenever these change.

- The Python agent shell enables you to test an interface to a robot with respect to the percepts it sends and the actions it supports, before developing any percept querying rules. You can use it even when the interface supports only an action repertoire to test just this, and to remotely control a robot. You can use it to time how long actions take to achieve some goals where there is no percept that can be used to determine the goal has been achieved (e.g. turning through 180 degrees without a compass).

- The generic `TeleoR` agent enables you to do the same as the Python agent shell, but also to query the returned precepts using `QuLog` *BeliefStore* rules. It also allows you to test, bottom up, a set of `TeleoR` control procedures by launching each as the initial call of an agent task, and then terminating it before the next procedure is tested. This is when the logger should definitely be used to monitor the decision behaviour of each procedure. If the `TeleoR` program you are testing is for a multi-tasking agent controlling several robotic resources you can run several tasks concurrently. The logger will then tell you which tasks are running, which resources they are using, which tasks are waiting, and the resources they need. It will also tell you which rules have been fired in each task wherever this changes, even in the waiting tasks.

- If you want to do this incremental testing of a set to `TeleoR` procedures when the robot interface or simulation is not available, you can simulate the robot using the Python robot shell. You imagine what will happen when certain action controls are received and, either immediately, or after a suitable time interval, you send back the percepts that would be generated from sensors. You do not have to do this realtime, it can be slow motion unless you have time dependent behaviour in your `TeleoR` procedures.

12 Summary of the application specific optional action procedures

The supplied TeleoR agent can be specialised by defining all or some subset of the following in your program file. None need be defined. If not there is a default behaviour as described earlier.

- `get_percepts_:[term]?` - an action called as soon as the agent is launched and each time a returned list of terms has been processed and converted into remembered `percept` facts.

- `handle_percepts_: [term]` - an action called after a new list of terms have been received from the robot interface, possible using a program defined `get_percepts_`. It must be defined if the `user` option is given when the agent is launched. It should check that each term `P` on the input list is a correctly typed ground `percept` term using a `percept(P))`. This test will use the program's `percept` declarations.

- `handle_message_: (term?,agent_handle)` - an action called when a message term is received from an agent identified by its agent handle `agent@host`. Any reply should be sent to `agent@host`. It will go the `messages` thread of this agent, even if the message was sent from some other thread within the agent.

- `checkBS_` - a no argument action procedure that can check the *BeliefStore* for consistency and perhaps remember and forget certain beliefs. It is called immediately after a program defined `handle_percepts`, or one of the default percept handlers, has updated the *BeliefStore*.

- `mod_controls_:([control_action_],?[control_action_])` - a relation called when the list of control actions generated from the firing of new TeleoR rules has been determined by comparing the new robotic actions with the last robotic actions that were determined. This is an opportunity to add or remove control action. The input first argument are the program determined control actions, the second argument is the possibly modified first argument. It is this output that is sent to the robot interface process. The procedure should rarely alter the action controls as the primary determiner of actions should be the TeleoR procedures. There is an example of the need to occasional adding of control action in the supplied two arm using block towers building program in the `qulog\examples\towers` directory.

- `send_robot_message_: robot_message_` - an action primarily called in order to send control actions to the robotic interface as a `controls_(CActs)` message. It must also handle the `initialise_` and `finalise_` messages, even if it just ignores them.