# `TeleoR` User Guide

Keith L. Clark and Peter J. Robinson

September 17, 2021

# Contents

# 1 Introduction

In this guide we focus on the TeleoR extension of QuLog. TeleoR is a major extension of Nilsson's *Teleo Reactive Procedures* [2][3], a rule based language for programming robotic agents. We shall refer to Nilsson's language as T-R. T-R evolved from the triangular program plans of Shakey [4], the first AI robot developed by a team lead by Nilsson in the 1960s. Example extensions of T-R in TeleoR are: use of the mode/type system of QuLog so that the compiler can check that every rule determined robotic action will be fully instantiated and correctly typed, extra rule forms for more nuanced control of robot behaviour, high level control of multi-tasking with the fair alternating shared use of robotic resources by the different tasks achieved simply by declaring which TeleoR procedures may be invoked to *start a task*, and which are *atomic*. The latter may only be entered by a task when all the robotic resources they may need are available and no other task has been waiting longer for any of these resources. Once entered, the task has exclusive use of these resources whilst the procedure is active.

## 1.1 TeleoR Robotic agents: *Deductive Belief Store*, robotic resource actions, sensor percepts, inter-agent communication

A TeleoR *robotic agent* is a *multi-threaded software process* that controls of one or more *robotic resources*, e.g. one mobile robot, two robotic arms, possibly alternating their use between several concurrent tasks. It controls the robotic resources using a repertoire of control actions, such as `turn(left,0.4)`, `pickup(arm1, block2)`, in order to perform its tasks. The devices exist in a real or simulated environment external to the agent. The tasks have goals to bring about, or to maintan, or to prevent, certain states of the agent's environment. The agent does this by frequently sensing the environment of the robotic resources, and by responding with appropriate resource actions, as the sense data changes. Other agents may be in the same environment controlling different robotic resources. They could be working on completely independent tasks, or co-operating on some or all the tasks.

The sensors may be attached to the robotic resources, or be completely independent. A camera mounted on a mobile robot, or at the end of a moveable telescopic arm, is an attached sensor. A wall mounted temperature sensor, or a camera that automatically pans using a fixed trajectory, sending video images at frequent but fixed intervals, is an independent sensor.

The readings of the sensors are converted into *percepts*: facts such as `close(bottle,left)` and `on(block2,block4)`. This conversion is done by software external to the `TeleoR` agent. Each percept is a logical interpretation of one or more sensor readings that is relevant to the agent's tasks. The facts are handled by a special percept handling thread within the agent. At a frequency required by its tasks, this thread *atomically* updates the agent's *Belief Store BS*. During the update no other agent thread is active.

The *BS* facts are what the agent *believes* about the current (and perhaps past) state of its environment, about other robotic agents controlling robotic resources in the same environment, and the robotic resources that it and the other agents control. Non-percept belief facts may have been distilled from messages it has received from other agents, or may be facts encoding memories of past perceptions and actions.

Reasoning agents also have a separate knowledge layer *KL* of *fixed knowledge rules and facts*. The *KL* facts encode *fixed properties* of the environment and other agents relevant to the agent's tasks , e.g. that a red door connects two named rooms. They are fixed in that they will not change during the agent's lifetime. The *KL* rules are used to draw inferences from *BS* and the *KL* facts. They encode information about the semantics of the predicates used in these facts and define new task relevant fusions of the facts. Facts about the environment that might change as a result of exogenous events, such as which doors are open or closed or blocked, are dynamic beliefs. The dynamic beliefs and the knowledge rules and facts comprise the agent's *deductive belief store*, which we shall henceforth call its *DBS*.

Often all the robotic resources and sensors are on one robot, e.g. a small mobile robot with a forward facing camera, bump sensors and a gripper with touch sensors, *and* the agent is a process on a computer inside the robot. Alternatively, the robotic resources and sensors could be distributed around the rooms of a house with the agent being a process on a laptop using WIFI.

## 1.2 Agent communication and co-operation

Agents may communicate with other agents and non-agent processes directly via asynchronous message communication. This may be addressed communication with the identity of the receiver and sender attached to the message, or indirect communication using a publish/subscribe server. Additionally, agents might communicate indirectly by querying and updating an external shared memory, such as a data base or Linda tuple store[1]. They may even communicate by making certain changes to the environment state, e.g. the analogue of the pheromone trails left by ants.

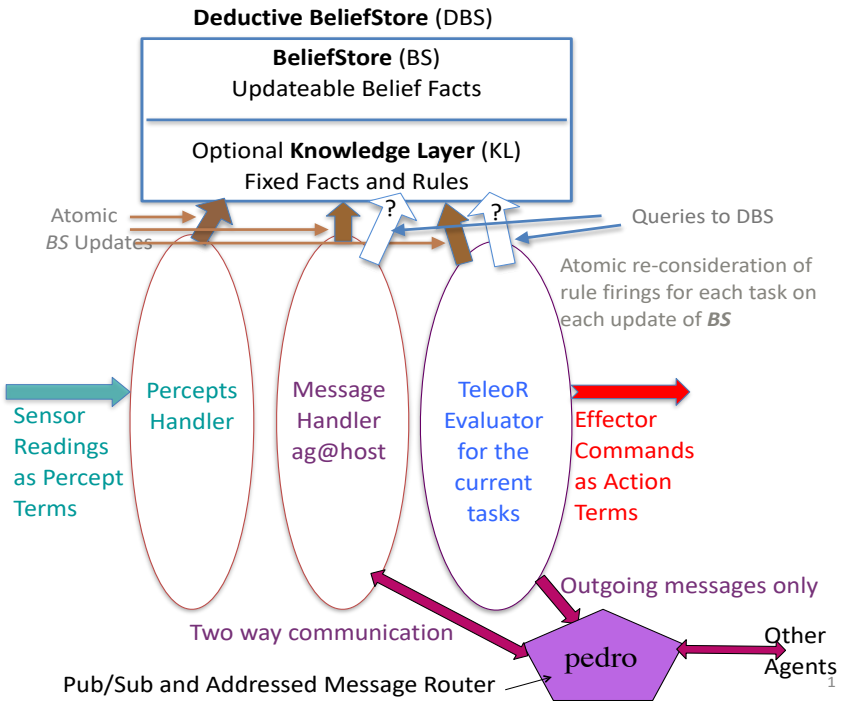


Figure 1: Three Thread `TeleoR` Agent Architecture

A robotic agent $RA$ queries its $DBS$ to determine which device action $A$ to use next for a particular *task*, $T$. If a group of agents are co-operating on $T$, this might be an action of a device controlled by another agent $RA'$. $RA$ may need to explicitly request that $RA'$ initiate the action, or $RA'$ may automatically do the action when it determines what needs to be done next for shared task $T$. Either way, `A` will be sent to the simulation, or to a ROS[6] or MQTT[5] node that has direct non-agent control of the robotic device in question. $A$ is being done to try to bring about a change in the common external environment of $RA$ and $RA'$ that will progress the task `T`.

### 1.3 A three thread agent architecture

Figure 1 depicts our three thread `TeleoR` agent architecture using an inter-agent communication server, `Pedro`. `Pedro` supports both addressed communication to an identified agent as well as publish/subscribe unaddressed communication. The `TeleoR` evaluator thread effectively re-evaluates the

initial `TeleoR` procedure call of each active task and dispatches the actions for all the robotic resources for which each task has exclusive control at that point in time.

### 1.4 Task goals and multi-tasking

At any time, a robotic agent has one of more *tasks*, each of which has a *goal* to bring about or to maintain some state of the environment using the robotic devices that the agent controls. As an example, an agent able to control a robotic arm at the end of which is a gripper so that the arm may be used to pick up and put down small objects on a table, might be given a task to build a tower of labelled wooden blocks. The task goal would be something like `tower([block3,block2,block7])`. If an agent has several tasks it can interleave the attempts to achieve a compatible subset of its task goals, alternating the use of its robotic resources between the tasks. Tasks waiting for the use of robotic resources are queued and become active only when the robotic resources they need have been released by another task or tasks. Tasks release robotic resources only when they have completed some atomic sub-task, a task that should only be interrupted if its execution pre-condition no longer holds. If two tasks at some stage will use different robotic resources, and their respective action uses will not interfere with each other, the robotic resources may be used in parallel, e.g. two robotic arms may be moved concurrently providing they will not clash.

The percept construction is usually done by a process external to the robotic agent, and is often programmed in an imperative language. Each new batch of percept facts is used to update the percept facts in the agent's *BS* that record sensed properties of the robot's current environment state, properties relevant to its controlling agent's tasks. On each update, percepts that no longer hold are removed and new percepts are added. Percepts that persist from the previous snapshot of the environment state are left in the *BS*.

Keeping only the lastest percepts is sufficient for many applications. However, each time `T` the *BS* is updated as well as remembering each *new* percept as a fact such as `open(room1,door2)`, we can also remember a time stamped meta fact `remembered_at(open(room1,door2),T)`. If in some subsequent percepts update at time `T'` `open(room1,door2)` is no longer included, the meta fact `forgotten_at(open(room1,door2),T')` is added and `open(room1,door2)` is removed. This way the different time intervals in which these key perceptions were once believed is rememberd.

After each such percept update the agent's control program for each

active task will determine the appropriate next action to make progress towards, or to maintain the task goal. It may also communicate certain percept updates to other agents, e.g. a mobile robot observing that a door is now closed could broadcast this new belief via a publish/subscribe server.

If the environment is the real world, changes may be brought about by people and acts of nature - such as wind. To an individual agent, these other changes to the environment are exogenous events. They may *help* or *hinder* the agent in achieving a task goal. Ideally a task action selection program should be robust to such exogenous events. That is, when the effect of the event has been sensed and reported in a percepts update, the control program should take advantage of any help and recover from any hindrance. This robustness is a key feature of teleo-reactive control.

## 2 Overview of the TeleoR language

A TeleoR program comprises sequences of *guard* ~>*action* rules clustered into parameterised procedures of the form:

```
tel p(t1,..,tk) % declaration of the argument types of p
p(X1,..,Xk){
    G1 ~> A1
    .
    .
    Gn ~> An
    }
```

The $t_i$ are the types of the procedure's $k$ arguments named by the $k$ different parameter variables $X_1,..,X_k$. Each $t_i$ may be a primitive QuLog type such as num, list, or a QuLog program defined type, or a QuLog type expression for a relation, function or TeleoR procedure. The typing of TeleoR procedures and the allowing of code arguments is a major difference between TeleoR and T-R.

The parameters $X_1,..,X_k$ are *global variables* of the action rule sequence. They must be given fully instantiated values of the required type (or subtype) when the procedure is called. Other variables appearing in a rule are *local variables* of that rule. Unless explicitly quantified, they are implicitly universally quantified over the rule in which they appear.

Each guard $G_i$ is a QuLog query to the agent's *BS*. This is an & conjunction of conditions, negated conditions, and universally quantified condition implications. Disjunction is not allowed in a guard, just as it is not allowed

in a QuLog rule body. (A disjunctive condition can always be captured by having a call to an auxiliary QuLog relation with two or more defining rules.) The *action* is one of:

- a tuple of actions for robotic resources, to be executed in parallel,
- a call to a TeleoR procedure, including a recursive call on the procedure p,
- a type correct call $\texttt{Exp}_j\texttt{(....)}$ where $\texttt{Exp}_j$ is an expression with value a TeleoR procedure named and defined in the program,
- a timed sequence of $k > 1$ of the above actions of the form;
  $\texttt{[RA}_1\texttt{:T}_1\texttt{, ...,RA}_k\texttt{:T}_k\texttt{]}$
  where the time limit $\texttt{T}_k$ on the last action is optional

together with a communication action.

Below are two example TeleoR procedures for a search and grasp task for a mobile robot.

```
def thing ::= bottle | glass
def robotic_action ::=
          move(num) | turn(dir) | grasp() | release()
def dir ::= left | centre | right

percept see(thing, num, dir), holding(thing)

tel get_hold_of(thing)
get_hold_of(Th) {
    holding(Th) ~> ()
    not exists AnyTh holding(AnyTh) & see(Th, 0, centre) ~>
                                      grasp()
    not holding(_) ~> get_to(Th)
     % The guard is equivalent to  not exists X holding(X)
    true  ~> release()
    }

tel get_to(thing)
get_to(Th) {
    see(Th, 0, centre) ~> ()
    see(Th, 0, Dir) ~> turn(Dir)
    see(Th, centre) ~> move(6)
```

```
see(Th, Dir) ~> move(4), turn(Dir)
% The concurrent move and turn actions will cause the robot
% to swerve towards Th, hopefully bringing it into centre view
true ~> turn(left)
}
```

The two uses of `Dir` in the second and fourth rules of the `get_to` procedure are both implicitly universally quantified over their respective rules. The use of `AnyTh` in the second rule of `get_hold_of` is inside an existential quantification. Local variables in different rules with the same name are *different* variables. Any local variable appearing in the rule *action* must also appear in the rule *guard*, and be such that it will be have a fully instantiated value by the end of the guard evaluation. These conditions are checked at compile time.

An agent task is started by a single call $\texttt{p(v}_1\texttt{,..,v}_k\texttt{)}$ to some program procedure `p`, e.g. `get_hold_of(glass)` or `get_to(bottle)`. The fully instantiated arguments given in the call typically partially or completely instantiate each guard of `p`'s rules, e.g. the first rule of `get_hold_of` procedure for the call `get_hold_of(glass)` is fully instantiated, and all the rules of `get_to` for call `get_to(bottle)` become fully instantiated. Suppose the guard of the first rule becomes $\texttt{G}'_1$. This is the *task goal*. That is, by sending a sequence $\texttt{a}_1\texttt{,..,a}_n$ of `n` actions to the robotic resources, eventually some instance $\texttt{G}''_1$ of $\texttt{G}'_1$ should become inferable from the agent's updated *BS*. From the time that the task was started, the *BS* will have been updated as a result of a percepts update or a received message, at least as many times as different actions have been sent to the robot.

A task call evaluation involves finding the *first* partially instantiated guard $\texttt{G}'_j$ which has an instance $\texttt{G}''_j$ inferable from the current *BS*. The j'th rule of the call is then *fired*. The corresponding ground instance[1] $\texttt{A}''_j$ of the rule's action is either a robotic device action, or it is a call to a `TeleoR` procedure. If it is a procedure call, again the first rule of this call with an instance of its guard inferable from the current *BS* is found, and that rule fired. Eventually a rule with a tuple of robotic actions will be fired and this tuple will be dispatched to the robotic devices interface, or to the simulator. As mentioned in the final section, it is up to the robot side to deal with these tuples by, for example, stopping actions that are no longer active and modifying actions whose arguments have changed.

[1] Compile time analysis ensures that the action of every `TeleoR` rule will be ground if its guard is inferable.

The robotic action of the first rule of each TeleoR start procedure is generally the empty action (). When () is received by the robotic device interface it will cause the last dispatched action $\mathtt{a}_n$ to be terminated. If the first rule does not have the empty device action, it should have an action that will normally preserve the task goal. After the first goal achieved rule has been fired, an exogenous action may 'interfere' and after a percept update of the *BS* the guard of the first rule may no longer be inferable. In that case, the TeleoR task evaluator will start testing the other guards of the procedure, starting with the second guard. In effect, the call $\mathtt{p(v}_1\mathtt{,..,v}_k\mathtt{)}$ is re-evaluated to try to re-achieve the task goal.

Actually, a TeleoR task programmed agent responds to each update of its *BS* by effectively re-evaluating the task call $\mathtt{p(v}_1\mathtt{,..,v}_k\mathtt{)}$[2].

The environment in which the robot is situated may contain other separately controlled robotic devices, as well as humans, all of which may have their own goals. Their actions may sometimes hinder agent *RA* or help it. No matter what, the TeleoR control program will determine an appropriate response. It will skip actions if helped and redo actions if hindered. Procedure p, and all the other TeleoR procedures that may be called as part of the evaluation of call *C*, are a universal conditional action plan for achieving any goal which is a call instantiation of the guard of p's first rule. TeleoR programs recover from setbacks and take advantage of good fortune. This makes TeleoR programs rather different from traditional programs.

TeleoR agents typically reside on the robotic device hardware, especially when this is a mobile robot, but one TeleoR agent could be controlling several robotic devices. When discussing TeleoR agents and programs we will use the term *robot side* to mean the part of the hardware for collecting perceptual information (percepts) and for realising actions sent from the agent: turning on/off motors, actuators etc.

TeleoR programs typically contain QuLog declarations of types and definitions of relations, functions and actions and so we assume the reader has read the QuLog user guide.

The system comes with its own interpreter that extends the QuLog interpreter with extra functionality that is required to compile and run TeleoR programs. Unless the TeleoR control agent directly communicates with the external robotic devices and sensors using something like ROS [6], *and* has no need to communicate with other agents using the Pedro communication server, we start the TeleoR interpreter giving the agent process a name

[2]In practice the call stack is examined from the task call entry - the bottom entry - to the entry for the last rule firing - the top entry.

which will be registered with Pedro.

```
teleor -AagentName
Teleor Version 1.0

| ?~>
```

The `| ?~>` is the `TeleoR` command prompt.

The agent process can now communicate with other agents and processes via Pedro, including the robot side process that handles action commands from the agent and sends it sensor readings as percept facts. The actions and the percept facts communicated between the agent and robot side can be tailored to the tasks that the agent is programmed to perform. So programming the robot side from scratch, or modifying and previously programmed robot side, is usually needed in addition to the `TeleoR`+`QuLog` programming of the agent. We assume the reader has the competence to do this robot side programming.

The `TeleoR` language is introduced via a series of example programs. All the programs discussed here can be found in sub-folders of the `examples` folder.

# 3 Example `TeleoR` Agents

## 3.1 Introductory Example: Declarations, Logger, Robot Shell

In `qulog/examples/introduction` there is a very simple `TeleoR` program `tr_eg.qlg`. We shall use this to introduce the basic declarations and semantics and describe how to use the logger and robot shell. The purpose of the `TeleoR` agent programmed in this file is to search for an object, move to it and grab it.

`TeleoR` procedures require type declarations similar to that of relations, functions and actions. Like functions they only take ground arguments and so mode are not included in the declaration. We have two declarations of `TeleoR` procedures:

```
tel get_object()
tel get_to()
```

As well as declaring the types of the `TeleoR` procedures, the minimal requirement is to declare the percepts and primitivie actions:

```
percept see(num, dir), holding()
def robotic_action ::=
        move(num) | turn(dir) | grab() | release()
```

The percept declaration has a dual purpose: it declares the terms that the robot side will send to the agents percept handling thread; and also declares them as `dyn` terms. The percept handler processes percept messages and updates the *BS*.

In response to percept changes the agents evaluator thread will typically send a collection of robotic actions to the robot side to execute. The `robotic_action` declaration lists the allowed robotic actions.

These two declarations are therefore describing the interface between the `TeleoR` agent and the robot side.

As we said above, the purpose of the `TeleoR` agent programmed in this file is to search for an object, move to it and grab it. It contains the top-level `TeleoR` procedure `get_object` and the `TeleoR` procedure `get_to` that is called from the first:

```
tel get_object()
get_object() {
    holding() & see(0, centre)       ~> ()
    not holding()  & see(0, centre)  ~> grab()
    not holding()                    ~> get_to()
    true                             ~> release()
    }

tel get_to()
get_to() {
    see(0, centre)       ~> ()
    see(0, Dir)          ~> turn(Dir)
    see(_, centre)       ~> move(6)
    see(_, Dir)          ~> move(4) , turn(Dir)
    true                 ~> turn(left)
    }
```

Each `TeleoR` procedure definition consists of an ordered sequence of rules that are guarded actions. In simple `TeleoR` procedures the guard of the first rule is the goal of the procedure and the action is typically the empty action and, in particular, the guard of the first rule of the top-level procedure is the

overall goal of the program. In this case that goal is to be holding something and to see an object immediately in front. In this example we take that to mean the robot is holding the object. Later we will see procedures where the goal of the procedure is not the guard of the first rule.

The guards of the procedures typically query directly, or indirectly, the *BS*. In this example the guards directly access the percept part of the belief store. This means that, as the percepts change, different rules will be chosen and different actions will be enacted. Later we will see examples where the belief store contains facts that come from sources other than the percept handler.

To start the program running we first have to start the `TeleoR` agent:

```
| ?~> start_agent(robo@localhost, all)
```

The first argument is the address of an external process which is the interface to the robotic devices and the sensors. It could be a simulation process. The second argument gives the percept update convension. When this argument is `all`, the agent will expect the external process to send it a list of percept facts representing the lastest readings of every sensor whenever any sensor reading changes. When this argument is `updates`, the agent will expect only to be told the percept facts that must be deleted, added, or changed to record just the changed sensor readings.

The `start_agent` call does the following:

- start the message handling thread - it will receive agent messages
- start the task evaluation thread that is responsible for starting and ending tasks and to (re)evaluate each task's call stack (a sequence of information about the procedure calls which is discussed below) when the *BS* changes
- start the percept handling thread and carry out a handshake with the robot side (using the address in the first argument)

The percept handler sends the message `initialise_` to the robot side to let the robot side know the address of the percept handler and that it should send back an initial list of percepts. If the robot side does not respond it will resend the message every five seconds until the robot side responds with an initial list of percepts. The percept handler will update the *BS* with these percepts and then enter a loop where it will wait for, and then process, percept messages.

The next step is to start a task that that starts with an initial `TeleoR` procedure call:

```
| ?~>  start_named_task(get_object(), collector)
```

The first argument is the initial procedure call and the second argument is the name given to this task. You will see this task name in the logger and will become more important in later multi-task agents.

This call will cause the evaluator thread to wake up, add this task to the list of tasks the evaluator manages, and evaluate its call stack and possibly send actions to the robot side.

The typical sequence of events that now happens as the program runs is that the actions that are sent to the robot side cause the robot to either change the environment or change its view of the environment which in turn changes the robots perceptions which in turn causes the robot side to send new percepts to the percept handler thread. This causes the percept handler thread to update the *BS* which in turn causes the evaluator thread to wake up and re-evaluate the call stack of the task, sending new actions to the robot side. This process continues for as long as the task is running even if it has achieved its goal.

How the robot side responds to action messages and when and what percepts are sent to the agent is determined by its programming. We will discuss that later.

We will shortly explain the semantics of this `TeleoR` program but first we show you how to set things up so that you can experiment with the program. This will involve running the program, running a tool so you can act as the robot side and running the logger so that you can see what the program is doing.

In order to do this you will need three terminal windows.

In terminal 1, you should start the Pedro inter-process communication server (if it is not already running) and the logger process, naming it `logger` (or another name of your choice), with the two OS commands:

```
pedro
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, you should 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 later.

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

```
teleor -Aagent          % Start the teleor interpreter and name the process
| ?~> [tr_eg].          % Consult the example program
success                 % System response
| ?~> logging logger.   % Turn logging of task behaviour on
success
| ?~> go().             % start the agent and task
success
```

In the program there is the following definition for the `go` action.

```
go : act()
go()  ~>
    start_agent(robo@localhost, all) ;
    start_named_task(get_object(), collector)
```

This starts the agent and the task that runs the program, giving an architecture as depicted in Figure 1. For this experiment the source of the percepts and the destination for the control actions is the robot shell process.

You can use the task name to later terminate the task using a

```
kill_task(collector)
```

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 they call already tested procedures one at a time - bottom up testing of your control program. If your consulted `TeleoR` program allowed for multiple robotic resource use, then you can have more than one task running at a time.

Immediately after `start_agent` is executed, you will see an `initialise_` message in the robot shell window - this is part of the handshake with the percept handling thread.

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

We will now step through a scenario that you can reproduce by running the programs as described above - remember to start to robot shell and logger before the agent.

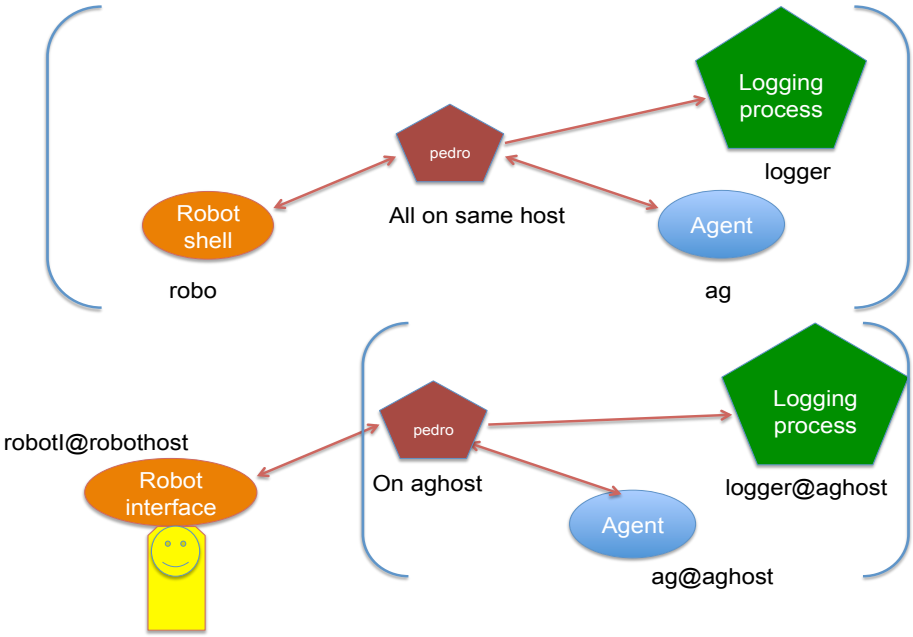


Figure 2: Emulating a robot and remote robot control

The first thing you will notice is that the agent shell will display `initialise_` in the window and if you do nothing then it will repeat this message. If you now press the `Send` button without anything in the Percepts entry box it will send the empty list of pecepts to the agent.

In the logger you will now see

```
agent::: ------ Current Percepts and Dynamic Beliefs ....
[]

agent::: collector: New call stack evaluation:
********************
get_object() - 3 fired
 get_to() - 5 fired
   [turn(left)]

Robotic actions for collector are: [turn(left)]
```

and in the robot shell you will see

```
actions(collector, [turn(left)])
```

The logger shows that the empty list of percepts arrive and then displays the result of the evaluator calculating the call stack. The third rule of `get_object` is chosen because the guard of this rule is the first guard that is true reading from the first rule. The action of this rule is a call to `get_to` and its fifth rule is chosen for the same reason. The action for this rule is the singleton list containing the robotic action `turn(left)`. This is shown in the logger and the `actions` term, which includes the task name and the list of actions is sent to the robot shell.

The call stack is actually a sequence of data structures that contain the `TeleoR` procedure call, the index of the rule chosen and the instantiation of any variables from the guard of the chosen rules that are used in the action.

Now, playing the role of the robot, you see an object on the left at 10 units distance. You simply enter `see(10, left)` in the percepts entry box in the robot shell. The agent responds by re-evaluation the call stack based on this new percept. The logger diaplays

```
agent::: ------ Current Percepts and Dynamic Beliefs ....
[see(10, left)]

agent::: collector: New call stack evaluation:
get_object() - 3 continued
 ********************
 get_to() - 4 fired
   [move(4), turn(left)]

Robotic actions for collector are: [move(4), turn(left)]
```

The call stack above the line of asterisks are unchanged in the re-evaluation (as the first true guard has not changed) but the chosen rule for `get_to` has changed because there is now a `see` percept. Now the chosen action is a combination of moving and turning. Although not shown in the logger, the entry in the call stack for `get_to` has the variable `Dir` instantiated to `left`. The robot shell displays the new robotic actions.

Now imagine something in the environment moves this object and so you might enter `see(8, right)`. Again, the agent will re-evaluate its call stack based on the new percept and the logger will show

```
agent::: ------ Current Percepts and Dynamic Beliefs ....
[see(8, right)]

agent::: collector: New call stack evaluation:
get_object() - 3 continued
 ********************
 get_to() - 4 refired
   [move(4), turn(right)]

Robotic actions for collector are: [move(4), turn(right)]
```

The same rule of `get_to` is chosen but the variable `Dir` is now instantiated to `right` causing the robot to now start turning right. The robot shell displays the updated actions.

Now you might enter the percept `see(5, centre)`. The agent will re-evaluate its call stack based on the new percept and the logger will show

```
 agent::: ------ Current Percepts and Dynamic Beliefs ....
[see(5, centre)]

agent::: collector: New call stack evaluation:
get_object() - 3 continued
 ********************
 get_to() - 3 fired
   [move(6)]

Robotic actions for collector are: [move(6)]
```

Now you might imagine the robot arriving at the object and so you enter the percept `see(0, centre)`. The logger will now display

```
  agent::: ------ Current Percepts and Dynamic Beliefs ....
[see(0, centre)]

agent::: collector: New call stack evaluation:
********************
get_object() - 2 fired
  [grab()]

Robotic actions for collector are: [grab()]
```

Now rule 2 of `get_object` is chosen and you might now enter the percepts:

```
see(0,centre), holding()
```

Note that you need to repeat the `see` percept because you are using the all percept interface.

The logger now shows

```
 agent::: ------ Current Percepts and Dynamic Beliefs ....
[holding(), see(0, centre)]

agent::: collector: New call stack evaluation:
********************
get_object() - 1 fired
  []

Robotic actions for collector are: []
```

Now the overall goal of the program has been achieved.

If this was a traditional program you might think the program would now terminate as it has achieved its goal. However this is not the case as `TeleoR` prgrams are achieve-and-maintain and so the program continues to run.

If you now imagine removing the object from the robot's grasp then you would send the empty (set of) percepts. In response to this the logger will now show the same as when you began:

```
  agent::: ------ Current Percepts and Dynamic Beliefs ....
[]

agent::: collector: New call stack evaluation:
********************
get_object() - 3 fired
 get_to() - 5 fired
   [turn(left)]

Robotic actions for collector are: [turn(left)]
```

and it would start scanning for a new object.

We now consider the part of the semantics of `TeleoR` programs necessary to explain the behaviour of this program. We will give the extended semantics of other constructs as we introduce them via the later examples.

The robot side will send lists of percepts (starting with the initial percepts) to the percept handling thread. Each time the percept handler gets a list of percepts it will update the *BS*. This will continue for as long as the agent is running.

At some point `start_task` will be called and that will cause the evaluator thread to generate an initial call stack for the task. Each call stack is a sequence of triples consisting of a procedure call, a rule number and a list of variable bindings for variables that are instantiated in the rule guard and used in the rule action. The procedure call of the first triple in the sequence is the call in `start_task`. For each subsequent triple in the sequence the procedure call is the action of the chosen rule of the previous element in the sequence.

The action of the chosen rule of the last element of the sequence is a list of robotic actions (possibly the empty list).

To compute the chosen rule for each element of the call stack the evaluator begins at the first rule and evaluates the guard. If that guard is true (possibly instantiating variables) then that rule is chosen. If the guard is false then it moves on to the second rule and repeats the process. Therefore the chosen rule is the first rule whose guard is true.

If none of the guards are true the evaluator raises an exception and the agent is terminated.

When this program is run (assuming that the percept handler has not added any percepts to the *BS*) then, as highlighted in the logger, the initial call stack will be

```
<(get_object(), 3, []), (get_to(), 5, [])>
```

When the percept handler updates the *BS* with `see(10, left)` the evaluator will build the following call stack.

```
<(get_object(), 3, []), (get_to(), 4, [left])>
```

Recall that the variable `Dir` in the guard will be instantiated to this value.

Now when the percept `see(8, right)` arrives the percept handler will replace the fact `see(10, left)` with `see(8, right)` in the *BS*. The evaluator will then build the call stack

```
<(get_object(), 3, []), (get_to(), 4, [right])>
```

We say that, upon re-evaluation of the call stack, an entry that remains the same is a *rule continuation*, an entry with the same procedure and same rule number but with a different variable instantiation is a *rule refiring* and otherwise is a *rule firing*.

So, in the example above the first entry is a continuation in each re-evaluation, the second entry in the first re-evaluation is a firing and the second entry in the second re-evaluation is a refiring.

The astute reader will notice that, for this program, it is not necessary to construct a call stack at all. Simply following the chain of chosen rules would be enough. We will see later as we add more constructs that we will need to keep track of the previous rule choices and variable instantiations - the previous call stack.

Related to semantics, Nilsson introduced two definitions that are useful for both designing and reasoning about `TeleoR` procedures: regression and completeness.

A `TeleoR` procedure has the *regression property* if the action of each rule (except the first) will normally, eventually, make an earlier guard true and it has the *completeness property* if, for a given state of the system, at least one guard is true. If a `TeleoR` procedure has these properties then it will typically make progress towards its ultimate goal.

We say "normally, eventually" because the environment might continually thwart the agent. We assume the environment is not so hostile that the agent won't be able to make progress if it tries for long enough.

As you might imagine, keeping these ideas in mind when designing `TeleoR` agents can be of great help. They have also been used to prove properties about progress given reasonable assumptions on the environment.

Later we will see examples where we slightly relax the regression property and another example where we generalize this property making it more flexible in certain applications.

## 3.2 Simple Bottle Collector: commit while and timed sequences

The file `qulog/examples/bottle_collector/bottleCollector.qlg` contains a slight extension of the example program above. In this case, the agent repeatedly finds and goes to a bottle, picks it up and delivers it to a drop. This example is supported by a Python simulation of the environment that you can interact with: creating and removing bottles and dragging them around. Instructions on running the program are in the comments near the beginning of the file.

We now update the type declarations to add `dead_centre` as a direction and a type declaration for things the agent is interested in.

```
def dir::= left | centre | dead_centre | right
def thing::= bottle | drop
```

The percepts and the robotic actions are extended.

```
percept holding(), gripper_open(), see_at(thing, int, dir),
    over_drop()

def robotic_action ::=
        open_gripper() | close_gripper() |
        move(num) | turn(dir,num)
```

We have added a speed for moving and turning that we vary as we get close to the object of interest.

The top-level `TeleoR` procedure is more complex as it needs to collect a bottle and deliver it.

```
tel collect_bottle()
collect_bottle(){
    delivered() ~> ()

    next_to_centre(drop) & holding() ~>  open_gripper()

    holding() ~>  get_next_to(drop)

    next_to_centre(bottle) & gripper_open() ~>  close_gripper()

    next_to(bottle,Dir) & gripper_open() ~>  turn(Dir,0.2)

    gripper_open() ~> get_next_to(bottle)

    true ~>  open_gripper()
    }
```

The guards are now typically relation calls that are defined in terms of the facts in the *BS*. For example

```
rel next_to(?thing,?dir)
next_to(Thing, Dir) <= see_at(Thing, Dist, Dir) & Dist < 1

rel delivered()
delivered() <= next_to(drop,_) & next_to(bottle,_) & gripper_open()
```

You will notice that the action of two rules are both calls on the `TeleoR` procedure `get_next_to` but with different arguments. This procedure is defined as follows.

```
tel get_next_to(thing)
get_next_to(Th){
    next_to_centre(Th) ~> ()

    next_to(Th,Dir) ~> turn(Dir,0.3)

    close_to(Th,_) ~> approach(Th,0.5,0.5)
    % Now very near to Th, slow right down

    near(Th,_) ~> approach(Th,1.5,0.5)
    % more slowly to achieve very_near(Th,Dir)

    see(Th,_) ~> approach(Th,2.5,0.3)
    % approach Th quickly to achieve  near(Th,_)

    true ~>  [turn(left,0.5):7,move(2):2]
    % Th not in sight, wander hoping to see it
    }
```

Again notice that the `TeleoR` procedure `approach` takes different arguments (speeds) depending on the situation.

The new construct in this procedure is the action of the last rule: a *timed sequence*. This is a list of action, time pairs and the semantics for this case is that as long as this rule continues then the turn action will be active for 7 seconds and then the move action will be active for 2 seconds and this will be repeated indefinitely (for as long as the rule is active). This gives the robot a simple wandering (searching) behaviour. If the last entry has no time component then the semantics is that the last action will be active indefinitely.

This example shows where we really need the call stack as it keeps track of any timers (delays) and the evaluator needs to be able to access all the current delays recorded in the call stack.

The objective of the procedure `approach` is to get close to a given thing.

```
tel approach(thing,num,num)
approach(Th,Fs,Ts){

    see(Th, Dir) & centre_dir(Dir) ~> move(Fs)

    see(Th,Dir) commit_while dir_or_centre(Th, Dir) ~>
```

```
        move(Fs),turn(Dir,Ts)
}
```

The second rule uses another new construct: `commit_while`. In the example the semantics is, when this rule fires, the rule will stay active within this procedure while `dir_or_centre(Th, Dir)` is true for the fired instantiation of `Dir` even if `see(Th,centre)` becomes true. In other words when this rule fires it "takes over" the procedure. The purpose of this construct is to "over achieve".

Superficially, it seems that the `commit_while` is not needed - if the second rule is chosen it will turn in the appropriate direction until the thing is seen at centre. The problem is that we will flip rules as soon as the boundary between `left`, say, and `centre`. After moving a small distance we will flip back to the second rule. This causes the robot to "flutter" along the boundry between `left` and `centre`. If you remove the `commit_while` and run the program and simulation you will see this happening. By commiting until `dead_centre` is reached it will be a while before it moves off track.

Because the `commit_while` takes over you need to be very careful getting the test right. If it's not quite right it might not give up control when it should. The definition of `dir_or_centre` covers the cases in which we want the action to continue:

```
rel dir_or_centre(thing, dir)
dir_or_centre(Th, Dir) <=
    see(Th, Dir)
dir_or_centre(Th, _) <=
    see(Th, centre)
```

If the environment moved the thing from, say, `left` to `right` then `dir_or_centre(Th, Dir)` will fail and the second rule would refire and the action would now be to turn right. On the other hand if the thing stays where it is then `dir_or_centre(Th, Dir)` will remain true until the direction becomes `dead_centre`.

### 3.3 Cooperating Bottle Collector: or_while, messages, attached QuLog actions, modified regression

The file `qulog/examples/bottle_collector/coopBottleCollector.qlg` is also a `TeleoR` control program for a bottle collecting robot but with important differences from the program just discussed. The controlled robot is

in a physical space occupied by second bottle collecting robot controlled by another agent. Collisions, even near misses, must be avoided, and the robot agents have a joint task of having their robots collect and deliver to a single drop area a specified number of bottles. When this total has been reached each robot should stop collecting bottles. Both robots are controlled using the same `TeleoR` program.

In order that each robot can be stopped when the requisite number of bottles have been collected their controlling agents communicate via the Pedro server. When each agent is launched using the `teleor` command it is given its own public name and each agent sends messages to the other using its public name.

To avoid collisions the robots need to have some means of 'seeing' each other. Minimally each robot needs to be able to sense when the other robot is 'near' and 'in front'. Each has a camera with simple image analysis software that generates the `see(thing,dir)` percepts where `thing` is a `bottle` or the `drop`. We shall assume this is now able to generate `see` percepts where `thing` is `robot`. If we are just using colour blob detection, all we need to assume is that each robot has a different colour from bottles and the drop. We could have, for example, painted each robot in such a way that the camera image processing could determine the approximate orientation of a seen robot as well as its distance and relative direction. Generally, the more sophisticated the percepts the less work the agent needs to do in determining what to do. We have deliberately kept visual processing simple to show that, when we have communication, message exchange can be used to enhance perception.

Recall that when an `TeleoR` agent is started it creates a message thread for processing messages - all messages sent to an agent are placed in the message queue of this thead. We must therefore program the message handler to convert such messages into dynamic beliefs which can be queried along with the latest percepts in `TeleoR` action rules. In addition `TeleoR` action rules have to include message send actions to other controlling agents, as well as action messages to controlled robots.

### 3.3.1 Programming the message handler

The agent message handler is programmed by defining a `QuLog` action procedure with system declared type[3]:

```
act handle_message(!message,!agent_handle)
```

[3]The `handle_message` type need not, indeed should not, be included in the agent program. Only the `QuLog` rules defining the `handle_message` action need be given.

Here `message` is a programmer defined type specific to the agent application. The `message` type, and any other user defined types it uses, defines the inter-agent communication ontology of the application.

The `TeleoR` agent system code that is executed as the outer loop code of the message handling thread calls the application specific `handle_message` `QuLog` procedure whenever it receives a ground message of type `message`. All other messages are discarded *unless* the programmer has also defined the action `handle_invalid_message` of system declared type:

```
act handle_invalid_message(!string, !agent_handle)
```

This takes a string representation of the non-message as argument.

In this application it is easy to see that the only messages are ground messages of type `message` and so there are no invalid messages. Hence we do not need to define `handle_invalid_message`. However, for purposes of illustration, we assume there might be other agents, perhaps written in a different language, that also send messages to the robotic agents. It could also be argued that when developing the application code it might be good defensive programming to assume invalid messages might be received by an agent.

In the example below this action simply displays a message to standard output. If these were real robots with the agent running on the robot then there would be no standard output and so, instead, we could send a message to a monitoring process for dispay or logging.

We start with the message type declaration.

```
def message ::= count(int) | stopped() | stopped_dir(dir)
```

The first constructor is for updating the collection count and the second and third are for conflict avoidance.

We also want to send these messages to the simulation so the messages can be displayed and to do that we add the following type declaration.

```
def display_info_t ::= display_info(message)
```

We now give the two system declared, programmer defined message handling actions.

```
handle_message(count(Count),_) ~>
    other_collected := Count
handle_message(stopped(),_) ~>
    forget([stopped_received()]);
    remember_for([stopped_received()], 4)
handle_message(stopped_dir(Dir),_) ~>
    forget([stopped_received()]);
    remember_for([stopped_dir_received(Dir)], 4)

handle_invalid_message(Message,Agent) ~>
    write_list(["Invalid message received: ",Message,nl_])
```

Two of the above rules update the belief store and so we declare appropriate dynamic relations below. Global variables are used and so are also declared below.

```
int other_collected := 0
int collected := 0

dyn stopped_received()
dyn stopped_dir_sent(dir)
dyn stopped_dir_received(dir)
```

Each action for `handle_message` forwards the received message wrapped with `display_info` to the Python `env` process which displays them. The first rule updates the count for the other robot. The next two remember a fact for 4 seconds. The addition and then removal of these facts cause the agent to pick different `TeleoR` rules in an attempt to avoid conflict. The last rule deals with invalid message messages from other agents - the other agent sees the message sent to it from this agent as invalid.

### 3.3.2 Exception handling rules

The typical structure of a `TeleoR` procedure is a linerisation of a sub-goal tree routed at the guard of the first rule. When called, the partially instantiated first rule guard is the goal of the call, and the partially instantiated guards of the later rules are sub-goals of the call goal. The leaf goals of this sub-goal tree represent initial states of the robotic agent's environment that might prevail when the procedure is invoked. They are handled by an action

rule with a guard LG towards the end of the rule body with an action that will normally bring about an environment state in which some guard G of a higher rule will hold.

However, in some applications exception situations can occur at any time during the procedure call, usually entirely outside of the agent's control. An example is a possible collision that is signalled by a percept that another robot is close. The action of such a rule should ensure there is no collision and ultimately result in an environment state in which another robot is not perceived as close. That is, the goal the action response should achieve is often just the negation of the rule guard. The issue is where to place such exception rules.

One solution is to place them at the end of the normal rules. Such a TeleoR procedure body would then take the form

```
{
 G1 ~>  A1
 ...
 Gn ~> An

 EG1 ~> RA1
 ...
 EGm ~> RAm
 }
```

where G1,...,Gn are the guards detecting the "normal" environment states which have a sub-goal ordering rooted at the guard G1 and EG1,...,EGm are the guards of the *exception* rules. The Gi guards fall into normal sub-goal tree where a later guard will not hold, so a later rule will not fire, when an earlier guard holds, and that is exactly what we want to happen.

But with exception rules we need them to fire as soon as the exception is detected. It will not generally be the case that when an exception occurs that none to the normal guards G1,...,Gn holds. To give precedence to exception handling, we will have to augment the normal guards to explicitly rule out the possibility that an exception needs to be handled. In other words we would have to include the equivalent of the conjunction

```
not EG1 & ... & not EGm
```

as part of each Gi.

Further, `EGm` is typically the worst exception and `EG1` is the least worst exception and it is usually the case that each `EGi` will explicitly include the conjunction of the negation of later exception rule guards.

In the 'logical' semantics of `TeleoR` rules each guard implicitly contains the conjunction of the negations of all earlier guards. In the example above we have had to explicitly add the conjunction of negations of exception rule guards and so, from a logical semantics point of view it would make more sense to put the exception rules first, even though their guards will not usually have a sub-goal relationship and the guard `G1`, giving the goal of each procedured call, will no longer be the guard of the first rule. For the same reason, when we put the exception rules first we would typically invert the order.

It could also be argued that from a readability point of view putting the exception rules first signals their importance. On the other hand, also from a readability point of view, the casual reader of such a procedure might not notice that the exception rules are special or that normal regression is not being followed.

Our solution is to bracket the exception rules using `>>>`, `<<<` chevron sequences (which are purely syntactic). The chrevons indicate that the rules inside the chevrons are special. They take priority over the normal sub-goal achieving rules but their guards are not sub-goals of the goal for which the procedure is invoked, and their actions are not required to 'achieve' the guard of a higher exception rule. Quite the contrary, their actions are intended to change the environment so that none of the exception guards will hold, allowing a return to the normal sub-goal regression using the rules below the `<<<`.

The revised procedure body would then have the form:

```
{
 >>>
     EGm ~> RAm
     ...
     EG1 ~> RA1
 <<<
 G1 ~> A1
 ...
 Gn ~> An
 }
```

Now, `G1,...,Gn` do not now have to explicitly mention the negation of the exception rule guards.

The guard evaluation order for the exception rules is before-after as for the regression rules. So where there is more than one exception condition that needs to be handled, the rules will typically be ordered so that the most serious exception condition is handled by the guard `EG1` of the first exception rule. Further, the guard `EGi` of an exception rule lower down the exception rule sequence will typically be a lesser exception condition, and the repair action `RA1` of the first rule could be such that it normally brings about the lower exception rule guard `EGi`. So we could have an inverted sub-goal tree of exception rule guards with actions that are intended to descend the exception sub-goal structure of regression rule guards so that eventually a normal goal achieving following the `<<<` can be fired.

### 3.3.3 The co-operating bottle collector main procedure

The top-level `TeleoR` procedure contains several new constructs.

```
tel communicating_collect_bottles(nat,agent_handle)
communicating_collect_bottles(Total,OthrAg){
    >>>
        near(robot, _)  commit_while min_time 6 ~>
            avoid_robot(OthrAg)
        % Rules between the chrevron >>> and <<< brackets
        % are exception reaction rules for handling
        % exceptional events
    <<<
    % The normal goal directed regression rules start here
    $collected + $other_collected >= Total ~> ()
    delivered() or_while min_time 5 ~>
                [turn(right,0.5):3,move(1.5)] ++
                    update_and_communicate_count(OthrAg)
    holding() & next_to_centre(drop) ~> open_gripper()
    holding() ~> get_next_to(drop)
    next_to_centre(bottle) & gripper_open() ~> close_gripper()
    next_to(bottle,Dir) & gripper_open() ~> turn(Dir,0.3)
    gripper_open() ~> get_next_to(bottle)
    true ~> open_gripper()
    }
```

As discussed in the previous section, the top-level goal of this agent procedure is the guard of the *second* rule, not the first rule as is normally the case. The guard of the first rule, `near(robot,_)`, is an *exception* condition, a condition that if all goes well will not occur. But if it does occur, we must temporarily *jump out of* the sub-goal regression behaviour of the rules below the `<<<`, to bring about a situation in which the percept `near(robot,_)` is no longer received by either robotic agent executing separate calls to `communicating_collect_bottles`.

Notice that the exception handling rule has a `min_time` component. This rule is a shorthand for

```
near(robot, _) commit_while min_time 4 ~> avoid_robot(OthrAg)
```

and instead of commiting to the rule while a test is true we commit to the rule for 4 seconds. This gives the agent 4 seconds to avoid the conflict.

The third rule uses a `or_while`. As with `commit_while`, the test is an alternative to the rule guard *after* the rule has been fired. The action of the rule will continue while the guard or the test remain inferable, *providing* no earlier rule in the procedure has an inferable guard. It normally takes a test and in that case it takes over this rule (only) while the test is true. In this case it means that the action sequence `[turn(right,0.5):3,move(1.5)]` will continue for 5 seconds providing neither of the two earlier rules can fire, that is providing the other robot is not seen as near and the sum of the counts of the delivered bottles does not equal or exceed the target total. Recall that messages are concurrently being received in the agent's message handling thread so a `count` update message from the agent controlling the other robot may cause the second rule to fire. The first rule will fire if the other robot is seen as close.

This rule also contains the `++` operator. This is a `QuLog` action that is attached to the rule action and when this rule is fired or refired this action is executed. In this case this action updates it's local count and communicates that to the other agent.

Attached actions are also used in the `TeleoR` procedure below but in a shorthand form.

```
tel avoid_robot(agent_handle)
avoid_robot(OthrAg) {

    stopped_dir_sent(OtherRobotDir) &
      stopped_dir_received(MyRobotDir)
        ~> avoid_move(MyRobotDir,OtherRobotDir)
```

```
    other_robot_is_stopped()  & near(robot,OtherRobotDir)
        ~+> send_stopped_dir_remember_sent(OtherRobotDir,OthrAg)

    true
        ~+> send_stopped(OtherAg)
    }
```

The rule operator ~+> may be used when the TeleoR action is the null action to emphasise that there is no robot action. The last rule of avoid_robot is shorthand for:

```
    true ~> () ++ send_stopped(OtherAg)
```

The robots avoid one another using the same set of avoidance rules:

```
tel avoid_move(dir,dir)
avoid_move(MyRobDir,OtherRobDir){

    centre_dir(MyRobDir) & centre_dir(OtherRobDir) ~>
                                [turn(left,0.5):2, move(1.0)]
    % Robots see each other head on, both turn left then move forward

    MyRobDir=OtherRobDir ~> [turn(MyRobDir,-0.2):2, move(1.0)]
    % Each robot sees the other on its left or right. Each turns slightly in opp
    % direction to increase divergence of their paths then move forward.

    centre_dir(MyRobDir) ~>
                  [turn(OtherRobDir,-0.4):2, move(1.0)]
    % Only this robot sees the other in centre view. Other robot will do
    % nothing, as per rule below, until it can no longer see this robot. This
    %  robot turns to point behind the other robot then moves forward.

    centre_dir(OtherRobDir) ~> ()
    % Other agent will be doing an avoid move using rule above.
    % This robot waits until other robot no longer seen.

    true ~> [turn(MyRobDir,-0.4):2, move(1.0)]
    % One sees other on its  left, the other sees it on its  right. Each
    % turns slightly away from the other robot. They then pass each other
    %  slowly. Both are using this rule.
   }
```

### 3.4 Nilsson's Simple Block Tower Builder: Cognitive control and Recursion

The file `qulog/examples/towers/towerBuilder.qlg` contains a very simple tower builder program - a direct translation of the example given by Nilsson in [3].

We give this simple example for three reasons: firstly to pay homage to Nilsson and his elegant invention of Teleor-Reactive Programming; secondly to give an example of a recursive `TeleoR` procedure; and thirdly as a precursor to the introduction of multiple tasks, sharing and alternating the use of one or more physical resources.

The tower builder problem introduced by Nilsson consists of a table containing a collection of numbered blocks and a robot arm that can pick up blocks and build a tower of numbered blocks in a given order. The environment might cover needed blocks, shuffle (partly built) towers or even help build the tower or uncover needed blocks. There is a simplifying assumption that there is enough room on the table for every block to be placed directly on the table in the area that can be reached by the robot arm.

The relatively high level robotic actions and percepts that we can assume are:

```
def robotic_action ::= pickup(block) |  put_on_block(block) |
             put_on_table()

percept holding(block), on_table(block), on(block,block)
```

In this case the emphasis of the `TeleoR` agent control program is cognitive control - knowing what block move action should be attempted next to efficiently achieve a built tower goal.

The top-level `TeleoR` procedure does not directly invoke any robotic actions. It calls subsidiary `TeleoR` procedures, and itself. It is recursive.

```
tel makeTower(list(block))
makeTower(Blocks){
    tower(Blocks) ~> ()

    stack(Blocks) & Blocks=[Block,..] ~> unpile(Block)

    Blocks=[Block1,Block2,..Rest] & tower([Block2,..Rest]) ~>
        move_to_block(Block1,Block2)

    Blocks=[Block] ~> move_to_table(Block)

    Blocks = [_,..Rest] ~>  makeTower(Rest)
    }
```

The `unpile` procedure clears blocks above a block of interest.

```
tel unpile(block)
unpile(Block){
    clear(Block) ~> () % The goal of the procedure is achieved

    on(OnBlock, Block) ~> move_to_table(OnBlock)
    }
```

`tower`, `stack` and `clear` are inferred from the `on_table` and `on` percepts using the following rules.

```
rel stack(?list(block))
% The  ? means list of labels may be given or generated
stack([Block]) <= on_table(Block)
stack([Block1,Block2,..Blocks]) <=
    on(Block1,Block2) & stack([Block2,..Blocks])

rel tower(list(block))
tower([Block,..Blocks]) <=
    not on(_,Block) & stack([Block,..Blocks])

rel clear(block)
clear(B) <= not on(_,B)
```

The basic move procedures are given below.

```
tel move_to_block(Block1:block, Block2:block)
```

```
% Moves Block1 from wherever to be on Block2
move_to_block(Block1,Block2){   % Only called if Block2 is clear

    on(Block1,Block2) ~> ()         % The goal is achieved

    holding(Block1) & clear(Block2) ~> put_on_block(Block2)

    holding(_) ~> put_on_table()

    clear(Block1)  & clear(Block2) ~> pickup(Block1)

    clear(Block2) ~> unpile(Block1)
    % Need to clear  Block1 to pick it up

    true ~> unpile(Block2)
    }

tel move_to_table(block)
move_to_table(Block){
    on_table(Block) ~> ()  % Goal is achieved

    holding(_) ~> put_on_table()

    clear(Block) ~> pickup(Block)

    true ~> unpile(Block)

    }
```

Note that `makeTower` is a recursive procedure and that `unpile` and `move_to_table` are mutually recursive.

To reinforce the differences between `TeleoR` procedures and procedures of traditional procedural code we focus on two aspects of this program.

First we consider recursion. In traditional programming a recursive call is made and eventually the child call will return and the parent will continue and eventually return itself. For `TeleoR` procedures the parent will pick a rule that calls the child `TeleoR` procedure. This is similar to traditional procedure calls. The important difference is that the child procedure never returns. Each procedure is continually monitoring the state and eventually the parent will pick a different rule and the child procedure will no longer

be active.

The second procedure we consider is `unpile`. On superficial reading you might think that the second rule will move blocks to the table. This is only partially true. Consider the case where `Block` is covered by exactly one block `OnBlock`. In this case the third rule of the procedure call `move_to_table(OnBlock)` will fire causing the arm to pick up the block. As soon as the block is picked up the first rule of `unpile` fires and so it does not complete the move to the table. For this reason it might have been better to rename `unpile` to something like `remove_blocks_above`.

You will notice that both `move_to_table` and `move_to_block` have a rule to deal will holding an unwanted block. This is partially to deal with this problem but also because the environment might rearrange the blocks in a partial tower and the block being held intended for that partial tower will now need to be put on the table.

### 3.5 Simple Multi-Task Tower Builders: Tasks, Resources

The previous example consisted of a single task that builds a single tower. What if we want to have multiple tasks each building their own tower? The first problem is if two tasks need to use the same block to build their towers. Clearly this is not going to be possible. However, assuming the blocks needed for each tower are disjoint, we must ensure that the tasks do not attempt to control the one arm at the same time. Unless we allow each task to finish its tower before starting another tower building task, which is not multi-tasking, we must fairly alternate the use of the arm between the tasks.

The arm can be thought of as a *resource*. A tower building task `T` needs to have exclusive use of the resource whenever it needs to move a block. It can then release the resource when the block move is completed, or is abandoned because that move is no longer the next thing that must be done in task `T`, due to outside interference. For example, the move of a block `B1` to lie on top of another block `B2` will be abandoned if outside interference:

- has covered `B2`,
- moved `B2` so it is no longer the top block of a partly built tower,
- covered `B1` with a block,
- helped by moving `B1` to be on top of `B2`.

In traditional programming exclusive use of a resource is often accomplished by using locks. In TeleoR, exclusive access to one or more resources is achieved in a more declarative fashion by simply specifying which procedures may be called to start a resource sharing task, and which procedures must be given exclusive access to the resources they need before a call to the procedure in some task T can proceed. The resources will be released immediately T terminates the call to the procedure, for whatever reason. The management of the resource sharing is done using a combination of extra code generated for such resource using procedures, and by the multi-tasking TeleoR evaluator. The result is round-robin fair allocation of resources between tasks, with no task starved of the resources it needs.

As an example, we use the program in the file
qulog/examples/towers/oneArmTowerBuilders.qlg
which is almost identical to the above program.

The first change is to decide which TeleoR procedures can be called to start a task in a multi-tasking context. The procedures so declared are the only ones that can be used as the top-level procedure call of a task. These declarations are used by the TeleoR compiler to generate possible call tree information related to resource use.

In this example we simply change the declaration of make_tower to

```
start_tel makeTower(list(block))
```

The only other change we need to make is to decide the granularity of the interleaving of the use of the arm resource by tasks. We do this by declaring certain procedures as atomic_tel. There is one major constraint on our choice of the atomic procedures - the *resource use constraint*. Every procedure P that has a robotic action rule, i.e. a rule with one of the actions:

```
pickup(block), put_on_block(block), put_on_table()
```

must either be declared an atomic_tel procedure, or in the call graph of the program procedures every path beginning at makeTower must pass through an atomic_tel procedure before it reaches a procedure with robotic action rules.

The maximum interleaving will be achieved by declaring as atomic_tel only those procedures that contain at least one robotic action rule and are the first such procedures on some path of the program call graph starting at makeTower.

Figure 3 shows the call graph of the tower building program. We could satisfy the resource use constraint by declaring makeTower to be atomic_tel.

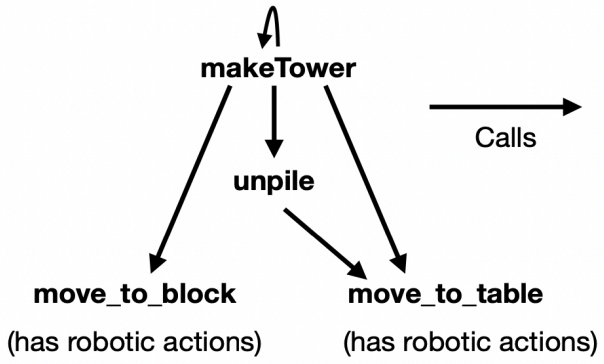


Figure 3: Call graph of single arm tower builder program

The first task to be launched will acquire the arm resource and will release it only when its tower is completed and `makeTower` fires its first rule. There will be no interleaving. For maximum interleaving we should declare `move_to_block` and `move_to_table` as `atomic_tel` procedures. Both procedures contain robotic action rules and in the call graph no path from either procedure back to `makeTower` has procedure with a robotic action rule.

```
atomic_tel move_to_block(block, block)

atomic_tel move_to_table(block)
```

In this example that is all the programmer needs to do. Now we can have multiple tasks running, each building a different tower, with reasonably fair interleaving of their use of the robot arm with every tower eventually built providing there is no persistent outside interference that repeatedly takes apart built towers before all have been built.

It should be noted that the first rule of each atomic procedure has to have the null action as the robotic action. This signals the evaluator that the task can give up the resource even if no parent procedure chooses a different rule when the goal of the atomic task has been achieved.

In this case we have only one resource and so we can think of the atomic tasks as having to grab all resources. In the next example we have multiple resources and tasks might only want to grab a subset of resources. This can lead to tasks running in parallel if they use non-overlapping resource.

The task evaluator maintains a set of *running* tasks and a queue of *waiting* tasks together with what resources they are using or want to use. When the task evaluator re-evaluates tasks it first considers the running tasks. If the call stack changes only inside the outermost atomic call (and the first rule is not fired) then the task will continue to run. Otherwise it is added to the end of the wait queue.

Once the running tasks have been processed it considers the waiting tasks in queue order. If the re-evaluation of the first task leads to a call on a atomic procedure and its resources are not being used it becomes a running task. If not it maintains its position in the wait queue possibly waiting on a new set of resources.

The evaluator continues through the remainder of the wait queue and a waiting task `T` will become running it the required resources do not overlap with any running task, or waiting task that is earlier on the wait queue than `T`. The second of these requirements provides a degree of fairness.

If you run this program using the simulator you will see that the tasks take turns building their towers. However, if you add extra blocks on top of two of the towers, you will notice that instead of the arm alternating the removing of the extra blocks from each tower, all the extra blocks are first removed from one tower before any is removed from the other. This is because `unpile` is called from within `move_to_table`, which is declared to be atomic.

For this reason we have also included
`qulog/examples/towers/oneArmTowerBuildersIterUnpile.qlg`
that uses an iterative version of `unpile` and so is not mutually recursive with `move_to_table`. It therefore produces better interleaving of the tasks when unpiling. In making this change we also added two more rules to `makeTower` while removing two rules from `move_to_block` and one from `move_to_table`. In this program the definition of `unpile` is

```
unpile(Block){
    clear(Block)  ~> ()

    top_block_above(TopBlock, Block)
        or_while holding(TopBlock) ~> move_to_table(TopBlock)
    }
```

In this version, apart from being iterative, we also use `or_while`. The reason for this is, when the rule fires, `TopBlock` will be instantiated to the current top block above `Block`. When this block is picked up as part of the move to table then, without the `or_while` the rule will refire with a different instantiation of `TopBlock`. This will cause the atomic `move_to_table` to become waiting. Another task will become running, putting this block down. With the `or_while` the task will complete the block move and only then give up control to another task. If you run this program and a variant with the `or_while` deleted you will notice by carefully observing the task colours on the arm that this is indeed the case.

Note that, semantically, `or_while` is different from using a disjunction. Consider the two rules

```
g(X) or_while w(X) ~> a(X)
```

and

```
g_or_w(X) ~> a(X)
```

where `g_or_w(X)` is the disjunction of `g` and `w`. Now consider a case where the first rule is fired with the instantiation `X = v` making `g(X)` true. In order for the second to have the same semantics we would need to call `g(X)` before `w(X)`. Now imagine that `w(v)` becomes true and at the same time `g(v)` becomes false. The first rule will continue but the second rule might, instead, refire if `X = v` is not the first instantion found that makes `w(X)` true.

### 3.6 Multi-Task, Multi-Arm, Multi-Table Tower Builders: Resource Declarations

The file `qulog/examples/towers/twoArmCoopTowerBuildersMod1.qlg` contains a tower buider that has two "home tables" where the towers are to be built with each task having a home table and with a shared table that sits between the two home tables. There are two arms with each associated with a home table and where each arm can reach both its home table and the shared table.

Now we have an additional problem we need to avoid and that is the two arms conflicting on the shared table. We avoid this by making both the tables and the arms as resources with the following type declaration.

```
def resources  == arm || table
```

If the user does not declare this type then the system uses a default declaration that is equivalent to the user making the declaration

```
def resources  := all__
```

Typically the atomic procedures will take one or more resources as arguments as in the following declaration.

```
atomic_tel move_to_block(arm,block,table,block,table)
```

In any case, if resources have been declared the `TeleoR` compiler looks at all possible call stacks of atomic procedures and determines the set of resources that might be used. If the atomic procedures have resource arguments they will be variables in the program and so the actual resources that are required are determined at runtime.

If a task needs to move a needed block from the other home table to a tower on its home table then it will first require the other arm, the other table and the shared table as resources and then the home arm, home table and the shared table.

On the other hand if the task only needs to move a block locally then it only needs the home arm and the home table leaving a task to build a tower on the other table using its home arm. We therefore get tasks running in parallel when they have non-overlapping resource needs.

We can further improve parallelism if we consider the case where one task is moving a block from the shared table to `table2` using resources `arm2, shared, table2` and another task wants to move `arm1` to the shared table. As soon as the first task starts moving towards `table2` - i.e. `tracking(arm2)` becomes true then the second task can safely use the shared table. We can accomplish this by defining the system declared, user defined relation `overlapping_resources` as follows

```
%% overlapping if the same arm is in both resources lists
overlapping_resources(Res1, Res2) <=
    arm1 in Res1 & arm1 in Res2
overlapping_resources(Res1, Res2) <=
    arm2 in Res1 & arm2 in Res2
%% if there are no arms in common then the only possible
%% overlap is because both resources included the shared
%% table and require different arms so below we check that
%% the shared table is required by both tasks. If both are
%% over their home tables but require the shared table then
%% there is an overlap
```

```
overlapping_resources(Res1, Res2) <=
    shared in Res1 & shared in Res2 &
    over_home(arm1) & over_home(arm2)
%% If one is over home and the other is over the shared table
%% but is not tracking then that task is not finished with the
%% shared table and so there is an overlap
overlapping_resources(Res1, Res2) <=
    shared in Res1 & shared in Res2 &
    over_home(arm1) & not tracking(arm2)
overlapping_resources(Res1, Res2) <=
    shared in Res1 & shared in Res2 &
    over_home(arm2) & not tracking(arm1)
```

If the user does not define this relation the system uses a default definition that is the same as if the user had made the following definition.

```
overlapping_resources(Res1, Res2)  <=
  R in Res1 & R in Res2
```

The evaluator uses this relation to determine if resources overlap and, if not, allows the tasks to run in parallel.

In the above we define `overlapping_resources` purely to improve paralellism. In `qulog/examples/towers/twoArmSlotTowerBuilders.qlg`, however, we can't use the default definition because each running task requires access to a range of slots and so overlapping resources is not based on equality but on intersection.

These programs use the system declared, user defined hook for accessing resources. In this case we have

```
  def resources_message_ ::=
      locked_resources(term) | waiting_resources(term)

resources_hook() ::
    get_active_resources(Resources) &
    get_waiting_resources(WResources)
        ~>
            locked_resources(Resources) to twoArmSim ;
            waiting_resources(WResources) to twoArmSim
```

This is so that our simulator can colour the arms and tables with the task colours to visualize which tasks are using the resources. Typically this will want to access the relations that retrieve the current active and waiting

resources as above. This hook is called by the evaluator each time it has completed its task re-evaluations.

It's up to the programmer to decide how to use this. For example, another process might be used to monitor task resource use over time.

Because this program sometimes requires transferring blocks across tables using both arms and because we wanted to increase efficiency by having tasks cooperate more when moving unwanted blocks we have made several modifications to the basic structure of the one arm tower builder.

The first modification is using the procedure `coop_move` to move unwanted blocks. Sometimes such a block is (eventually) needed by another task and so this task can help the other task by, for example, moving the block to the shared table so that the other task then only needs to do a single move.

The second modifications arises because of the more complex move sequences - instead of using an iterative unpile we have moved unpiling rules into `makeTower` and made them iterative.

The `makeTower` procedure is given below. Note that we have used `commit_while` in the rules involving `coop_move` rather than `or_while` as in `unpile` in the previous program. This is because we want each of these rules to commit to completing a cooperative move. As discussed in the next section there are some cases where the cooperative move is not completed by the task. We therefore also present two more variants of this program that avoid the problem.

```
start_tel makeTower(arm,list(block),table)
makeTower(Arm,Blocks,TowerTab){
    tower(Blocks,TowerTab) ~> ()

    Blocks = [Block, TopBlock,..Rest] & not on(_, Block) &
    tower([TopBlock,..Rest],TowerTab) ~>
        move_across_to_block(Arm,Block,TopBlock,TowerTab)

    stack(Blocks,TowerTab) & Blocks = [Block,.._Rest] &
    top_block_above(TopBlock, Block)
        commit_while holding(Arm, TopBlock) ~>
                coop_move(Arm,TopBlock,TowerTab)

    Blocks = [Block, TopTowerBlock,..Rest] &
    tower([TopTowerBlock,..Rest],TowerTab) &
    top_block_above(TopBlock, Block) &
```

```
can_reach_block(Arm,TopBlock,_)
    commit_while holding(Arm, TopBlock) ~>
            coop_move(Arm,TopBlock,TowerTab)

Blocks = [Block, TopTowerBlock,..Rest] &
tower([TopTowerBlock,..Rest],TowerTab) &
top_block_above(TopBlock, Block) &
OtherArm = other(Arm)
    commit_while holding(OtherArm, TopBlock) ~>
            coop_move(OtherArm,TopBlock,home_table(OtherArm))


Blocks=[Block] ~> move_across_to_table(Arm,Block,TowerTab)

Blocks = [_,..Rest] ~> makeTower(Arm,Rest,TowerTab)
}
```

This iterative approach turns out to simplify the overall code quite a bit but a consequence is that it does not exactly follow standard regression. For example, when the third rule fires the action does not necessarily lead directly to the guard of the first or second rule becoming true. Instead it might refire and remove another block. Eventually, however, by repeated use of this rule an earlier guard will become true (unless the environment interferes). This is a simple but useful extension of the regression property.

The problem with the above program (as discussed in the next section) is to do with a case when a recursive call on `makeTower` is made. That case is when `Blocks` is not a stack (but its tail is). This case also arises in the original Nilsson program but it's not an issue there because it isn't multi-tasking.

In the program
`qulog/examples/towers/twoArmCoopTowerBuildersMod2.qlg`
we make the recursive call only when the tail of `Blocks` is not a stack (or tower) and so we do not "return from recursion" until `Blocks` is a tower. This fixes the problem.

Another solution is to make `makeTower` iterative as well. We do this in
`qulog/examples/towers/iterTowerBuilder.qlg`
and
`qulog/examples/towers/iterTwoArmCoopTowerBuilders.qlg`

The downside of this approach is that we move even further away from standard regression and so we extend the idea of regression in a similar way

to a standard approach used to prove program termination. We define an integer valued "regression cost function" on the state that is bounded below (typically by the value for the goal state) and where the action of each rule strictly decreases this function. These programs contains comments describing the function and how each action reduces this function.

### 3.7 Atomic Procedure Behaviour

Earlier we noted that a major difference between `TeleoR` procedures and those of traditional procedural programs is that a `TeleoR` procedure call `C`, other than the initial call of a task, is always terminated because the evaluator picks a different rule in an ancestor call of `C`.

This will usually happen just before the first rule of `C`'s procedure is about to be fired because the action of the previous rule that `C` fired has resulted in receipt of an update of the agent's *BS* later rule fired intended goal has achieved. This is the call instantiated guard of the first rule of `C`'s procedure.

When this termination of `C` happens just as it is about to fire its first rule, we could think of this as a *normal* termination of `C`. But even this can happen not because the actions of the fired rules of `C` have step-by-step changed the environment until the goal of the call is inferable from the agent's *BS*, but because the environment has been updated by exogenous actions, and it is because of these events that `C`'s goal has been achieved.

This distinction is even more important when it comes to understand what atomic means for `TeleoR` procedures. In traditional procedural programming, resources will be acquired, some activity completed, and then the resources released.

In atomic `TeleoR` procedures the resources are acquired and the activity of that procedure is started. However, either because of the environment or because of the actions of this procedure, a different rule for an ancestor procedure may be chosen causing this atomic procedure to terminate. The resources will be released but it is not necessarily the case that the activity of this `TeleoR` procedure will be completed.

This distinction might come as a surprise and go against the intuition of what atomic means to traditional procedural programmers - the distinction is about completing an activity - the concept of acquiring and releasing resources is the same.

We urge the reader to carry out the following experiment involving
`qulog/examples/towers/twoArmCoopTowerBuildersMod1.qlg`
and

`qulog/examples/towers/twoArmCoopTowerBuildersMod2.qlg`
or
`qulog/examples/towers/iterTwoArmCoopTowerBuilders.qlg`.

The first of these programs exhibits this difference while the second and third behave more like how a procedural programmer might expect.

To run this experiment it helps to have the logger running to see how an ancestor procedure causes termination of an atomic procedure "before it finishes its job".

Start with `qulog/examples/towers/twoArmCoopTowerBuildersMod1.qlg` making sure you have it logging. Let the program finish building the towers. When that is done suspend the program by pressing the spacebar. It might be helpful to reduce the speed using the left arrow key. Now move block 3 to be directly under block 2 and press the spacebar again to resume the program. The arm and table change to the colour to be the colour of the task that is running with those resources acquired. This should not look surprising. What this manouver does do, however, is change the order of tasks in the wait queue at this point.

Now repeat this experiment by again moving block 3 under block 2. Now you will see a different and perhaps surprising behaviour. First task 2 will pick up block 2 and move it to the table as it wants to access block 3 for its tower. Now task 1 gets a turn and it wants to pick up block 3 and cooperatively move it to the other tower in order to clear its partial tower so that block 2 can be added. As soon as the arm picks up block 3 the colour changes signifying that task 1 becomes waiting and task 2 becomes the running task. This means task 1 did not complete the move of block 3 - it only picked it up.

Looking at the logger will explain the reason for this.

When task 1 became running (to move block 3) the call stack in the logger was

```
towers::: t1: New call stack evaluation:
makeTower(arm1, [2, 5, 9, 6], table1) - 7 continued
 ********************
 makeTower(arm1, [5, 9, 6], table1) - 3 refired
  coop_move(arm1, 3, table1) - 2 fired
   move_to_location(arm1, 3, table1, 4, table1) - 2 fired
    move_to_block(arm1, 3, table1, 4, table1) - 3 fired
      [pickup(arm1, 3, table1)]
```

```
Robotic actions for t1 are: [pickup(arm1, 3, table1)]
```

As soon as block 3 was picked up the following percepts were received.

```
towers::: Received new percept message:
[r_(holding(arm1, 3)), f_(on(3, 5))]
```

This caused a re-evaluation of the call stack yielding.

```
towers::: t1: New call stack evaluation:
********************
makeTower(arm1, [2, 5, 9, 6], table1) - 2 fired
 move_across_to_block(arm1, 2, 5, table1) - 2 fired
  coop_move(arm1, 3, table1) - 2 fired
   move_to_location(arm1, 3, table1, 4, table1) - 2 fired
    wait_choice(move_to_block(arm1, 3, table1, 4, table1))
     []

Robotic actions for t1 are: []
```

At this point a different rule of the top-level `makeTower` was chosen (rule 2 insted of rule 7). Because the atomic `move_to_block` was terminated this task stopped running before the block move was complete.

Now run the same experiment using
`qulog/examples/towers/twoArmCoopTowerBuildersMod2.qlg`

As in the first experiment, task 2 moves block 2 to the table and then task 1 gets a turn where its call stack is

```
towers::: t1: New call stack evaluation:
********************
makeTower(arm1, [2, 5, 9, 6], table1) - 3 refired
 coop_move(arm1, 3, table1) - 2 fired
  move_to_location(arm1, 3, table1, 4, table1) - 2 fired
   move_to_block(arm1, 3, table1, 4, table1) - 3 fired
     [pickup(arm1, 3, table1)]

Robotic actions for t1 are: [pickup(arm1, 3, table1)]
```

Now, however, in response to the percepts

```
towers::: Received new percept message:
[r_(holding(arm1, 3)), f_(on(3, 5))]
```

the call stack becomes

```
towers::: t1: New call stack evaluation:
makeTower(arm1, [2, 5, 9, 6], table1) - 3 continued
 coop_move(arm1, 3, table1) - 2 continued
  move_to_location(arm1, 3, table1, 4, table1) - 2 continued
   ********************
   move_to_block(arm1, 3, table1, 4, table1) - 2 fired
     [put_on_block(arm1, 4, table1)]

Robotic actions for t1 are: [put_on_block(arm1, 4, table1)]
```

The only change to the call stack is inside the atomic procedure `move_to_block` and so it continues and completes the move of block 3.

## 3.8 Initializing the Message Handler, Post Processing Percepts

The file `qulog/examples/navigation/navigate.qlg` contains a program where the robots follow paths between rooms on tracks while avoiding other robots.

In this section we discuss two system declared, user defined actions.

The first is `init_message_handler` that, if defined, is called when the message handler is first created. In this program it is used to set up subscriptions to messages from other robots and is defined below.

```
init_message_handler() ~>
  subscribe("door_status(D,Rm,S)", _);
  subscribe("finished_charging(Rb)", _);
  subscribe("path(Rb,Path,Rm)", _);
  subscribe("new_loc(Rb,Rm)", _);
  subscribe("backed_up(Rob,Rm,Dir)", _);
  subscribe("no_path(Rob)", _);
  subscribe("end_backed_up(Me)", _);
  subscribe("reserve_room(Rob, Rm)", _);
  subscribe("new_activity_room(Rom, Rm)", _);
  subscribe("joining(_)", _);
  write_list(["All Pedro subscriptions lodged",nl_]);
  ?(my_name(Me));
  joining(Me) to pedro
```

The second is `post_process_percepts`. If defined, the percept handler thread calls this action while atomically processing the percepts. It is called immediately after the handler updates the belief store. The arguments are the list of percepts that are forgotten and the list of percepts that are remembered. This gives the programmer an opportunity to carry out inferences and perform other actions based on this information.

In this program we use the percepts to infer when the robot has moved into a new room and to let other robots know. We also use it to announce when a robot is moving towards the centre of the room and is therefore no longer backed up.

The code is below.

```
  post_process_percepts(Forgets, Remembers) ~>
    check_for_room_change(Forgets, Remembers);
    check_for_backup_change(Remembers)

act check_for_room_change(!list(percept_term), !list(percept_term))
%% A robot gets a forgets close_doorway percepts and a remember through_doorway
%% percept and that signifies that the robot has just moved into a room
%% It notifies all robots about its new room
check_for_room_change(Forgets, Remembers) ::
    close_doorway(Door) in Forgets &
    through_doorway(Door) in Remembers &
    my_name(Me) & loc(Me,InRm) &
    connected(Door,InRm, OutRm, _Dir) ~>
        new_loc(Me,OutRm) to pedro
% Empty action when robot has not just passed through a door.
check_for_room_change(_,_) ~> {}


act check_for_backup_change(!list(percept_term))
check_for_backup_change(Remembers) ::
    see_centre_close() in Remembers &
    my_name(Me) &
    has_backed_up(Me, ChgrRm, _Dir) &
    reserved(Me, ChgrRm) &
    correctly_backed_up_in_room(Me, ChgrRm) ~>
        end_backed_up(Me) to pedro;
        finished_charging(Me) to pedro
check_for_backup_change(Remembers) ::
```

```
    see_centre_close() in Remembers &
    my_name(Me) & loc(Me,MyRm) &
    has_backed_up(Me, MyRm, _) ~>
        end_backed_up(Me) to pedro
check_for_backup_change(_) ~> {}
```

### 3.9 Embedded Agents

For low-end devices such as the Mindstorm EV3, we would typically want (or need) to run the agent, the percept gathering and the robotic action implementation within a single process. The folder `qulog/examples/ev3` contains support using the foreign function capabilities of Qu-Prolog and a very simple `TeleoR` program `trev3.qlg` that responds to being touched by backing away from the touch.

For embedded agents we need to define two system declared actions: `poll_sensors` and `control_device`.

In the program we have

```
  poll_sensors(P) ::
    P = [touched(Dir) :: Dir in [left,right] & is_touched(Dir)]
        ~> {}

rel is_touched(dir)
%% for the first argument 0 is left and 1 is right
is_touched(left) <= ev3_touch(0, 1)
is_touched(right) <= ev3_touch(1, 1)

control_device(OldActions, NewActions) ~>
    forall L, R { move(L, R) in OldActions ~> ev3_stop() };
    forall L, R { move(L, R) in NewActions ~>
                      ev3_run_forever(L, R) }
```

The responsibility of `poll_sensors` is to get the list of all the current sensor values as percepts. In this case we return `touched` percepts by using the foreign function interface to get the touch sensor values.

The responsibility of `control_device` is to stop the old actions and to start the new actions.

To start the agent we use `start_embedded_agent` passing in the time between polls of the sensors as, for example

```
act go()
```

```
go() ~>
    ev3_init();
    start_embedded_agent(0.1) ;
    start_task(ev3, frighten())
```

This means that the agent will poll the sensors (by calling `poll_sensors`) every 0.1 seconds

## 3.10 ROS and MQTT

In a situation such as a smart home we might be using MQTT for home sensor notifications and ROS for gathering sensor information from a robot and sending the robot control messages. In order for a `TeleoR` agent to work in such an environment we need to be able to communicate between the agent and other systems using either ROS or MQTT (or both).

In the examples folder there are two subfolders ROS and MQTT that contain example C-level interface code that is intended to run as another node in the system that communicates with the agent using Pedro and the other nodes using ROS or MQTT. The consequence of this is that each time percepts are sent to the agent two messages are required and the same goes for robotic action messages.

This approach means communication is a little slower than if, say, ROS is used for all communications but we feel this is not a problem for `TeleoR` agents for the following reasons.

The first reason is that `TeleoR` agents are not intended to be used in applications where it needs to react at a very high speed (a reflex reaction). The `TeleoR` agent is intended to control robots and other devices at a reasonably high-level, leaving the details of how this is to be achieved by low-level code, perhaps at the C-level. Even so, timing suggests that this communication is running at millisecond speeds.

The second reason is that typically some conversion between low-level sensor information and the high-level percepts is required and doing this conversion in this node effectively removes one step in communication. The same could be argued for converting robotic actions into low-level control commands.

The third reason is that this approach provides a standard interface to the `TeleoR` system and avoids having to worry about potential race conditions that might arise if ROS or MQTT were embedded in Qulog itself.

# 4 Thinking about `TeleoR` programming

When thinking about writing a `TeleoR` procedure the first step is to divide states up into sets distinguished by some properties. The next step is to order these states from the most desirable to the least desirable with the most desirable being the goal of the `TeleoR` procedure. Then for each set of states (other than the first) we determine an action that, under normal conditions, will achieve a state that is more desirable. This is the regression property.

This action might be a collection of primitive actions but when the situation is more complex it might be another `TeleoR` procedure whose goal is the more desirable state.

In doing this initial design, based on available percepts and primitive robotic actions, some rethinking of what constitutes the sets of interesting states might make designing the `TeleoR` procedure simpler. This might mean changing what constitutes the percepts and maybe even the robotic actions.

As a running example lets consider a simple block stacking problem similar to earlier examples.

We have a table that contains a collection of numbered blocks some of which might be on the table and some might be stacked on other blocks. We have an arm that has three robotic actions: `pickup(Block)` that will pick up Block provided it is uncovered, `put_on_block(Block)` that puts the held block on Block provided it is uncovered, and `put_on_table()` that puts the held block on the table. We will assume the arm will do nothing if called when the action conditions are not satisfied - e.g. no block is being held or the required block is covered.

Our task is to build a tower [1,2,3] where 1, 2 and 3 are the block numbers. 3 needs to be on the table, 2 on 3 and 1 on 2 (and nothing on 1). We assume percepts `on(Block1, Block2)`, `onTable(Block)` and `holding(Block)`.

Now lets consider dividing states up into sets with properties related to our problem. So the most desirable set of states is the one that has the property `tower([1,2,3])` where we can define the property (relation) in terms of the percepts `on` and `onTable`. Note that we don't care where on the table this tower is or where any other blocks are - the tower property only cares about the arrangement of these 3 blocks.

A slightly less desirable state would be `tower([2,3]) & holding(1)`. Applying the action `put_on_block(2)` will normally achieve the most desirable state (in which `tower([1,2,3])` is true).

A slightly less desirable state than this would be `tower([2,3]) & clear(1)` where `clear(1)` means 1 is uncovered. The action `pickup(1)` will normally achieve the state above.

Note that we say "normally". By this we mean that the environment does not itself modify the state.

For example say `tower([2,3]) & clear(1)` is true and we invoke the action `pickup(1)` but before the arm reaches block 1 the environment puts another block on top of 1. In this case the action will do nothing and we won't achieve a more desirable state.

Even less desirable states consist of variations where needed blocks are covered.

Now lets consider a first attempt at a `makeTower` `TeleoR` procedure. For moving a block from one place to another we can think of solving this by having two rules in the top-level `TeleoR` procedure - one whose action picks up the block and the other whose action is to put the block down. Since we will need to do this in several scenarios it might be better to hand off the problem to a sub-procedure as we have done below.

```
makeTower(Blocks) {
    tower(Blocks) ~> ()

    block_needed_for_tower(Block, TopBlock, Blocks) &
    clear(Block) ~>
        move_block_to_block(Block, TopBlock)

    block_needed_for_tower(Block, Blocks) &
    top_block_above(Block, TopBlock) ~>
       move_block_to_table(TopBlock)

    partial_tower_covered_by(Blocks, Block) ~>
        move_block_to_table(Block)

    tower_base(Blocks, Block) & clear(Block) ~>
        move_block_to_table(Block)

    tower_base(Blocks, Block) &
    top_block_above(Block, TopBlock) ~>
       move_block_to_table(TopBlock)
}
```

where

- relation `block_needed_for_tower(Block, TopBlock, Blocks)` is true if there is an uncovered partial tower of `Blocks` with `TopBlock` the top block of the partial tower and Block is the next block needed to extend the tower.
- `partial_tower_covered_by(Blocks, Block)` is true if a partial (or full) tower is covered by blocks and Block is the top covering block
- `top_block_above(Block, TopBlock)` is true if `TopBlock` is the top block above `Block`
- `tower_base(Blocks, Block)` is true if Block is the last element of `Blocks`

The auxliary `TeleoR` procedure are then

```
move_block_to_block(Block, ToBlock) {
    on(Block, ToBlock) ~> ()  % Done

    holding(Block) ~> put_on_block(Block)

    true ~> pickup(Block)
    }

move_block_to_table(Block) {
    on(Block, table) ~> ()

    holding(Block) ~> put_on_table()

    true ~> pickup(Block)
    }
```

To determine if these `TeleoR` procedure will solve the problem we first consider if the tower will correctly be built from any initial state in the absence of interference from the environement.

Consider a very simple case: there is only one block needed to complete the tower and it's clear. In this case rule 2 of makeTower fires causing rule 3 of `move_block_to_block` to fire and the block will be picked up. As soon as its picked up the guard of rule 2 of `makeTower` becomes false and no other guard is true. There are a few ideas we might consider to solve this problem:

1. Don't use auxilary predicates and inline the rules for `move_block_to_block` in makeTower

2. Inject a new second rule to deal with this situation

3. Change the guard to a disjunct

4. Use an `or_while` in the guard.

1. will work but has the disadvantage of miminizing code reuse and is going against a design principle that suggests breaking down a problem into a subproblem and solving that - i.e. writing a sub-procedure

For 2. we would need a rule like

```
block_needed_for_tower(Block, TopBlock, Blocks) & holding(Block) ~>
    put_on_block(TopBlock)
```

We might also need to follow this with a rule like

```
holding(Block) ~> put_on_table()
```

to deal with unwanted blocks being held.

This is not completely satisfactory as it adds complexity and reduces efficiency as there will be more reevaluating of guards.

The third idea looks promising. Lets say we also have a relation `clear_or_holding(Block)` and change the rule to

```
block_needed_for_tower(Block, TopBlock, Blocks) &
clear_or_holding(Block) ~>
      move_block_to_block(Block, TopBlock)
```

In this case this disjunction works quite well but in more complex cases it can be difficult to produce a suitable disjunction that has a reasonable intuition.

The last idea is to use the `or_while` construct as below.

```
block_needed_for_tower(Block, TopBlock, Blocks) & clear(Block)
    or_while holding(Block) ~>
        move_block_to_block(Block, TopBlock)
```

The semantics of the `or_while` is that when this rule is fired (because the guard is true but no earlier guard is true) then this rule will continue to be the active rule provided no earlier guard is true and `or_while` is true with the same binding of variables as when the rule was (re)fired.

In this case as soon as we pick `Block` up `clear(Block)` is no longer true but `holding(Block)` is and so this rule continues as the active rule.

We argue that the use of `or_while` is simpler and more intuitive than using a disjunction.

Now lets reconsider this modified `TeleoR` procedure:

```
makeTower(Blocks)
    tower(Blocks) ~> ()    % goal is achieved - do nothing

    block_needed_for_tower(Block, TopBlock, Blocks) & clear(Block)
        or_while holding(Block) ~>
            move_block_to_block(Block, TopBlock)

    block_needed_for_tower(Block, Blocks) &
    top_block_above(Block, TopBlock) ~>
       move_block_to_table(TopBlock)

    partial_tower_covered_by(Blocks, Block) ~>
        move_block_to_table(Block)

    tower_base(Blocks, Block) & clear(Block) ~>
        move_block_to_table(Block)

    tower_base(Blocks, Block) & top_block_above(Block, TopBlock) ~>
       move_block_to_table(TopBlock)
```

Provided the environment does not intefere with the execution of this procedure will the goal be achieved gived an arbitrary initial state?

First note that this procedure does not satisfy the regression property as stated earlier. In an initial state where some or all needed blocks are covered we will find we will typically repeatedly fire rule 3 to remove blocks above a needed block then, once it becomes clear, fire rule 2 and then fire rule 3 again.

However this procedure does satisfy a more sophisticated regression property in that the action associated with any fired rule will normally achieve a state that is "closer" to the desired goal state as discussed earlier.

For this discussion, we informally argue that the goal of the procedure will eventually be achieved as long as the environment does not interfere. One very undesirable state is one in which the base of the tower is not on the table and is covered by blocks. In this case the last rule fires removing

a block above the tower base block. This achieves a slightly more desirable state. Repeated use of this rule will eventually give us a state in which the tower base block is clear and so the second last rule will fire moving this block to the table again producing a more desirable state. Now, as discussed above, we will typically oscillate between rules 2 and 3 with each application improving the state either by building more of the tower or by removing blocks covering needed blocks. Eventually we will fire rule 2 to move the last block on to the tower, achieving the goal (rule 1 fires).

Above we did not discuss the need for rule 4. This rule is needed when the initial state has a covered (partial) tower. Once these covering blocks are removed this rule would not normally fire again.

The other concept we should check is completness. Recall that a `TeleoR` procedure is complete if, for any state, some rule is true. For top-level `TeleoR` procedure this is a very important property as without it we get undefined behaviour. In the implementation of teleor we raise an exception in this case. For sub-procedures like the one above we talk about completeness relative to the calling context. Typically sub-procedures are only called for states that satisfy some property and so we can assume that property to be true when deciding on the completeness of sub-procedures.

With a bit of thought we see that these procedures are complete.

So completeness tells us that some rule will fire for any given state and the regression property tells we will normally make progress towards the goal. This in turns tells us that, without inteference, we will achieve the goal from any initial state.

What about when the environment itself changes the state? This could happen in two ways:

1. The environment could help, for example, by putting a needed block on the tower; or

2. The environment could hinder, for example, by shuffling around the blocks in a partially built tower.

The beauty of `TeleoR` programming is that it (mostly) naturally deals with both cases. For 1. the environment has itself produced a more desirable state and the `TeleoR` procedure will flip to firing a rule that takes advantage of this state. For 2, the environment produces a less desirable state but the `TeleoR` procedure will flip to firing a rule to cope with this less desirable state.

So this block building robot will efficiently cope with inteference by the environment.

Later we discuss a situation where, without more careful thought in the design of TeleoR rules, we can get inefficiency or even wrong behaviour when the environment interferes.

We now discuss when commit_while might be useful to consider. The above example does not need to use commit_while so we now introduce another very simple example, similar to an eariler one, where commit_while is useful.

We have a bottle collector robot, part of whose job is to get next to a bottle so it can pick it up and take it to a depot. For this example we consider just the get_next_to_bottle TeleoR procedure. For this procedure the only percept required is see_bottle(Distance, Direction) where distance is far, near and close and the direction is left, right, centre_left, centre_right, dead_centre. At any time there will be at most one see_bottle belief - being a belief about the closest bottle. The primitive actions are to move forward at a given speed and to turn in a direction at a given speed with the turn speed being zero when the direction is a centre direction. If both actions are active the robot moves is a curved path.

As a first pass consider

```
get_next_to_bottle() {
    see_bottle(close, Direction) & is_centered(Direction) ~> ()

    see_bottle(close, Direction) ~> turn(Direction, 1)

    see_bottle(near, Direction) & is_centered(Direction) ~> move(2)

    see_bottle(near, Direction)  ~> move(2), turn(Direction, 2)

    see_bottle(far, Direction) & is_centered(Direction) ~> move(4)

    see_bottle(far, Direction)  ~> move(4), turn(Direction, 4)

    true ~> turn(left, 4)
    }
```

where is_centered(Direction) is true if Direction is centre_left, centre_right or dead_centre

The idea is when the robot is a long way from the bottle it can afford to move fast without loosing sight of the bottle but it slows down as it gets closer.

Note that the last rule is required for completeness. Now lets consider regression. What happens if there is no bottle on the table? In this case the last rule will fire and we won't make any progress. So we have to relax the idea of regression and say provided there is at least one bottle on the table then normally eventually we will achieve the goal.

This procedure is both complete and satisfies the simple regression property, well almost. Take the sceanrio when the robot does not see a bottle on the table (but there is one). In this case the last rule fires and starts turning. Eventually it will see a bottle on the left and rule 6 will fire and will start moving (and continue turning). Now rule 5 will fire as soon as we move from seeing the bottle on the left to seeing a bottle `centre_left` and it will stop turning but continue moving. Note that because we flip rules exactly when we cross the boundry from left to `centre_left` then, even if the robot tracks perfectly without any wheel slipage, we will shortly get the percept that we see the bottle on the left rather than `centre_left` and we will flip back to tuning. Shortly after that we will get the percept that we see the bottle `centre_left`. This will lead to a rapid fluttering of the robot. This is not desirable.

What we would like to do is once we start turning to centre we should commit to turning until we see the bottle dead centre. We can think of this as over achieving an earlier guard. This is where `commit_while` comes in. Consider the following variant using `commit_while`.

```
get_next_to_bottle()
    see_bottle(close, Direction) & is_centered(Direction) ~> ()

    see_bottle(close, Direction) ~> turn(Direction, 1)

    see_bottle(near, Direction) & is_centered(Direction) ~> move(2)

    see_bottle(near, Direction)
        commit_while not see_bottle(_, dead_centre) ~>
            move(2), turn(Direction, 2)

    see_bottle(far, Direction) & is_centered(Direction) ~> move(4)

    see_bottle(far, Direction)
        commit_while not see_bottle(_, dead_centre) ~>
            move(4), turn(Direction, 4)
```

```
    true ~> turn(left, 4)
```

The semantics of `commit_while` is that, once the rule is fired, we commit to the action of this rule while the `commit_while` test is true. This means it prevents the firing of any other rule in this procedure while the `commit_while` test is true. So, in the above scenario, when rule 6 fires we commit to the actions until we see the bottle dead centre. This avoid the fluttering discussed above.

So, without inteference, adding the `commit_while` improves the efficiency of the program. However, given the dramatic effect `commit_while` has on rule firing we have to be careful about environment inteference in the presence of `commit_while`. For example, consider the situation where rule 4 fires and after it fires the environment removes the bottle. The robot will continue moving and turning. Similarly, if the bottle is moved to the other side (e.g. moved from being seen on the left to being seen on the right) then the robot will continue to turn but now in the wrong direction.

The problem is the `commit_while` test it overly strong. We can fix this by defining the relation

```
rel dir_or_centre(thing, dir)
dir_or_centre(Th, Dir) <=
    see(Th, Dir1) & not Dir1 in op_dirs(Dir)
```

where, for example, `op_dirs(left) = [right, centre_right]` and changing rule 4 (and rule 6 similarly) to

```
  see_bottle(near, Direction)
        commit_while dir_or_centre(Dir) ~>
            move(2), turn(Direction, 2)
```

This example shows that we need to be careful when using `commit_while` and need to consider how we can "escape" from this commit when the environment has a negative effect. Here are three ways we can mitigate against this.

The first approach is similar to the above. Often we want to write something like

```
G commit_while C ~> Action
```

where the rule trigger G contains a part we want to be initially true and another part that we want to remain true. In other words G is rewritten as

`G1 & G2` where G1 is the triggering part and G2 is the continuing part. In that case we can write

```
G1 & G2 commit_while G2 & C ~> Action
```

and so we won't stay committed to this rule if G2 becomes false.

The second approach that works sometimes, for example, where the bottle collector needs to detect and avoid another robot is to put the collision detection/avoidance rules in the parent `TeleoR` procedure. Since the effect of `commit_while` is local to the procedure in which it is used then it will not block the detection/avoidance of the other robot.

The third approach is to make the action a `TeleoR` procedure and add rules to it that will allow recovery from environmental interference.

## 4.1 Robot Side

In the previous sections of this guide we have focused on the agent side and how to write `TeleoR` agents. In this section we look at the robot side and what the programmer is required to do in order to support the `TeleoR` agent. Most of the example programs come with a Python simulator that is behaving as both the environment and the robot side.

There are two major parts to consider when programming the robot side.

1. Collecting sensor information, converting them into percepts, and sending the percepts to the agent as a peer-to-peer Pedro message.

2. Turning robotic action messages from the agent into control actions, e.g. setting the speed of a motor or moving an arm.

We start by looking at percepts. How the programmer writes this depends on the percept interface being used. If the `all` interface is being used then, whenever the robot side decides to send percepts, it sends all the current percepts. If the `update` interface is being used the robot side sends percepts 'deltas' that we will discuss shortly.

Before getting into details we discuss the question of how often percepts should be sent independently of which interface is being used. One approach is to send percepts at a fixed time interval (as is done in the embedded agent case). Another approach is to only send percepts when "something interesting happens". Of course what interesting means depends on the application but lets consider the bottle collector example and consider the robot moving towards the bottle, sending `see` percept messages that include direction and distance.

If we send `see` percepts at regular intervals and the robot is a long way from the bottle then, probably, the agent will keep moving at the same speed until the robot gets close. A better approach might be to only send percepts when the robot gets close. In the Python simulator we use a kind of logarithmic scale. When we cross the 32 distance unit boundary we send a percept, then when we cross the 16 unit boundary we send another percept and so on. This means that when the robot is far away we only send an occasional percept but as we get closer we send percepts more frequently.

If this approach makes sense in the application this can be more efficient than a time based version as the number of messages is typically lower and the agent is not updating the belief store (and therefore re-evaluating) unnecessarily often.

A variant of, or in conjunction with, this approach is to make the percepts as high-level as possible. This means the robot side does most of the translation from sensor information to percepts leaving the agent to work with high-level percepts that more directly match the guards of `TeleoR` rules. Naturally, the choice of what the percepts should be is an application-level decision that both the agent and robot side need to conform to.

When the agent starts the percept handling thread sends an `initialise_` message to the robot side. When the robot side receives this message it will then have the address of that thread and it should use this address for sending percepts. The robot side should respond by sending an initial set of percepts consistent with the interface being used.

When the `all` interface is being used for a bottle collector we might expect the robot side to send a message like

```
[see(bottle, 0, centre)]
```

and then when the bottle is grabbed a message like

```
[see(bottle, 0, centre), holding()]
```

Note: all percepts need to be ground terms declared as percepts in the agent.

On the other hand, for the updates interface the percepts are wrapped with one of the following functors:

1. `r_` - the percept is to be remembered
2. `f_` - the percept is to be forgotten

3. `fa_` - all the percepts that match the percept pattern are to be forgotten

4. `u_` - the percept is to be updated.

For the first two, the enclosed percept must be ground and declared as a percept in the agent. The same is true for the fourth except some arguments may have a `'!'` wrapper as well. We discuss this shortly.

The enclosed percept in the third is a declared percept but may include variables in the arguments.

Note that, because the robot side is sending deltas, it needs to keep track of the current set of percepts so that it knows which wrapper to use for constructing the next list of wrapped percepts.

If we consider the bottle collector example where the robot is moving towards a bottle and assuming it sees the bottle on initialization. Then the initial percepts might be

```
[r_(see(bottle, 32, left)), r_(see(depot, 64, left))]
```

After a while we might get close enough to want to update the bottle distance and so we might want the percepts message to be

```
[u_(see(!(bottle), 16, centre))]
```

This is semantically equivalent to the message

```
[fa_(see(bottle, _, _)), r_(see(bottle, 16, centre))]
```

assuming we only produce a percept for the closest bottle.

The reason for wrapping the `bottle` argument is that we don't want to accedentially remove the `depot` percept.

Note that if there are no patterns that match an `fa_` then nothing happens and so the initial percept message could also have been

```
[u_(see(bottle, 32, left)), u_(see(depot, 64, left))]
```

but would have taken slightly longer to process.

Generally speaking, however, using `u_` rather than the semantically equivalent form produces a shorter message that is quicker to transmit and process.

It is possible for percepts to come from multiple sources but in that case the `updates` interface would be required and each source would need to be responsible for a fixed subset of percepts.

The other requirement of the robot side is to process robotic action messages from the agent such as the message

```
actions(collector, [move(4), turn(left)])
```

The message will be a list of ground terms of type `robotic_action`.

The programmer needs to decode the message string and translate the robotic actions into appropriate controls. Depending on the complexity of the message string the programmer might choose to use direct string processing or, for example, use one of the Pedro parsers supplied with the Pedro distribution. See, for example, the Python simulators or the ROS or MQTT C-level example interfaces in the `examples` folders.

Note that the robotic actions being sent are the ones that should now become active. This means that the robot side needs to keep track of the previous list of robotic actions.

Imagine that the above actions were previously sent and next the agent sends the action message

```
actions(collector, [move(2)])
```

This means that the `turn(left)` action needs to be stopped. Depending on the hardware being used, the `move` action needs to be modified or the old action stopped and the new action started.

In a multi-task application the robot side might need to keep track of the actions from each task so that the current set of actions can be correctly updated. The task argument of `actions` can be used to keep track of task spectific actions. If the robot side is interacting with several agents then the agent name in the sender address can be used for this purpose. In rare cases a combination of task and agent name might be required.

Some of the actions are *durative* (as above) and will continue unless stopped and others are *terminating* and will naturally terminate after a period of time such as an arm grasping an object. In the special case where the time to termination is effectively zero we call that action *discreet*. This distinction is application specific and the programmer needs to be careful to distinguish these cases. If an action is durative and is not present in the next set of actions it needs to be stopped. On the other hand if the action is terminating then it should only be stopped if it is still in progress.

Earlier we discussed using the tool `robot_shell.py` where the programmer can pretend to be the robot side so as to test `TeleoR` agent code. The dual of this is the supplied tool `agent_shell.py` where the programmer can pretend to be the agent. The programmer might find this tool useful when testing their implementation of the robot side. The programmer can send an `initilise_` message and see the response and then send robotic actions and see if the robot responds correctly (and sends back appropriate percepts as the robot interacts with the environment).

# References

[1] N. Carriero and D. Gelernter, Linda in context, *Communications of the ACM*. 32(4): 444-458, 1989.

[2] N. J. Nilsson, Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1, 139-158, 1994.

[3] N. J. Nilsson, Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5:99-110, 2001.

[4] Kuipers, B., Feigenbaum, E., Hart, P., Nilsson, N., Shakey: From Conception to History, *AI Magazine*, 38(1), pp. 88-103, Spring, 2017

[5] MQTT 3.1.1 Oasis Reference, 2014, At: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[6] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, Andrew Ng. ROS: an open-source Robot Operating System, 2009. At: http://www.robotics.stanford.edu/∼ang/papers/icraoss09-ROS.pdf