Robotic Agent Programming in `TeleoR`

Keith L. Clark[1] and Peter J. Robinson[2]

***Abstract*— We present an extension, `TeleoR`, of Nilsson's Teleo-Reactive (`TR`) rule based robotic agent programming language[22]. For both languages programs essentially comprise sequences of *Guard* ~>*Action* rules grouped into parameterised procedures. The *Guard* is a deductive query to a set of rapidly changing percept facts generated from the most recent sensor values. For `TR`, the *Action* is either a tuple of primitive robotic actions, to be executed in parallel, or a single call to a program procedure, which can be a recursive call, or a *BeliefStore*update. `TeleoR` has extra forms of action. The procedures encode goal (*teleo*) directed *reactive* task and sub-task behaviours of robotic agents. `TR`/`TeleoR` programs are robust and opportunity grabbing, and so are well suited to human/robot or robot/robot co-operative tasks requiring flexible behaviour.**

`TeleoR`, extends `TR` in having: types and higher order features; extra forms of action rules that temporarily inhibit other rules in the same procedure; repeatable sequences of time capped actions; wait/repeat re-start of failed actions; belief store update and message send actions linked with any rule; a flexibly typed higher order LP/FP programming language for *BeliefStore* inference; support for the high level programming of multi-tasking agents that interleave the use of subsets of a set of independent robotic resources. All the extensions were driven by application needs. The use of `QuLog` enables us to guarantee by compiler analysis that all guarded rule actions will be fully instantiated and correctly typed when sent to a robot, perhaps via a ROS interface. The focus of this paper is on the extensions for single task communicating robotic agents.

Our goal was to extend `TR` without losing the elegance and simplicity of Nilsson's language. We also wanted to be able to give the extended language a formally defined operational semantics, building upon one we had given for `TR`.

The extensions, their semantics, and their implementation were developed in parallel. A methodology we can recommend.

I. Introduction

Nilsson's Teleo-Reactive (`TR`) [22] agent programming language is a mid-level robotic agent programming language. It assumes lower level routines written in procedural programming languages such as C that do sensor interpretation, particularly for vision, and others that implement quite high level robotic control actions such as moving a jointed arm to a given location, or to be next to a recognisable object. `TR` is a language for deciding to make such an arm move, given that the object has just been 'seen', because doing so will opportunistically achieve some sub-goal of its current task.

`TR` programs are sequences of *guard*~>*action* rules clustered into parameterised procedures. The guards query, sometimes via rules defining 'interpretation' relations, a set of rapidly changing percept facts that are the agent's internal representation of the lower level sense data analysis. A rule action is: one or more robotic actions to be executed in parallel; or a call to a `TR` procedure, which can be a recursive call, or a *BeliefStore* update action. In each called procedure there is a current fired rule - always the first rule with a guard instance inferable from the current percepts. At the bottom of the call hierarchy the fired rule always has robotic actions that are dispatched to the physical devices to effect changes in the agent's environment.

In each procedure, the guard of the top rule is a percepts query that determines that the goal of a call of the procedure has been achieved, with an action that maintains the goal, often *do nothing*. Lower rules should be such that when they are fired the instantiated action should, providing the rule remains the fired rule of the call and the call is still active, *normally* bring about changes in the environment such that the guard of an earlier rule *becomes inferable*. Nilsson calls this the *regression* property.

If the action of the rule is a procedure call, the inferability of the guard of the rule, and the non-inferability of earlier rule guards, is then a context condition we can assume holds throughout the execution of the call. For each context condition of a call of a procedure, it should be the case that there is always a rule that can be fired in the procedure, often achieved by having a last default rule with guard `true`. If this is the case, and it satisfies the regression property, it is a *universal* procedure for its goal, for every call in the program.

Typically, initially called `TR` procedures query the percepts facts through several levels of defined relations. Via procedure call actions they eventually invoke `TR` procedures that directly query the percept facts and have robotic actions. This corresponds to a two tower and two thread architecture as depicted in Figure 1. For `TR` the interface between deliberation and reaction is procedure calling.

When each new batch of percepts arrives, perhaps via a ROS [26] interface, the percepts handler thread atomically updates the agent's *BeliefStore*. This triggers the `TR` evaluation thread to atomically reconsider *all* the rules that it has fired, starting with the initial procedure call and working down the call chain. If there is no change of fired rule in any procedure call the last actions continue. If there is a change in some ancestor `A` of the last procedure call this last call, and all other descendant calls of `A`, are terminated and a new call chain below `A` is unfurled until a rule with robotic actions is fired. These new actions are used to update the last set of actions, with some left unchanged, or modified (e.g. speed), or terminated or started.

[1]Keith Clark is an Honorary Professor at the Department of Information Systems and Electrical Engineering, University of Queensland, Australia `uqkclar4@uq.edu.au`

[2]Peter Robinson is with the Department of Information Systems and Electrical Engineering, University of Queensland, Australia `pjr@itee.uq.edu.au`

TR's unique operational semantics means the behaviour that a TR program encodes is robust and opportunistic. It automatically recovers from setbacks, redoing actions if need be. It skips actions, if helped. This makes TR well suited for human/robot and robot/robot collaborative applications.

Using various compiler optimisations it is possible to make this reconsideration of fired rules very fast, of the order of milliseconds, because the re-evaluation of rule guards in particular calls can often be safely skipped. Even when this is not the case, real robotic devices, often being mechanical, run very slowly compared with processor speeds so the sensor data transmission rate will often be such that all reconsideration can be concluded before the next sense data arrives. As a safeguard, the architecture ensures that all reconsideration of guards is concluded before the percepts handler gets to update the *BeliefStore*. This is because the re-evaluation response to new percepts is an atomic call within the evaluator thread. This means that the guard re-evaluation of each procedure call sees the same *BeliefStore* state. In the worst case some sense data may be lost. This should only result in a tens of milliseconds response delay on the assumption that the next sense data reading is similar to the lost one.

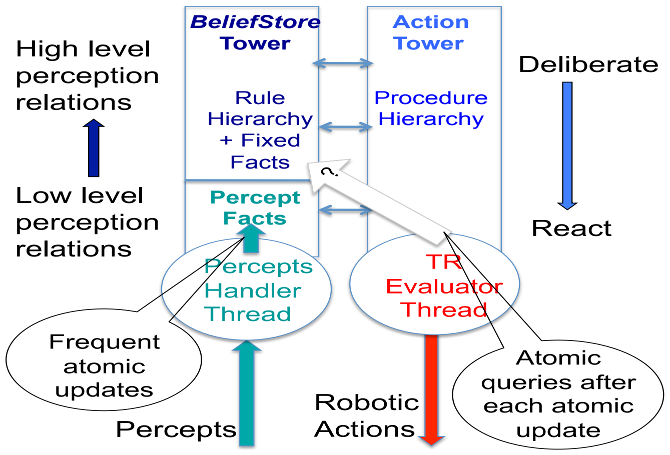

Fig. 1. Double Tower Architecture

The reconsideration down the call chain gives TR its a unique operation semantics. Procedure calls remain active even when the action of their fired rule was a procedure call. They have no stop or exit action. Procedures do not terminate themselves, they get terminated when an ancestor fires a different rule instance. This operational semantics evolved out of Nilsson's work on Shakey [21], particularly the triangular representation of Shakey's plans, with influences from Brooks's subsumption concept in his robot behaviour language [3].

Each modification or extension in TeleoR was motivated by an application need and we will motivate most of them by identifying shortcomings in an example TR program. The extensions are:

- procedures and the *BeliefStore* language are typed and higher order. This enables compile time guarantees that no guard inference will hit a runtime error due to incorrect use of a primitive, and that the robotic actions sent out to robots will be correctly typed and fully instantiated, a very important safeguard. The *BeliefStore* language is the typed and moded LP/FP language QuLog[8]. Giving details of QuLog is beyond the scope of this paper.
- *timed sequence actions* that are sequences of time limited durative actions, including procedure calls, for micro-behaviours where the actions do not achieve a sub-goal that can be tested by a percept query, but calibration can give us a good estimated time of when the sub-goal should be achieved.
- *wait/repeat* actions that may be repeated when they have not resulted in the firing of another rule after a specified wait time, indicating a possible jammed device.
- two new forms of rule that, when fired, temporarily inhibit the firing of rules *below* and rules *above* in the same procedure call, which can be combined:
  - a **while** rule used to delay the *re-achieving* of the guard of the rule when fired, in case the guard becomes false, and to allow its action to continue *while* a *continuation* query in inferable,
  - a dual **until** rule, which allows a rule action to continue in order to *over-achieve* the guard of a rule above by continuation of the rule's action *until* some condition holds and the rule's guard remains inferable.
- the ability to link *BeliefStore* updates, and message send actions to other robotic agents, with any rule action. These may not be executed immediately. They are executed when the robotic actions finally determined by the rule's use are executed.

If an agent can send messages it must also be able to receive them. To achieve this we add another thread in the base agent architecture, a message handling thread, which is the public interface of the agent. Such a thread can also handle queries, and task start and task terminate requests. The extended architecture is depicted in Figure 2. All incoming messages are handled by the message handler and both it and the percepts handler atomically update the agent's *BeliefStore*, the former by adding or removing told facts. Both told facts and percepts may be queried in rule guards allowing for the smooth integration of the two sources of data. The told facts can be percepts of other robotic agents.

Extra capabilities such as SLAM or abductive reasoning can be added as extra threads accessing and updating the agents belief store, using a common ontology. The outputs of such reasoning threads, and new beliefs added due to incoming messages, incrementally affect control behaviour. Earlier rules in procedures that query the facts that may be added by the extra threads become fireable and take over from later default rules that just deductively query the percept facts.

Pedro [24] is a publish/subscribe and addressed mes-

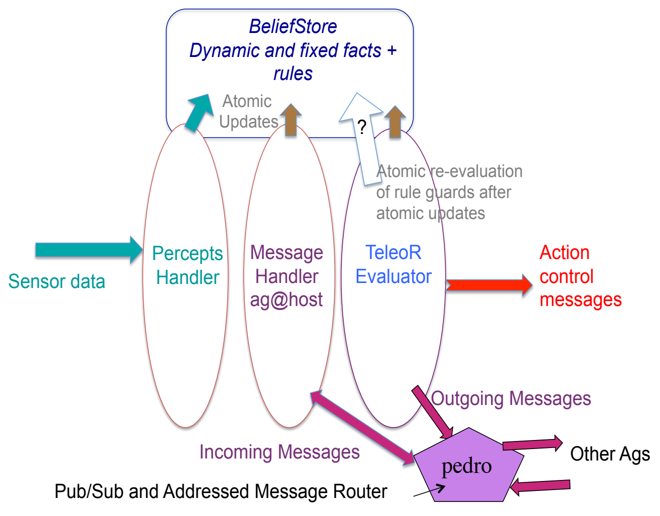

Fig. 2. Three Thread Communicating TeleoR Agent Architecture

sage router using Prolog technology. When an agent process is launched it registers a host unique name such as `collector1` with a Pedro server on the same or another host. It can then be sent messages using a Pedro handle of the form `collector1@HostName` and send messages to other agents using similar email style names if they are registered with the same Pedro server. An agent can lodge subscriptions with the server that are of the form `Ptn::Test` where `Test` is a Prolog query using only Pedro supported primitive relations. An agent can also publish a notification as a message term `Notify`. It will be routed to all agents with a current subscription `Ptn::Test` such that `Notify` matches `Ptn` and the linked `Test`, which typically tests values of variables in `Ptn` bound by the match, is inferable by Pedro.

After the Related Work Section, we describe Nilsson's `TR` language. We give a summary of `TR`'s simple rule syntax (slightly modified from that of [22]), the guidelines for how one writes a `TR` program, and further detail of `TR`'s unique operation semantics via an informal algorithm for evaluating a `TR` procedure call. A straightforward elaboration also applies to a `TeleoR` procedure call for a single task robotic agent. We exemplify the `TR` language by giving a program for an agent controlling one robot to collect bottles in an obstacle free space.

We then introduce `TeleoR` and its more elaborate syntax. We exemply its use by modifying the example `TR` program to a program for use by each of two robot controlling agents with a joint task of collecting a given number of bottles in the same space. The agents communicate so that they both 'know' when the joint total is reached. They also communicate to compensate for poor visual processing capabilities, in order to avoid collisions with minimal deviation from current paths.

The `TeleoR` language we present in this paper enables us to program single task, goal directed, robust communicating robotic agents with significant extra behaviour control than is possible, or as transparently programmed, in the original `TR` language.

We assume some familiarity with logic programming [19], robot behavioural programming [15], particularly with an AI component [25], and guarded action rules [10], [13].

II. Related Work

A comprehensive survey of extensions and applications of the teleo-reactive paradigm is given in [20].

In [1] a quite elaborate agent architecture is described that makes use of `TR` procedures represented as trees. There is a fork in the tree when there are different ways of achieving the guard at the fork. Special *and* nodes signal that several sub-gaols need to be achieved and may be achieved in any order, allowing for a more flexible rule firing strategy.

A extension of TR programs called TR+ is described in [32]. It has constructs for indicating parallel or sequential execution of actions. In addition different logical operators are used to indicate sequential or parallel evaluation of the components of rule guard, and the frequency with which a rule guard should be evaluated can be specified. The language is implemented on a network of computers using PVM. It is used as part of a robot control architecture that also has components for path planning and localisation.

[5] is an action skill representation system with the skills invoked in response to perceptions. Skills can invoke subskills which can be executed sequentially and conditionally. The skills are goal directed and reactive.

[16] presents a language for cooperative control that is typed, uses Dijstra style guarded commands, and has communication between control processes. It has a formal semantics but no concept of goals and sub-goals. [11] uses structured Engish to specify a required robotic behaviour as a set of action rules. This is mapped into linear temporal logic from which a control program is generated.

[28], [17], [9], [14], [12] and [30] present logic based approaches to programming software agents that either have been (e.g. [18] and [9]), or could be used for robotic agents with varying degrees of efficiency. None appear to offer compile time guarantees of type and mode safe inference, of type correct and ground actions, or optimised implementation avoiding unnecessary inference.

III. Nilsson's TR programs

`TR` programs comprise a set of optionally parameterised procedures each comprising a sequence of `guard ~> action` rules. A procedure `p` with k parameters has the form

```
p(X₁,..,Xₖ){
    G₁  ~> A₁     % Read ~> as do
    .
    .
    Gₙ  ~> Aₙ
    }
```

The order of the rules is important. Higher rules have priority. So the implicit guard of the i^{th} rule is

$$G_i \ \& \ \text{not} \ \exists \ G_{i-1} \ \& \ \ldots \ \text{not} \ \exists \ G_1$$

where the $\exists$ indicates existential quantification with respect to all the variables of the guard except the procedure parameters $\mathtt{X}_1,..,\mathtt{X}_k$.

When a TR procedure is called, either as the initial procedure of some task or as a sub-task action, all the parameters $\mathtt{X}_1,..,\mathtt{X}_k$ are given ground (variable free) values. This results in a sequence of partially instantiated rules $\mathbf{G}_i$ ~>$\mathbf{A}_i$ in which the parameter variables $\mathtt{X}_1,..,\mathtt{X}_k$ are replaced by their values.

Robotic actions can be *durative* - often continuing indefinitely unless explicitly changed or stopped, such as moving forward at a certain speed - or they are *discrete* (aka *ballistic*) - usually short duration action that cannot be modified or prematurely stopped - such as opening a gripper or sounding a beep. Opening a gripper would be durative if it could be stopped during execution.

1) Example TR *procedures:* Here are two procedures from a bottle collection control program for a robot with a camera with very simple image processing, and a gripper with sensors for detecting when it is open and when holding something. We use the Prolog convention that alphanumeric names beginning with an upper case letter or underscore are variable names. Underscore on its own is the anonymous variable. Repeated occurrences of _ denote different unnamed variables. Comments are preceded by %.

```
get_next_to(Th){        % Used when Th is bottle or drop
    at(Th) ~> () % Goal of the call achieved, do nothing
    Th=bottle&next_to(bottle,Dir) ~>
                                  turn(Dir,0.1)
  % Dir is left or right. Turn slowly to get bottle in centre view
    close(Th,Dist)&forward_speed(Th,Dist,Fs)
                           ~> approach(Th,Fs,0.2)
     % Very near to Th, approach with gradually slowing speed
    near(Th) ~> approach(Th,0.5,0.2)
                                  % near(Th) holds, approach
        % at constant slow speed to achieve close_to(Th,Dist)
    see(Th,_) ~> approach(Th,1.5,0.1)
  %  see(Th,_) holds, approach Th quickly to achieve near(Th)
    true ~> turn(left,0.5)
                       % Th not in sight, turn hoping to see it
    }
approach(Th,Fs,Ts){
           % Will only be called when see(Th,..) is inferable
    see(Th,centre) ~> move(Fs)
                        % whilst see(Th,centre), move forward
    see(Th,Dir) ~> move(Fs),turn(Dir,Ts)
                  % Parallel actions, swerve in  Dir direction
                           % to get Th back into centre view
    }
```

move(Speed), turn(Dir,Speed) are durative robotic actions, actions such as open_gripper and close_gripper are discrete robotic actions. at(Th), close(Th,Dist), near(Th) and see(Th,Dir) are defined relations that query changing percept facts of the form see_colour_blob(Col,Size,Dir), as well as fixed facts. For example, see is defined as

```
see(Th,Dir) <=
   colour(Th,Col) & see_colour_blob(Col,_,Dir)
```

at, near and close are similarly defined but they ignore the Dir argument and have an inequality condition on the Size argument of see_colour_blob. This is the size of the bounding rectangle of a dense array of pixels (a blob) of the given colour. close requires the area to be larger than does near. Dir is an indication of whether it is on the left, centre or right side of the camera image. We shall assume that different things of interest have different colours, and that each class of thing has the same size. This is how we can 'interpret' a blob of colour of a certain size as a particular thing at a certain distance away. Comments in the two procedures give further explanation.

2) Universal procedures: Both the above procedures are universal procedures for their goals. get_next_to always has a rule that can be fired as it has a last default rule with guard true. approach similarly always has a rule that can be fired assuming it is always called, as from get_next_to, when see(Th,_) is inferable. The rules of both procedures also satisfy the regression property.

As an example, see(Th,_) should normally become inferable when see(Th,_) is not inferable and the durative action turn(left,0.5) is executed. This is assuming Th will be within camera range of the mobile robot and there are no obstacles that might block its seeing Th. Similarly, the concurrent actions move(Fs),turn(Dir,Ts), assuming both Fs and Ts are positive, should normally result in see(Th,centre) being inferable from future percepts when it is not inferable from current percepts but see(Th,Dir), where Dir≠center, is. The *normally* caveat is because the agent might be continually thwarted by outside interference (an approached bottle gets moved), or because actions of faulty robotic devices fail (turning fails), or, because of environment conditions, are not effective (oily surface causes wheels to slip).

The get_next_to procedure has a special rule for Th=bottle. This is because at(bottle) is inferable only if see(bottle,centre) is inferable from the latest facts for the see_colour_blob percept, whereas at(drop) is inferable even if the centre line of the blob of blue does not lie in the range -10 to +10 degrees of the forward direction of the camera, the range mapped into the centre direction. We need the bottle to be seen more or less head on so that a close_gripper action is more likely to succeed.

No matter which of the get_next_to rules is fired first, it should normally progress one at a time up the rule hierarchy. It will skip rule 4 if rule 5 has been fired and someone quickly moves the bottle to be close to the robot. Conversely, if the bottle is then moved so as to be out of camera view, the procedure will immediately revert to firing its last rule to locate the moved bottle, or a different bottle, by turning on the spot.

IV. Informal TR Task Evaluation Algorithm

To make more precise the unusual operational semantics of both TR and TeleoR, we give the following 8 step informal evaluation algorithm for a start procedure call $TaskCall$.

$FrdRules$ is the set of indexed procedure calls. Each element of $FrdRules$ is a 4-tuple of the form $(Dp, Call, R, \theta)$ where $Dp - 1$ is the number of intermediary procedure calls between $Call$ and $TaskCall$, R is the number of the partially instantiated rule of the procedure for $Call$ that was last fired, and θ is the set of generated bindings for all the variables of the action of that rule produced by the inference of the rule's guard from the *BeliefStore*. Dp is the index of the tuple.

```
1) $LActs := \{\}; FrdRules := \{\}; Index := 1;$
   $Call := TaskCall.$
2) $Index > MaxDp$: A **call-depth-reached** failure.
3) Evaluate the guards for the rules for $Call$, in turn, to
   find *first* rule $\boldsymbol{G} \sim> \boldsymbol{A}$, number $R$, with an inferable
   guard, with $\theta$ being the *first* returned answer substitu-
   tion for variables of $\boldsymbol{A}$.
   Add $(Index, Call, R, \theta)$ to $FrdRules$.
   If no such rule: signal a **no-fireable-rule** failure.
4) If $\boldsymbol{A}\theta$ is a procedure call.
   $Call := \boldsymbol{A}\theta$; $Index := Index + 1$; Go to step 2.
5) Else $\boldsymbol{A}\theta$ is a tuple of primitive actions.
   Compute controls $CActs$ to change $Acts$ to $\boldsymbol{A}\theta$;
   Execute $CActs$; $LActs$:=$\boldsymbol{A}\theta$.
6) Wait for a *BeliefStore* update. On update resume.
   Index:=1.
7) If we can determine that rule $R$ of $Call$, where
   $(Index, Call, R, \theta)$ in $FrdRules$, *must continue* as
   the fired rule of $Call$ with firing substitution $\theta$, perhaps
   without re-evaluating guards.
   If $Index = \#FrdRules$ goto step 6
   else $Index := Index + 1$; repeat step 7.
8) Otherwise, evaluate the guards for the rules for $Call$,
   in turn, to find *first* rule $\boldsymbol{G} \sim> \boldsymbol{A}$, number $R'$, with an
   inferable guard, with $\theta'$ being the *first* returned answer
   substitution for variables of $\boldsymbol{A}$.
     a) No such rule: signal a **no-fireable-rule** failure.
     b) If $R' = R$ and $\theta' = \theta$ (same rule firing for $Call$):
        If $Index = \#FrdRules$ goto step 6
        else $Index := Index + 1$; goto step 7.
     c) If $R' \neq R$ or $\theta' \neq \theta$ (new rule firing for $Call$)
        $FrdRules := \{(Dp, N, C, \psi) \mid$
        $(Dp, N, C, \psi) \in FrdRules \wedge Dp < Index\}$
        $\cup \{(Index, Call, R', \theta')\}$; Go to step 4.
```

Steps 1 to 5 of the algorithm will effectively generate a call stack of procedure call descendants of $TaskCall$ using the initial state of the *BeliefStore*. The last entry in $FrdRules$, with index $\#FrdRules$, will record the firing of a rule with primitive actions, control actions for which will be executed by step 5.

A more formal state transition operation semantics of `TR` programs is given in Chapter 7 of [7]. It is elaborated in a later chapter of the book to make more precise the continuation test of step 7 of the above algorithm, which is necessary to give the operational semantics of `TeleoR`. [7] also gives a non-algorithmic state transition semantics, and identifies optimisations that can be made if we pre-compute the percept dependencies of the guards of each procedure rule, both negative and positive.

Another optimisation is to have the relatively low level defined relations such as `close`, `near` etc have all their instances inferred each time the percepts are updated. This saves repeated use of their rules, but also helps regarding the optimisations to avoid re-evaluating rule guards. This is explained in [7]. The percent handler's inference and remembering of all instances of these rule defined relations is a special form of tabling [29]. In `TeleoR` we signal this should be done by declaring these rule defined relations as `percept` relations. All such relations must be independent of one another and not recursively defined.

V. Collaborating bottle collecting agents programmed in `TeleoR`

A. `TeleoR` *procedure syntax*

The most general form of a `TeleoR` procedure is:

```
p:(t₁,..,tₖ)  % declaration of the argument types of p
p(X₁,..,Xₖ){
  G₁ while WC₁ min WT₁ until UC₁ min UT₁ ~> A₁
  .
  .
  Gₙ while WCₙ min WTₙ until UCₙ min UTₙ ~> Aₙ
  }
```

where rule components are dropped if they contain vacuous constraints, such as a `WC` that is `true`, or a `min` time that is `0`. Here, each $\mathtt{A}_i$ is one of: a tuple of robotic actions, a single call to a `TeleoR` procedure, a timed action sequence, a `wait T repeat n` action.

We give the key procedures of this program as well as the type definitions and declarations we need for type safety, ground action guarantees, and optimised compilation. The main task procedure is used by each of two agents controlling their own robot. It has two arguments, `Total`, an integer number of bottles the two agents must have their robots collaboratively collect, and `OthrAg`, the Pedro handle of the collaborating agent for sending it messages. These messages will automatically go to `OthrAg`'s message handling thread. When this thread receives an `update` message it will atomically update its `count` value. So `count` will be updated by two threads - the evaluator and the message handler.

```
dir::= left | centre | right | dead_centre
thing::= bottle | drop | robot
col::=green | blue | brown | amber | yellow
                                % Enumerated type defs
percepts gripper_open, holding,
         see_colour_blob:(col,num,dir)
                        % The sensed percepts and their types
durative move:(num), turn:(dir,num)
                    % Actions that may be altered or stopped
discrete open_gripper:(), close_gripper:()
                                     % Ballistic actions
int collected:=0
 % Syntactic sugar for an updatable collected(0) belief fact
belief colour:(thing,col)
    % Type declaration for another updatable belief predicate
```

```
colour(bottle,green)
colour(drop,blue)
percepts near:(thing),close:(thing),
        see(thing,dir),next_to:(thing, dir)
    % Percept decls for rule defined percepts (rules not given)
    % Instances inferred by percepts handler, on each update

com_collect_bottles:(int,pedro_handle)
com_collect_bottles(Total,OthrAg){
     $collected>=Total ~> ()
            % $collected is current value in collected belief
     delivered while min 8 ~>
          turn(right,0.5) for 3;
          forward_avoiding(1.5,centre,0)
          ++ collected +:= 1;update  to OthrAg
% For 8 secs inhibit rules below, even though delivered will
% not be inferable soon after the turn starts, to get robot away
% from drop. Also do BeliefStore update and a message send
     holding ~> deliver_bottle
     true ~> get_bottle
     }
at: (thing)      % Type/mode decl. for defined relation
at(drop) <= next_to(drop,_)
at(bottle) <= next_to(drop,centre)

delivered <=
      at(drop) & at(bottle) & gripper_open

forward_avoiding:(num,dir,num,pedro_handle)
forward_avoiding(Fs,Dir,Ts,OthrAg){
   not near(robot) ~>
          move(Fs),turn(Dir,Ts)
   see(robot,Dir) ~>
         wait_to_do_avoid_move(Dir)
         ++ stopped(Dir) to OthrAg
    % Stop robot, tell other agent seen direction of its robot
   }
wait_to_do_avoid_move: dir,pedro_handle
wait_to_do_avoid_move(Dir,OthrAg){
   see(robot,CurDir) & othr_stopped(MyDir) &
      avoid_move(CurDir,Myir,Fs,TDir,Ts) ~>
            move(Fs), turn(TDir,Ts)
   see(robot,CurDir) & CurDir\=Dir ~>
         () ++ stopped(CurDir) to OthrAg
   }
avoid_move: (dir,dir,num,dir,num)
avoid_move(left,left,0.5,right,0.1)
% Move slowly in slight arc to right when see other robot on left
...                         % Facts giving other avoid moves

approach:(thing,num,num)
approach(Th,Fs,Ts){
   see(Th,centre) ~>
          forward_avoiding(Fs,centre,0)
   see(Th,Dir) while see(Th,centre) until
      see(Th,dead_center) ~>
          forward_avoiding(Fs,Dir,Ts)
      % When 2nd rule fired it inhibits firing of rule above
            % until Th is seen dead_centre, not just centre
     }
get_bottle{
   holding ~> ()
   next_to(bottle,centre) & gripper_open ~>
            close_gripper wait 3 repeat 2
 % Repeat twice at 3 sec. intervals if no observable effect,
     next_to(bottle,centre) ~>
             open_gripper wait 3 repeat 2
 % Open the gripper to be ready to grab
     true ~> get_next_to(bottle)
     }
```

colour is declared as a **belief** relation so that it may be updated whilst the agent is executing. This means that if we want it to collect brown bottles the same size as green bottles we can send it a tell or inform message containing the fact colour(bottle,brown). Providing we have programmed the message handler to accept such messages and to add the fact they contain to the *BeliefStore*if not already present, the agent controlled robot will, immediately after message receipt, start collecting brown bottles as well as green ones.

The collection task procedure has the goal of getting the collected count at or above Total, an argument to the procedure. The next rule is a **while** rule that inhibits the firing of rules below for 8 seconds as it executes a timed sequence.

while and **until** rules both temporarily inhibit the firing of other rules of the same procedure call and are reminiscent of the inhibition concept in the Behavioural Language of Brooks [3]. However, inhibiting rules were not added to TeleoR because inhibition is in the Behavioural Language. A UQ colleague Ian Hayes had the need of a form of **until** rule to allow the semantically clean programming of a safety critical systems test case. We weakened his **until** and added the dual **while** rule. There is a combined **while/until** rule that approximates the semantics of Haye's **until**. Deciding on the pros and cons of different rule form extensions was considerably helped by first formulating their state transition semantics.

A **while/until** with a **min** T condition makes any durative actions of the rule locally ballistic for T seconds. No other rule of the procedure call can be fired for T seconds. Of course, firing of a different rule in an ancestor call may result in termination or modification of these durative actions.

A simple **while** rule, which has the form

G **while** C ~> A

temporarily prevents the re-achieving of its guard *while* the condition C remains inferable, even though the guard G may *no longer be inferable*.

An simple **until** rule, which has the form

G **until** U ~> A

allows over-achieving of the guard of a rule above, providing its guard G *remains inferable*, *until* the condition U becomes inferable. Its purpose is to prevent a too early need to re-achieve the guard above. In the above use, when the centre line of the colour blob of the approached Th is no longer in the range -10 to +10 degrees of the centre of the camera image, the swerve correction is continued until this line falls inside -5 to +5, which is the dead_centre range. This prevents too much oscillation between the firing of rules 1 and 2 of the approach procedure.

The purpose of the timed sequence of the **while** rule is to turn the robot through roughly 180 degrees and to move it away from the drop to get out of the way of the other robot. We need to use a **while** rule as very soon after the robot starts turning delivered will no longer be inferable. With no **while** condition, as soon as that happened the last rule of the procedure would fire turning the robot next to the drop. We want to move it away before it starts turning looking for a bottle.

The **while** rule does *not* inhibit the firing of the first rule of the procedure. So, whilst this robot is moving away from the drop the other agent may have its robot leave a bottle at, say, the other side of the drop area. It will then send a update message to this agent. If this results in the message handler increasing collected to Total, the first rule will immediately fire and the robot will be stopped in its tracks. The other robot will have been stopped next to its just delivered bottle.

The **wait/repeat** rules are used in case the grippers get stuck in the open or closed position. Re-doing the action sends another signal to the motor and may free the grippers. After the third attempt the TeleoR system adds an action_failure belief to the agent's *BeliefStore* identifying the problem action. This can be 'caught' by an outer procedure that is the one called to start a bottle collection task. It has a rule for responding to such beliefs when they appear. An example is

```
bottle_task_wrapper:int,pedro_handle,
                          pedro_handle
bottle_task_wrapper(Total,OthrAg,Helper){
     action_failure(open_gripper) ~> ()
          ++ needs_repair(gripper) to Helper
     ...
     true ~>
          com_collect_bottles(Total,OthrAg)
     }
```

The first rule sends a message to a helper person as soon as an action_failure(open_gripper) fact is added to the *BeliefStore* by the run-time system. If and when it is fixed - the gripper is oiled, say - the person sends a message, say repaired(gripper) to the agent. Its message handler should respond to that by removing the system inserted failure belief.

This outer TeleoR procedure call has co-operatively achieved the fixing of a fault by asking for assistance. The call will immediately respond to the removal of the failure fact by switching to firing the rule that calls the collection procedure. If the robot is still next to a bottle with its gripper open, a re-called get_bottle will try to grab the bottle using the oiled gripper. The collection task has resumed at the point the **wait/repeat** failed.

The get_next_to procedure is more or less as given earlier except that we would define forward_speed as a function and write its second rule as

```
close(Th) ~>
     approach(Th,forward_speed(Th,Dist),0.2)
```

The above program has an agent A1 communicate the relative direction in which its robot R1 sees the other robot R2 when R2 is seen as near. It does this using a stopped(Dir) message which tells the other agent A2 that R1 is temporarily stationary and sees A2's robot in the Dir relative direction. The auxiliary procedure wait_to_do_avoid_move will determine an appropriate avoidance move if similar relative direction information in a stopped message is received by A1 from A2, and converted into an other_stopped:dir belief. This belief is looked for in the guard of its first rule. The belief will not be present if the other robot is actually moving away from or across the path of the stopped robot R1, as R1 will then not be in the field of view of R2's camera. In that case R1 will eventually not see R2 as near, and the forward_avoiding procedure will again fire its first rule to start R1 moving again. This is *providing* an ancestor procedure call has not fired a different rule terminating the forward_avoiding call. This could happen if the other robot R2 is at the drop delivering the final bottle. The received update message will cause R1's collected count to be increased so that there is an immediate firing of the first rule of com_collect_bottles.

Each agent's message handler must handle stopped and update messages. It is implemented using QuLog action rules. It responds to the receipt of a stopped(Dir) message by doing the update actions

forget othr_stopped(_);
remember othr_stopped(Dir) for 2

after having checked that Dir is an atom of the dir type with a run-time type test type(Dir,dir).

This message response removes any current othr_stopped belief, and adds an othr_stopped(Dir) fact to the agent's *BeliefStore*. This is automatically removed, unless updated, after 2 seconds. This is done because after 2 seconds the agents will have started their avoiding moves, or the near collision was a false alarm and the other robot seen as near was not approaching the stopped one. In the latter case, the other agent did not have near(robot) percept in its *BeliefStore* whilst this agent's robot was stopped. The automatic removal of the remembered othr_stopped means it will not be in the *BeliefStore* should a near collision occur later. The message handler responds to an update message by executing collected +:= 1.

As an alternative to communicating seen relative directions, we could have the robots painted with their front half in one colour and back half another colour. Then the left/right juxtaposition of the two robot colours, and their relative sizes, will enable us to generate from the image processing software `see_colour_blobs(Col1,Size1,Col2,Size2,Dir)` percepts. From these, we can infer crude orientation information about a seen robot. For example, if `Col1` is the front colour and its size is quite small, the seen robot is essentially moving away, but not directly away.

Another solution to the collision avoidance problem, where we try to divert each robot as little as possible from its current path in case it is approaching a bottle or the drop, is to control both robots using one two-task agent. This agent gets the sense data from both robots. The two task agent will be able to infer the same relative direction information from the `see_colour_blob` percepts that now identify the robot source, that the two agents must exchange using messages.

VI. Multi-tasking in `TeleoR` and a Future BDI Extension

One agent controlling the two robots bottle collecting is a multi-tasking agent. In this case the robot resource can be allocated to each task at the start and does not change. More generally, we might need tasks to alternate the use of a pool of resources. The programmer determines the granularity of the interleaving by program structuring and declarations that certain procedures are *task atomic*. The task atomic procedures are then compiled so that the tasks co-ordinate use of the resources themselves, using the *BeliefStore* as a Linda tuple store [4]. The low level co-ordination is opaque to the `TeleoR` programmer.

A major planned future extension is the incorporation of concepts from the BDI concept language AgentSpeak(L)[27] and its implementation in Jason [2]. We will extend `TeleoR` rules so that they can have `achieve Goal` actions as well as direct procedure calls. An extra non-deterministic top layer of *option* selection rules of the form

```
achieve Goal :: BSQuery ~~> ProcCall
```

is then used to find alternative calls for these goal actions, dependent upon current beliefs *when* the `Goal` needs to be achieved. As in Jason, these same selection rules can be used when the agent is asked to achieve a goal. Goal requests enable inter-agent task requests at the level of a common environment description ontology and do not require other agents or humans to know the names of another agent's task procedures and their argument types. We will also add similar rules for starting tasks in response to significant *BeliefStore* update events. Failure of a chosen option can now lead to selecting another option, using the option selection rules, adding another more course grained recovery mechanism to `TeleoR`.

We also intend to develop an inference supported IDE along the lines of LTMMop of [11]. This will aid the systematic develpment of `TeleoR` correct programs.

Other future extensions are to `QuLog`: the ability to handle uncertainty in sense data, and constraint handling.

References

[1] S. Benson, and N. Nilsson, Reacting, Planning and Learning in an Autonomous Agent, in *Machine Intelligence 14*, K. Furukawa, D. Michie, and S. Muggleton, (eds.), Oxford: the Clarendon Press, 1995

[2] R. H. Bordini, J. F. Hubner, M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*, Wiley-Interscience, 2007

[3] R. Brooks, *The Behaviour Language; User's Guide*, MIT AI Memo 1227, 1990.

[4] N. Carriero, D. Gelernter, Linda in context, *CACM*, 32(4), 1989.

[5] D. Choi, M. Kaufman, P. Langley, N. Nejati, D. Shapiro, An architecture for persistent reactive behaviour. In *Third International Joint Conference on Autonomous Agents and Multi-agent Systems-Volume 2* (pp. 988-995). IEEE Computer Society, 2004.

[6] K. L. Clark, Negation as Failure, *Logic and Data Bases*, eds. J. Minker and H. Gallaire, Plenum, 1978.

[7] K. L. Clark, P. J. Robinson, *Programming Robotic Agents: A Multi-tasking Teleo-Reactive Approach*, to be published by Springer, late 2015. Six chapters downloadable from: http://teleoreactiveprograms.net

[8] K. L. Clark, P. J. Robinson, *Engineering Agent Applications in QuLog*, to be published by Springer, late 2015.

[9] M. Destani, 2APL: A practical agent programming language, *Autonomous Agents and Multi-agent Systems*,16:214-248, Springer, 2008.

[10] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8), 453-457, 1975.

[11] C. Finocane et al, Designing Reactive Robot Controllers with LTL-MoP, *AAAI WS on Automated Action Planning for Autonomous Mobile Robots*, AAAI, 2011.

[12] A. Ferrein, G. Lakemeyer, Logic-based robot control in highly dynamic domains, *Robotics and Autonomous Systems*, 56:980-991, 2008.

[13] E. Hanson and J. Widom, An overview of production rules in data base systems, *The Knowledge Engineering Review*, vol 8, no 4, pages:121-143, 1993.

[14] K. V. Hindriks, Programming Rational Agents in GOAL, in *Multi-Agent Programming: Languages and Tools and Applications*, Springer, pages:119-157, 2009.

[15] J. Jones, D. Roth, *Robot programming: a practical guide to behavior-based robotics*, McGraw-Hill, 2004.

[16] E. Klavins, A Language for Modelling and Programming Cooperative Control Systems, *ICRA 2004*, IEEE Press.

[17] R. Kowalski and F. Sadri, Teleo-Reactive Abductive Logic Programs, *Festschrift for Marek Sergot*, (eds: Alexander Artikis et al, Springer, 2012.

[18] H. Levesque, M. Pagnucco, Legolog: Inexpensive Experiments in Cognitive Robotics, *Cognitive Robotics Workshop*, ECAI 2000.

[19] H. Levesque, *Thinking as Computation*, MIT Press, 2012.

[20] J. L. Morales, P. Sanchez, D. Alonso, A systematic literature review of the Teleo-Reactive paradigm, *Artificial Intelligence Review*, 1-20, 2012

[21] Nilsson, Nils J. Shakey The Robot, Technical Note 323. AI Center, SRI International, Apr 1984.

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

[23] P. J. Robinson, Home Page, http://itee.uq.edu.au/∼pjr/

[24] P. J. Robinson, K. L. Clark, Pedro: A Publish/Subscribe Server Using Prolog Technology, *Software Practice and Experience*, 40(4) pp 313-329, Wiley, 2010.

[25] Robin Murphy, Introduction to AI Robotics, MIT Press, 2000.

[26] Morgan Quigley et al, ROS: an open-source Robot Operating System, 2009. At: http://www.robotics.stanford.edu/∼ang/papers/icraoss09-ROS.pdf

[27] A. S. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language. In *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, LNAI, Springer, pp 42-55, 1996.

[28] R. Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, MIT Press, 2001.

[29] T. Swift, D. S. Warren, XSB: Extending Prolog with Tabled Logic Programming, *Theory and Practice of Logic Programming*, 12 (1-2), CUP, 2012.

[30] Michael Thielscher, *Reasoning Robots: The Art and Science of Programming Robotic Agents*, Springer-Verlag, 2005.

[31] M. Wooldridge, *An Introduction to Multi-Agent Systems*, 2nd edition, Wiley, 2009.

[32] J. S. Zelek, M. D. Levine, SPOTT: Mobile robot control architecture for unknown or partially unknown environments. *AAAI Spring Symp. on Planning with Incomplete Information for Robot Problems*, 1996