Robotic Agent Programming in TeleoR

Keith L. Clark¹ ² and Peter J. Robinson²

¹ Department of Computing, Imperial College London
www.doc.ic.ac.uk/~klc k.clark@imperial.ac.uk
² School of ITEE, University of Queensland, Brisbane
www.itee.uq.edu.au/~pjr pjr@itee.uq.edu.au

Abstract. We present an extension of Nilsson’s Teleo-Reactive rule based robotic agent programming language, the version presented in [24] which has the deductive BeliefStore. TR programs are sequences of rules guard ~> action clustered into procedures. The procedures encode goal (teleo) directed reactive task and sub-task behaviours of robotic agents. TR programs are robust and opportunity grabbing so well suited to human/robot or robot/robot co-operative tasks.

TeleoR has extensions motivated by the need to program communicating robotic agents with an extended range of task behaviours that can be cleanly programmed. Our goal was to do this without losing the elegance and simplicity of Nilsson’s language. We also wanted to be able to give TeleoR a formally defined operational semantics, building upon one we had given for TR (see Chapter 6 of [9]). The extensions, their semantics, and their implementation were developed in parallel. A methodology we can recommend.

In this paper we motivate and illustrate the TR extensions in TeleoR that facilitate the programming of more complex behaviours of single task communicating robotic agents. In other papers we will describe and illustrate the quite simple further extensions to support concurrent multi-tasking, with multiple robotic resources being shared by the tasks, and we will give the formal semantics.

1 Introduction

Nilsson’s Teleo-Reactive (TR) [24] 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 resource 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 procedures. The guards query, sometimes via fixed 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 resource actions to be executed in parallel; a call to a TR procedure, which can be a recursive call. In each called procedure there is a current fired rule. At the
bottom of the call hierarchy the fired rule always has robotic resource actions that are dispatched to the resources to effect changes in the agent’s environment.

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

When each new batch of percepts arrives, perhaps via a ROS [27] 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.

Using various compiler optimisations it is possible to make this reconsideration very fast, of the order of milliseconds, because the re-evaluation of guards in particular calls can often be safely skipped. Even when this is not the case, real robotic resources, often being mechanical, run very slowly compared with processor speeds. As a safeguard, the architecture ensures that all reconsiderations of guards is concluded before the percepts handler gets to update the BeliefStore, because the re-evaluation response to new percepts is an atomic call within the evaluator. This means each procedure call sees the same BeliefStore state for guard evaluation.

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. A procedure has no stop or exit action. Procedures do not terminate themselves. They get terminated when an ancestor call fires a different

![Fig. 1. Double Tower Architecture](image-url)
rule, or a different instance of the same rule. This operational semantics evolved out of Nilsson’s work on Shakey [22], particularly the triangular representation of Shakey’s plans, with influences from Ashby’s homeostasis concept [1] for an artificial brain, Miller et al.’s TOTE concept [20] of persistence until success, and Brook’s subsumption concept in his robot behaviour language [5]. In fact the TR language saw the light of day in a sabbatical visit by Nilsson to Brook’s lab at MIT.

Our extension, TeleoR, has evolved out of experience programming communicating agent applications in Qu-Prolog [8][14], and experience in the programming of robotic agent applications using an embedding of TR in Qu-Prolog using both simulators and a Lego Mindstorms robot.

In the rest of the paper, we start with a summary of TR’s simple syntax (slightly modified from that of [24]), the guidelines for how one writes a TR program, and an informal description of TR’s unique operation semantics. The 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.

We then illustrate the language giving most of the TR program for an agent controlling a mobile robot collecting bottles and delivering them to a drop area. Although a quite simple program it will enable us to illustrate all our single task extensions of TR in TeleoR.

Next we introduce our extensions, motivating each by a shortcoming in the TR bottle collector program. The extensions are:

- actions that are sequences of time limited durative actions, including procedure calls, for micro-behaviours where the component actions do not achieve a sub-goal that can be tested by a percept query,
- 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:
  - a while rule used to delay the re-achieving of the guard of the fired rule and to allow its action to continue while an alternative percept 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 rules action until some condition holds.

Temporary inhibition of some behaviour responses to new percepts when another has been selected is a concept in the behavioural language of Brooks [5], but our motivation for extending TR with this feature was not the behavioural language. It was an attempt by a colleague at UQ, Ian Hayes, to write a TR program for a safety critical systems test case. We amplify on this later.

TeleoR also has inter-agent communication actions. They enable us to program a bottle collection program that can be used by each of two agents controlling their own robot co-operatively collecting bottles in the same space, with
a joint goal to collect a given number of bottles. The agents communicate so that they both ‘know’ when the joint total is reached. They also communicate to compensate for poor visual processing capabilities to avoid collisions with minimal deviation from current paths.

A TeleoR program is typed. The BeliefStore percepts and relation defining rules are also typed. Using abstract interpretation [11] we can guarantee that all rule actions will be correctly typed and fully instantiated when a rule is fired. This is important when the actions are robotic resource actions that will be dispatched, perhaps via ROS [27], to real robotic resources. It is not clear to us that other logic based agent programming languages can give the same guarantee.

The TeleoR extension we present in this paper enables us to engineer, with significant compiler support to eliminate runtime errors and to give an optimised implementation, a goal directed robust robotic agent with significant extra fine grained control extensions over the original TR language.

We conclude by mentioning related work, the already implemented multi-tasking capabilities of TeleoR, and plans for further extensions, particularly the incorporation of concepts from the Beliefs, Desires, Intensions (BDI) languages AgentSpeak(L) [28], and its successor Jason [4].

We assume familiarity with logic programming [19], multi-agent systems [33], and robot behavioural programming [16],[26].

2 Nilsson’s TR programs

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

$$p(X_1, \ldots, X_k)\{$$

$$K_1 \sim> A_1,$$

$$\ldots,$$

$$K_n \sim> A_n\}$$

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

$$K_i \& \neg \exists K_{i-1} \& \ldots \neg \exists K_1$$

where the $\exists$ indicates existential quantification with respect to all the variables of the guard except the procedure parameters $X_1, \ldots, X_k$ that will be given variable free values when the procedure is called.

When a TR procedure is called, either as the initial procedure of some task or as a sub-task action, all the parameters $X_1, \ldots, X_k$ are given ground (variable free) values. This results in a sequence of partially instantiated rules $K_i \sim> A_i$, in which the parameter variables $X_1, \ldots, X_k$ are replaced by their values. Robotic actions can be durative - often continuing indefinitely unless explicitly changed.
Robotic Agent Programming in TeleoR

or stopped, such as moving forward at a certain speed, or 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.

**Example TeleoR 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 %.

```prolog
get_next_to(Th) {  % To be used when Th is bottle or drop
    at(Th) ~> ()  % Goal of the procedure 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)
        % As very near to Th, approach gradually slowing speed as get closer
    near(Th) ~> approach(Th,0.5,0.2)  % near(Th) achieved, approach
        % at constant slow speed to achieve close_to(Th,Dist)
    see(Th,_) ~> approach(Th,1.5,0.1)  % see(Th,_) achieved, 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 active 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 to Dir }
```

move(Speed), turn(Dir,Speed) are durative robotic actions, 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 using fixed facts about the colour of different things, 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. We ask the reader to study the comments in the two procedures for a further explanation.

**Goal regression** How can we be sure that the action of a rule is a suitable response to the environment state as determined by the rule’s guard? The answer is that an instance $A_i\theta$ of the partially instantiated action $A_i$ of the $i$th rule of a procedure call, corresponding to the first inferred instance $K_i\theta$ of the partially instantiated guard $K_i$ of the rule, *when no* instance of the guard of any prior rule is inferable, normally brings about a change in the environment such that an instance of a guard $K_j$, $j < i$ will eventually become inferable.

This change will be recorded by a future set of received percept facts, atomically added to the agent’s BeliefStore by the percepts handling thread, from which this achieved instance of the guard $K_j$ will become inferable. This may be after some time should $A_i$ be a procedure call, or contain a durative action. Nilsson calls $K_i$ the *regression* of this prior guard $K_j$ through $A_i$. The guard $K_1$ of the first rule is the goal of the procedure call. Its action, often () which is the do nothing action, should maintain that goal.

As an example, true is the regression of $\text{see}(\text{Th},\cdot)$ through the action $\text{turn(}\text{left},0.5\text{)}$. Assuming Th is within camera range of the mobile robot and there are no obstacles that might block its seeing Th, $\text{see}(\text{Th},\cdot)$ should become inferable from received percepts before the robot has turned through 360 degrees. $\text{at(Th)}$ is the goal of $\text{get_next_to(Th)}$.

A procedure is a *universal* procedure for its goal if every rule below the first rule has an action that normally will eventually bring about a sensed state in which the guard of an earlier rule is inferable (the regression property), *and* there is always a rule that can be fired, trivially satisfied if the last one has guard true. The *normally* caveat is because the agent might be continually thwarted by outside interference, or because actions of faulty robotic resources fail, or, because of environment conditions, are not as effective as normal (slippery floor).

The $\text{get_next_to}$ procedure has a special rule for $\text{Th=bottle}$. This is because $\text{at(bottle)}$ is inferable only if $\text{see(bottle,centre)}$ is inferable from the latest facts for the $\text{see_colour_b blob}$ percept, whereas $\text{at(drop)}$ 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.

On the reasonable assumption that the action move will normally lessen the distance between the robot and anything it can see, the first procedure is a universal procedure for its goal. No matter which of its rules is fired first, it should normally progress one at a time up the rule hierarchy. It will skip a rule
if being used to approach a bottle and someone moves the bottle closer, perhaps jumping from rule 5 to rule 3. 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 a bottle.

Each time the percept facts are updated the rule guards of all called procedures are checked to see if their last fired rule should remain the call’s fired rule. The retesting starts with the first call - the top level task call. Starting with the first rule of the call, and moving down the sequence of rules, the partially instantiated guards of each rule are re-evaluated against the new percepts just as they were when the task started.

The operational semantics of TR programs is fully explained in [9] where Chapter 6 gives an algorithm for responding to BeliefStore updates. [9] 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. A negative dependency means that removal of a fact for that percept predicate may make the guard inferable. The optimisations allow the avoidance of the re-evaluating of rule guards of certain procedure calls, or the safe skipping of the checking of some initial sub-sequence of guards. We have the percept handler record meta-information in the BeliefStore about which percept relations it has just been updated, and in which way (by adding or deleting a fact or both). The ability to skip re-evaluation of guards is very beneficial when we have task procedures with guards that do quite complex inference but only access percepts or other dynamic facts that change relatively slowly.

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 [9]. The percent handler’s inference and remembering of all instances of these rule defined relations is a special form of tabling [31]. 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.

**Need for type correct and ground action guarantees** Consider the last rule of the approach procedure. Its set of actions are the robotic resource actions move(Fs), turn(Dir,Ts). These, as control messages, will be dispatched to a real robot to control drive motors. We need to have a guarantee that Fs and Ts will have numeric values and that Dir will be bound to a symbol that is a direction that will be understood by the robot interface process. This is why we use the typed and moded companion QuLog[10] language for our agent’s BeliefStore. A mode is an annotation on the argument type of a relation that indicates if the argument must be given, or must be an unbound variable in a call, or can be either (the default if un-annotated).

**BeliefStore updates and inter-agent communication** Nilsson allows rule
actions to be *BeliefStore* updates. Our view is that one should do the update concurrently with robotic resource actions.

Suppose that the agent was collaborating with another agent to collect bottles and that they have a joint goal to collect *Total* number of bottles. As well as updating the count of delivered bottles as a *BeliefStore* update, that another bottle has been delivered should be communicated to the other agent, and vice versa, so that they can both know when their joint goal has been achieved.

How is such an incoming message handled by the agent? We simply add another thread in the agent architecture, a message handling thread, which is the public interface of the agent. Such a thread can also handle queries and task requests. Like the percepts handler it is in written in QuLog action rules.

### 3 TeleoR Agent Architecture and Example Program

A single task TeleoR robotic agent architecture has three threads as depicted in Figure 2. Pedro [25] is a publish/subscribe and addressed message 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 address style names to other agents registered with the same Pedro server. The agent can lodged subscriptions with the server that are of the `Ptn::Test` where `Test` is a Prolog query using only Pedro supported primitive relations. It 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` unifies with `Ptn` and the linked `Test` is inferable by Pedro.

![Fig. 2. Three Thread Communicating TeleoR Agent Architecture](image-url)
Collaborating bottle collector program in TeleoR

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 used by each of two agents controlling their own robot 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 a delivery 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

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

belief colour:(thing,col)
  % Type declaration for another updatable belief

communicating_collect_bottles:(int,pedro_handle)
  % Type decl. for a TeleoR procedure with integer and pedro handle as args

communicating_collect_bottles(Total,OthrAg)
{
  $collected$=Total $\rightarrow$ () $\%$ $collected$ is current value in collected_belief

  delivered while time 8 $\rightarrow$
    turn(right,0.5) $\| 3$; forward_avoiding(1.5,centre,0) ++
    collected += 1; delivery to OthrAg
    % For 8 secs inhibit rules below even though delivered will not be inferable
    % soon after the turn starts. Uses timed sequence action, turn for 3 secs, then call
    % forward_avoiding. At start, do a BeliefStore update and a message send action
    holding $\rightarrow$ deliver_bottle

  true $\rightarrow$ get_bottle
}
```
at: (thing)  % Type/mode declaration 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 this robot, send message telling other agent
}

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)
% Should move slowly in slight arc to right when each sees other robot on the left
...
% Other facts giving other avoid moves

approach:(thing, num, num)
approach(Th, Fs, Ts){
    see(Th, centre) -> forward_avoiding(Fs, centre, 0)
    see(Th, Dir) until see(Th, dead_center) -> forward_avoiding(Fs, Dir, Ts)
    % When 2nd rule fired it inhibits rule above until Th is seen dead_center,
    % not just centre
}

get_bottle{
    holding ~> ()
    next_to(bottle, centre) & gripper_open ~> close_gripper wait 3 repeat 2
    % Repeat action twice at 3 sec. intervals if no observable effect, in case stuck open
    next_to(bottle, centre) ~> open_gripper wait 3 repeat 2
    % Gripper cannot open if testing this guard
    true ~> get_next_to(bottle)
}
We ask the reader to study the comments in the program to get an understanding of its key features and differences from a TR program. Since the relation argument types are un-notated they are all dual input/output mode. A ! annotation indicates ground input only, a ` that the call argument will be a variable that will be bound to a ground value by the call. Relations and functions over lists and trees etc. can be defined in QuLog. Functions always have ground arguments and return a ground value.

\texttt{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 \texttt{tell} or \texttt{inform} message containing the fact \texttt{colour(bottle,brown)}. Providing we have programmed the message handler to accept such messages, the agent controlled robot will immediately after message receipt start collecting \texttt{brown} bottles as well as \texttt{green} ones.

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

\texttt{while} and \texttt{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 \cite{Brooks1986}. However, as mentioned in the introduction, inhibiting rules were not added to \texttt{TeleoR} because inhibition is in the Behavioural Language. A colleague Ian Hayes felt the need of a form of \texttt{until} rule to allow the semantically clean programming of a safety critical systems test case. We weakened his \texttt{until} and added the dual \texttt{while} rule. There is a combined \texttt{while/until} rule that approximates the semantics of Haye’s \texttt{until}. Deciding on the pros and cons of different rule form extensions was considerably helped by first formulating their state transition semantics. A \texttt{while/until} with a \texttt{time} \texttt{T} condition makes any durative actions of the rule ballistic for \texttt{T} seconds, providing the procedure call in which it is fired remains active.

A \texttt{while} rule, which has the form

\begin{align*}
\text{K while } C \sim & \ A \\
\text{temporarily prevents the re-achieving of its guard while the condition } C \text{ remains inferable, even though the guard may no longer be inferable. An } \texttt{until} \text{ rule, which has the form}
\end{align*}

\begin{align*}
\text{K until } U \sim & \ A \\
\text{allows over-achieving of the guard of a rule above, providing its guard remains inferable, until the condition } U \text{ 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 \texttt{Th} is no longer in the range -10 to +10 degrees of the centre of the camera image, the swerve correction is continued until}
\end{align*}
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 **delivery** 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 be 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 a special form of resource failure belief to the agent’s **BeliefStore**. This can be ‘caught’ by an outer procedure that has a rule for responding to such beliefs when they appear. This rule can send a message to a person for help. If and when it is fixed - the gripper is oiled, say - the person sends a message, say **continue**, to the agent telling it to resume. Its message handler could 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 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** message which also test the other agent **A2** that **R1** is temporarily stationary. 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, queried in the guard of its first rule. It will not be received 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 and **forward_avoiding** is no longer an active call.
Each agent’s message handler handles *stopped* as well as *delivery* messages. It responds to the receipt of a *stopped(Dir)* message by doing the update actions

```plaintext
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.

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 moving 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.

## 4 Related Work

A comprehensive survey of extensions and applications of the teleo-reactive paradigm is given in [21].

In [3] 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-goals 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 [34]. 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.

[7] is an action skill representation system with the skills invoked in response to perceptions. Skills can invoke sub-skills which can be executed sequentially and conditionally. The skills are goal directed and reactive. Its extension [8] has concurrent execution of skills.

[29], [17], [13], [15] and [32] present logic based approaches to programming software agents that either have been (e.g. [18] and [13]), or could be used for robotic agents with varying degrees of efficiency.

None of the other logic based approaches offer compile time guarantees of type and mode safe inference, of type correct and ground actions, and optimised implementation avoiding unnecessary inference. However others [30], [2] see the need for type safe agent programming languages.

5 Multi-tasking in TeleoR and a Future BDI Extension

We have already introduced the idea of a multi-tasking agent controlling multiple resources - the one agent controlling the two robots bottle collecting. In this case the resources could be allocated to each task at the start and do not change. More generally we might need tasks to alternate the use of a pool of resources. (For an example of this see the video accessible from the first author’s web page of one agent alternately using two robotic arms and three tables as resources, in four concurrent tasks.) TeleoR co-ordinates the use of resources by multiple concurrent tasks using the concept of a task atomic procedure. The programmer determines the granularity of the interleaving by the program structuring and choice of the task atomic procedures. 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 [6]. The low level co-ordination is opaque to the TeleoR programmer.

Our main planned future extension is the incorporation of concepts from the BDI concept language AgentSpeak(L)[28] and its implementation in Jason [4]. 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

\[
\text{achieve Goal :: BSQuery } \sim \rightarrow \text{ProcCall}
\]

is then used to find alternative calls for these goal actions, dependent upon current beliefs when the Goal need to be achieved. As in Jason, these same selection rules can be used when the agent is asked to achieve a goal. They enable inter-agent task requests at the level of a common environment ontology and do not require other agents or humans to know the names of the 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.
Robotic Agent Programming in \textit{TeleoR}

\textit{TeleoR} and \textit{QuLog} are currently compiled into Qu-Prolog, we intend to re-implement them by compiling to Java or Scala.

\section*{References}
12. Brad Darrach, Meet Shaky, the first electronic person, \textit{Life}, 20th November, 1970