Concurrent Task Programming of Robotic Agents in TeleoR

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

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

Abstract. `TeleoR` is a major extension of Nilsson's Teleo-Reactive (`TR`) rule based robotic agent programming language. Programs comprise sequences of guarded action rules grouped into parameterised procedures. The guards are deductive queries to a set of rapidly changing percept and other dynamic facts in the agent's *Belief Store*. The actions are either tuples of primitive actions for external robotic resources, to be executed in parallel, or a single call to a `TeleoR` procedure, which can be a recursive call. The guards form a sub-goal tree routed at the guard of the first rule. When partially instantiated by the arguments of some call, this guard is the goal of the call.

`TeleoR` extends `TR` in being typed and higher order, with extra forms of rules that allow finer control over sub-goal achieving task behaviour. Its *Belief Store* inference language is a higher order logic+function rule language, `QuLog`. `QuLog` also has action rules for programming behaviour threads within an agent. The action of a `TeleoR` rule may be a combination of the action of a `TR` rule and a sequence of `QuLog` actions.

`TeleoR`'s most important extension of `TR` is the concept of *task atomic* procedures, arguments of which may belong to a special but program specific `resource` type. This allows the high level programming of multi-tasking agents using multiple robotic resources. When two or more tasks need to use overlapping resources their use is alternated between task atomic calls in each task, in such a way that there is no interference, deadlock or task starvation.

The co-ordination of the use of the resources is done by code generated by the `TeleoR` compiler for the task atomic procedures. It is a concurrent algorithm in which the concurrent tasks atomically query and update special co-ordination facts in the agent's *Belief Store*. The `TeleoR` programmer does not need to know about this co-ordination.

This multi-task programming is illustrated by giving the essentials of a program for an agent controlling two robotic arms in multiple block tower assembly tasks. It has been used to control both a Python interactive graphical simulation and a Baxter robot building real block towers, in each case with help or hindrance from a human. The arms move in parallel whenever it can be done without risk of clashing.

1 Introduction

TeleoR is a major extension of Nilsson's Teleo-Reactive (TR) language [22]. Both are mid-level robotic agent programming languages. They assume basic level routines written in procedural programming languages such as C. These routines will do sensor interpretation, particularly for camera images, and may implement quite complex robotic resource control actions such as moving a jointed arm to a given co-ordinate location, or to be next to a detectable object.

The results of the sensor interpretation routines become available to the robotic agent as rapidly changing percept facts held in its *Belief Store*. An example is `on_table(1)`, reporting that a block labelled with the number `1` is 'seen' directly on top of the table. The action routines are invokable by the agent by starting an action such as `put_on_block(1)`, when the agent believes no block is directly on top of block `1`. This *normally, eventually* puts the currently held block, say block `8`, on top of block `1`.

Why the caveat *normally, eventually*? It is because we assume the environment may be affected not only be robotic actions but by exogenous events. Before the action of putting the held block on top of block `1` completes, another block may be placed on top of block `1` by an interfering person, meaning that a required condition for the `put_on_block(1)` arm action no longer holds. Block `1` may also be temporarily moved to be out of reach of the arm. Only when there are no interfering exogenous events, or the events do not for ever thwart the robotic action, will the action eventually succeed.

TR and TeleoR are languages for deciding to invoke the C implemented arm action because doing so will achieve some logically expressed sub-goal of a current task goal, assuming the current percept beliefs accurately describe the world. This sub-goal might be to have block `8` be directly on top of block `1`, on route to a task goal of building the block tower `[5,3,8,1]`.

A simple two thread architecture for a TeleoR agent is depicted in Figure 1. Notice that the percept thread's update of the *Belief Store*, and the determining of a new action response, are both atomic. Having determined an action response, TeleoR evaluator thread will suspend until the *Belief Store* is updated. It is reactivated immediately the *Belief Store* is updated to determine a new tuple `As` of robotic actions. Any currently executing action not in `As` is terminated, any new action is started, and other actions are allowed to continue perhaps with modified parameter values - `move(4)` may become `move(3)`.

In the next section we describe standard TeleoR syntax, which corresponds to TR syntax, and we informally give the unusual operation semantics of TR and standard TeleoR, which is formally defined in Chapter 5 of [7]. In the following two sections we describe the features of TeleoR not in TR, first for programming single task agents, and then for programming multi-tasking agents sharing and interleaving the use of a set of robotic resources to achieve their different but compatible goals. We conclude by mentioning related work, and our plans for further extensions of TeleoR and our agent architecture.

The paper assumes familiarity with logic programming [19] and robot behavioural programming [16],[20].

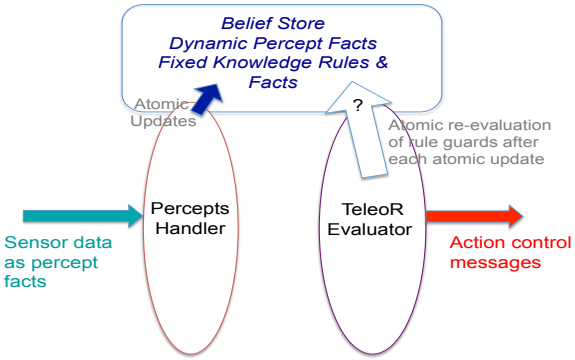

Fig. 1. Simple Two Thread TeleoR Agent Architecture

2 Standard `TeleoR` procedure syntax and informal operational semantics

A standard syntax `TeleoR` procedure, corresponding to a `TR` procedure, comprises a parameterised *sequence* of *committed choice guarded action* rules of the form:

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

Here the $\texttt{G}_i$ are the guards, the $\texttt{A}_i$ the actions, and the parameters $\texttt{X}_1\texttt{,..,X}_k$ can appear in any guard or action. When a procedure is called the parameter values partially instantiate the guarded rules which are tried in the order given.

A rule guard is a `QuLog` query to the agent's *Belief Store*. `QuLog` is a flexibly typed higher order logic+function+action rule programming language, see Chapter 3 of [7]. Its dynamic facts constitute the agent's changing beliefs. Its rules and fixed facts comprise the agent's *knowledge*, allowing higher level and context dependent interpretation of the *Belief Store* dynamic facts.

A rule action is a tuple of robotic resource actions executed in parallel, e.g. `move(4.5),turn(left,0.5)`, or a single call to a `TeleoR` procedure, which may be a recursive call.

Example procedure calling an auxiliary procedure Here are two procedures for trying to get a mobile robot close to something `Th` making use of

independent `move` and `turn` actions, and a general `see` percept. We use the Prolog convention that variables begin with an upper case letter or underscore.

```
get_close_to(Th){
    see(Th,close,_) ~> ()
    see(Th,near,_) ~> approach_until(close,Th,3.0,1.0)
    see(Th,far,_) ~> approach_until(near,Th,4.5,0.5)
    true ~> turn(right,0.5)
    }
approach_until(Dist,Th,Fs,Ts){
    see(Th,Dist,_) ~> () % Th being approached is now Dist away
    see(Th,_,centre) ~> move(Fs)
    see(Th,_,Dir) ~> move(Fs),turn(Dir,Ts)
    % Dir is left or right. move forward turning Dir to bring back into centre view.
    }
```

The underscores in the `see` conditions of the first procedure, and the first rule of the second procedure, indicate that the orientation of the seen `Th` is not relevant for the action of the rule. Those in the last two rules of the second procedure indicate that the distance is not relevant.

`move` has one argument, a numerical forward speed. `turn` has two arguments, a direction of turn `left`, `right` or `centre` and a turn speed. The second call to `approach_until` has a higher forward speed and a lower correctional turn speed as `Th` is further away. So, there is more time to bring it back into centre view.

`see` is a three argument percept that identifies the thing seen from a small set of alternative objects that a vision routine can recognise, it gives a qualitative measure of its distance from the robot's on-board camera, as `close`, `near` or `far`, and indicates whether the seen thing is within, or to the left or right of a central area of the camera's field of view.

Guards as goals and goal regression The guards of a `TeleoR` procedure call should lie on a sub-goal tree routed at the guard of the first rule. When partially instantiated by the values of parameters $\mathtt{X}_1,..,\mathtt{X}_k$ of some call, this guard instance $\mathtt{G}_1'$ is the *goal of the call*.

The goal of `get_close_to(bottle)` is $\exists Dir$ `see(bottle,close,Dir)`. The goal of `approach_until(near,bottle,4.5,0.5)` is $\exists Dir$ `see(bottle,near,Dir)`.

If an action $\mathtt{A}_j$ is started when its guard is the first inferable guard, and continued whilst this is the case, it should be such that it will *normally, eventually* result in progression up the sub-goal tree of guards. That is, eventually the guard of an earlier rule $\mathtt{G}_i$, $i < j$, should become the first inferable guard. Nilsson calls $\mathtt{G}_j$ the *regression* of $\mathtt{G}_i$ through $\mathtt{A}_j$.

As an example, let us consider the second procedure. Suppose for the call `approach_until(near,bottle,4.5,0.5)` the last rule is the first rule with an inferable guard. It must be the case that `see(bottle,far,Dir)`, where `Dir` is `left` or `right`, is the latest percept. If the distance argument was `near` the first rule

of the procedure call would have been fired. If it was `close`, the first rule of `get_close_to(bootle)` would have fired and the `approach_until` call would no longer be active. If `Dir=left`, the action is `move(4.5),turn(Dir,0.5)`. This is move forward accompanied by a parallel turn. Providing the `see` percept in the *Belief Store* continues to be `see(bottle,far,left)`, this parallel pair of actions will continue. It should normally eventually result in either the first rule firing, because the latest received percept is `see(bottle,near,Dir)`, for some `Dir`, or in the second rule firing, because `see(bottle,far,centre)` is the latest received percept.

The reader might like to satisfy themselves that this property holds for all the rules of the two procedures. A similar program is discussed in more detail in Chapter 2 of [7].

Covering all eventualities The partially instantiated guards of a procedure call should also be such that for every *Belief Store* state in which the call may be active there will be at least one inferable guard. Nilsson calls this the *completeness* property of a procedure. This property holds for both our example procedures. For the first it trivially holds since the last rule will always be fired if no earlier rule can be fired. It holds for the second procedure given the two guard contexts from which it is called in the first procedure, both of which require a `see` percept to be in the *Belief Store* while the call is active.

Universal conditional plans Nilsson calls a complete `TR` procedure satisfying the regression property a *universal* procedure for achieving its goal for any call. It may also be viewed as a *universal conditional plan* for achieving its call goals. These concepts also apply to `TeleoR` procedures.

Informal operational semantics for a standard `TeleoR` task A task is launched by calling some procedure such that the task goal is implied by the call's goal. The *first* rule of the call with a guard instance inferable from the current *Belief Store* is *fired*, resulting in a fully determined action. If this is a procedure call, its first rule with an inferable guard is fired, and so on until a rule with robotic actions is fired. Its actions are started.

When each new batch of percepts arrives, perhaps via a ROS [24] interface, this process of finding and firing the first rule of each call with an inferable guard is restarted. This is in order to determine as quickly as possible the appropriate tuple of robotic actions response to the new percepts. Actions that were in the last tuple of actions are allowed to continue, perhaps modified. Other actions of the last tuple are stopped. New actions are started. For example, if the last tuple of actions was `move(4.5), turn(left,0.5)` and the new tuple is just `move(3)`, the `turn` action is stopped and the `move` action argument is modified to `3`.

Elasticity of complete procedure programs This reactive operational semantics means that each `TeleoR` procedure is not only a universal conditional

plan for its call goals, it is also a plan that recovers from setbacks and immediately responds to opportunities. If, after a *Belief Store* update a higher rule of some call *PCall* can unexpectedly be fired, perhaps because of a helping exogenous event, that rule will be fired *jumping* upwards in the task's sub-goal tree. If instead a lower rule of *PCall* must be fired, a detected unexpected result of some robotic action, or a detected result of a interfering exogenous event, this means that the climb up the sub-goal tree of *PCall*'s rule guards must be reattempted from a different sub-goal of its call goal. There has been a setback in the progress towards the task goal but the recovery response action should *normally and eventually* result in its being achieved.

For our example procedures, for a task call `get_close_to(bottle)`, suppose that initially there is no `see` percept in the agent's *Belief Store*. The last rule of the call will be fired. Assuming there is at least one bottle in the environment of the robot within range of its camera, before the robot has completed a 360 degree turn one of the first three rules should fire. If it is the first rule, the task goal has been achieved. Its `()` empty action will cause the turn action to be stopped.

Now suppose that the bottle is moved away from the robot and the percept `see(bottle,far,left)` is received. Immediately the program tries to recover from this outside interference by firing the call's second rule. The third rule of the auxiliary call `approach_until(near,bottle,4.5,0.5)` will then be fired causing the robot to move forward slowly, swerving slightly to the left. As it is doing this, and before either the `bottle` is seen in centre view or as `near`, suppose the bottle is moved back to be `close` to the robot and in view. The robot has been helped. The first rule of `get_close_to(bottle)` will again be fired, with the result that both the `move` and the `turn` actions will be terminated.

In fact, if it is only ever called from `get_close_to(bottle)`, the first rule of `approach_until` auxiliary call will never be fired as its firing will always be pre-empted by the firing of a different rule of the parent call. This is typical for an auxiliary procedure. Unless it is the initial procedure call of some task, its goal achieved first rule usually does not get fired. This is because the procedure has been called to achieve the guard of a higher rule of its parent procedure call and that higher rule will be fired as soon as the goal of the auxiliary procedure call has been achieved, pre-empting the firing of its goal achieved rule.

There is a scenario in which the task goal will never be achieved. This will happen if whenever the robot is about to get close to a bottle the bottle is either moved out of sight or further away. The robot will doggedly chase the bottle until its battery runs out.

From deliberation to reaction Although not the case for our example procedures, typically, initially called `TeleoR` procedures query the percept facts through several levels of defined relations. Via procedure call actions, they eventually call a `TeleoR` procedure that directly queries the percept facts and mostly has non-procedure call actions. So, for `TeleoR` and `TR` the interface between deliberation about what sub-plans to invoke to achieve a task goal, to the invoking

of a sensor reactive behaviour to directly control robotic resources, is a sequence of procedure calls.

3 Extra features of TeleoR for programming single task agents

The TeleoR extension of TR was created in two stages. The first stage was to make the language more suited to programming single task agents controlling real robotic resources, perhaps via a ROS interface. A primary concern was to have a compile time guarantee that actions sent out to robotic resources would be fully determined and correctly typed. To this end TeleoR was made a typed language, and the untyped Prolog like *Belief Store* inference language of TR, as used in [22], was replaced by the typed higher order language QuLog. This has a declarative core of logic + function rules and a top layer of imperative rules for programming agent threads. Imperative rules can use query relations and call functions but can also execute primitive actions for forking new threads, for updating an agent's *Belief Store* dynamic facts, and for inter-agent message communication. The communication uses our independently developed external communications server Pedro [27].

Type definitions and declarations A TeleoR procedure named p must be given a type declaration of the form:

`tel p(`t_1`,..,`t_k`)` *% declaration of the argument types of* p

where each t_i is a QuLog type expression.

In addition, the predicate names and argument types of the percept facts must be declared, as well as the names and argument types of the robotic actions. The actions are also classified as discrete or durative. A discrete action executes for a short time and cannot be prematurely terminated, for example a bleep sound. A durative action can be stopped and modified before it naturally terminates, and may not even naturally terminate. An example is move(S) which makes a mobile robot move forwards more or less in a straight line at a speed S. S can be changed causing the robot to speed up or slow down, and the action continues unless explicitly stopped. For our two example procedures we need to have:

```
def thing ::= bottle | basket | ..
% Enumerative type def. of the recognisable things
def dir ::= left | centre | right
def distance ::= close | near | far
percept see(thing,distance,dir)      % Just one percept relation
durative move(num), turn(dir,num)    % Two independent durative actions
tel get_close_to(thing)     % Type declarations for the two TeleoR procs.
tel approach_until(distance,thing,num,num)
```

The relations that query the percept facts, defined by rules as well as facts, must have their argument types and modes of use declared. The modes of use specify which arguments must be given as fully determined (*ground*) values, and which can be underspecified, given as an unbound variable or a non-variable term containing variables (*a template* term) when the relation definition is 'called'. Arguments that do not need to be ground in the relation call, but which will be given a ground value if the call succeeds, are annotated with a preceding `?`.

The `TeleoR` compiler uses the declared types of the parameters of a procedure, which must always be given ground values when it is called, the argument types for the percept relations, and the moded argument types for the program defined and built-in relations, to check that:

- each rule guard only has correctly typed queries for percepts, rule defined and primitive relations,
- all arguments moded as needing to be ground will have ground values when a relation is queried by the left to right evaluation of guard conditions,
- all variables in the rule's action will have correctly typed ground values if the guard succeeds.

Communication and *Belief Store* update actions We mentioned earlier that a `QuLog` action call sequence can be optionally added to the robotic action of a `TeleoR` guarded action rule. The most useful `QuLog` primitive actions to use are message send actions, and *Belief Store* update actions within an agent.

Messages are communicated between agents using our Pedro [27] communications server. This supports both peer-to-peer communication, in which the recipient agent is identified by an email address style `agent_handle` of the form `agent_name@host_name`, and publish/subscribe communication. For the latter, the destination of the recipient is given as `pedro` and the message is forwarded to all agents that have a current subscription lodged with the Pedro server which *covers* the notified message.

As a simple example of the use of a notification and a *Belief Store* update we could change the first rule of our first example procedure to be

```
see(Th,close,_) ~> () ++ update_count(Th,OldC); count(Th,OldC+1) to pedro
```

using the `QuLog` dynamic relation declaration and update action rule

```
dyn count_for(thing,nat)
count_for(bottle,0)
count_for(basket,0)
count_for(....)
....
act update_count(thing,?nat)
update_count(Th,C) :: count_for(Th,C) ~>
      forget count_for(Th,C) remember count_for(Th,C+1)
% Atomic update of count_for fact for Th
```

Each time the robot gets close to a thing it updates a count fact in the *Belief Store* of how many times this has happened. It also sends out a notification of the new count value which will be forwarded to every agent that has lodged a covering subscription with the Pedro server of the of form `count(_,N) :: integer(N).`

With communication and *Belief Store* update actions a single task `TeleoR` agent now has the three thread architecture of Figure 2. All incoming messages

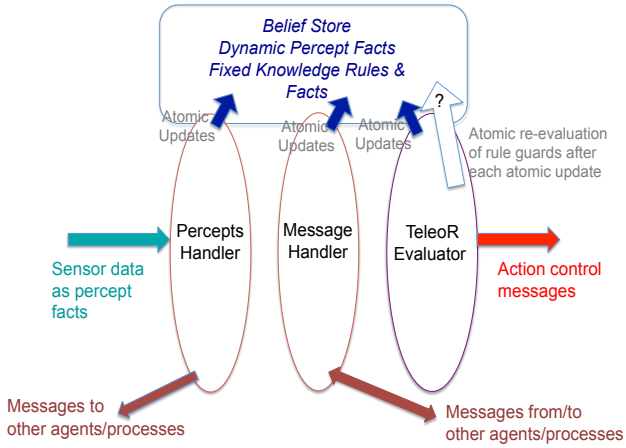

Fig. 2. Single Task Three Thread TeleoR Agent Architecture

go to the message handling thread which must also lodge and maintain the agent's Pedro subscriptions. How this is done, and how the agent's message handling thread handles received messages is outside the scope of this paper. It is explained in Chapter 3 of [7].

The `TeleoR` single task extensions of `TR` are more fully described in [6]. The paper has an example of the use of communication between two mobile robots co-operatively collecting bottles and delivering to a drop area. Communication is used so that each robot knows how many bottles they have jointly collected, stopping when a certain total is reached. It is also used to compensate for poor vision. Another robot can be seen and its distance determined but its direction of travel cannot be perceived. Communication allows the robots to avoid collisions with minimal divergence from their current path.

The main focus of this paper is the `TeleoR` programming of a multi-tasking agent where each task is a `TeleoR` evaluation thread within the agent. Each thread can therefore access the same set of percepts and any facts that are remembered by a task thread. This is because percepts and remembered facts are all stored in the shared *Belief Store*. Any communication is via *Belief Store* updates.

We shall just need to make use of two new forms of rule that have a different semantics from the standard `TR` style rules with respect to how long its action continues after a rule has been fired.

`until` rules: *Guard* `until` *UCond* `~>`*Action*
When the rule is fired with firing instance *Guard*′ of its guard, the corresponding fully instantiated *Action*′ will continue whilst remains inferable from the changing *Belief Store*, *even if* a higher rule of the procedure call could be fired, *providing* the corresponding instance *UCond*′ of the `until` condition also remains inferable. As soon as *Guard*′ is no longer inferable, or *UCond*′ is not inferable, *Action*′ will be replaced by the action of another rule firing of the procedure call providing this is still active. (The new rule firing could be a refiring of the same rule, with a different inferable instance of the *Guard*). This form a rule is often used to allow *Action*′ to *over-achieve* the guard of an earlier rule of the procedure call.

`while` rules: *Guard* `while` *WCond* `~>`*Action*
After the rule as been fired with inferred guard instance *Guard*′, the corresponding instance *WCond*′ becomes an alternative to *Guard*′. Providing no earlier rule becomes fireable after a *Belief Store* update, the corresponding action *Action*′ will be continued if *Guard*′ or *WCond*′ remains inferable. *WCond* is not an alternative firing condition. The rule is *not* equivalent to *Guard* `or` *WCond* `~>`*Action*.

`while...until` rules: *Guard* `while` *WCond* `until` *UCond* `~>`*Action*
This rule allows the *Action*′ action instance to continue even if a higher rule can be fired, providing either *Guard*′ or *WCond*′ remains inferable. We shall not need this form of rule for our example program.

4 Multi-tasking and task atomic procedures

The second phase of extension of Nilsson's `TR`, and arguably the more important, were changes to the language and the way a source program is analysed and compiled. This was to allow the high level programming of multi-tasking agents dynamically sharing the use of multiple robotic resources. This second extension is the primary subject of this paper.

Special robotic resource type The resources that must be shared are identified by declaring a special `resource` type. The granularity of the interleaved sharing of the resources is then specified by the declaration that certain procedures that have `resource` arguments are `task_atomic`. A task atomic procedure call is a critical region for a task. The procedure call may be entered only if all its `resource` arguments are free. Whilst the task is firing rules of that task atomic call no other task can use any of its resources. The resources are released immediately the task atomic call is no longer active, because of a different rule firing in an ancestor call. This frees them for use by a waiting task, or for re-use by the same task if no other task is waiting to use any of the freed resources.

Avoiding deadlock To avoid deadlock, only the resource arguments of the first task atomic call *TaCall* entered by a task `T1` may be used throughout the evaluation of *TaCall*. This is guaranteed by a compiler check that only the resource arguments of each task atomic procedure *TaP* are passed as arguments to auxiliary procedure calls that may be made directly or indirectly from *TaP*. So after having acquired the resource needs of its initial task atomic procedure call, task `T1` will not need to wait for extra resources in order to enter an inner task atomic call. The compiler also makes sure that only the resource arguments of *TaCall* are used as action parameters of any rule that might be fired by *TaCall* and any auxiliary procedure calls it might make.

Deadlock could occur if an extra resource was needed by `T1` which was being used by a concurrent task `T2`, and `T2` also required an extra resource being used by `T1`. Breaking the deadlock by releasing resources when extra resources are required would mean that the `task_atomic` calls were not *task atomic*.

4.1 Architecture of a multi-tasking agent using multiple resources

An agent that can concurrently execute several tasks using *multiple* resources has an architecture as depicted in Figure 3. All the tasks threads are active and on each percepts update they re-compute their sequence of fired rules.

The co-ordination of the use of resources is done by the tasks themselves using code generated by the `TeleoR` compiler for each task atomic procedure. This atomically queries and updates special co-ordination facts in the agent's *Belief Store*. These record which tasks are currently `running_`, i.e. inside a task atomic call, and which tasks are currently `waiting_` for resources, and when they started waiting. Because the execution of threads is time shared the waiting start times are different and define a wait queue order. Separate `resources_` facts record the resource use and resource needs of each task.

The running tasks respond to a *Belief Store* update in any order. If a running task exits its initial task atomic call it typically suspends at the firing of a rule that has a new task atomic call as its action, which could be a new call to the procedure of the call it just exited. The compiled code for the task then updates the task's `resources_` fact to the resource needs of the new call, forgets the task's `running_` fact, and remembers a `waiting_` fact for the task recording the current time. This puts it at the end of the current queue of waiting tasks.

The waiting tasks respond to a *Belief Store* update in wait queue order. The response to a *Belief Store* update by a waiting task may also result in the need to enter a different task atomic call, with a corresponding update of its `resources_` fact, but not its `waiting_` fact.

Transition from waiting to running preventing starvation After each *Belief Store* update, after it has determined any change of resource needs, a waiting task will immediately transition to a running task if none of its resource needs are being used by a running task or is needed by a waiting task ahead of it on the wait queue. (As all these other tasks have already responded to the

latest *Belief Store* update their recorded resource needs are up to date.) If this check succeeds, the waiting task immediately transitions to a running task by forgetting its `waiting_` fact and remembering a `running_` fact. Its `resources_` fact is unchanged. Not allowing a waiting task to become a running tasks if a task ahead of it on the wait queue has overlapping resource needs prevents *starvation*.

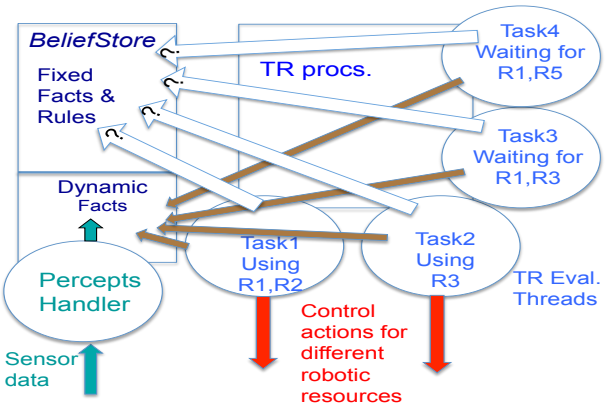

Fig. 3. Multi-Task `TeleoR` Agent Architecture without Communication

Multi-resource multi-tasking example We exemplify `TeleoR` programming of a multi-tasking multi-resource using agent with a program for an agent sharing two robot arm resources between multiple concurrent configuration tasks. We use the classic block tower configuration task. Two arms are needed as blocks are distributed over three tables as in Figure 4, and each arm can only reach two tables: a home table and a shared table between the two home tables.

A task to build a tower using blocks labelled `[2,6,3,1]` on `table2`, with block 1 directly on the table, can mostly just use `arm2`. However, when block 6 needs to be fetched to be put on top of block 3, the task must first use `arm1` to move block 6 to the shared table, so `arm2` can reach it to put it on top of 3 on `table2`. So the arms must be dynamically acquired by tower building tasks.

Let us suppose the agent has a concurrent task to build tower `[4,7,9,10]` on `table1`. (All the concurrent tasks must use different blocks.) That second task can mostly use `arm1`. However, when 4 needs to be put on top of 7, `arm2` must be used to first transfer this block to `shared`. Since the tasks are running concurrently we cannot allow either task to just start using the other arm when it has the need.

Stable sub-goals to prevent interference of compatible tasks We must have a way for a task to occasionally release resources, but only if the task has achieved a *stable* sub-goal of its task goal unless the attempt to achieve

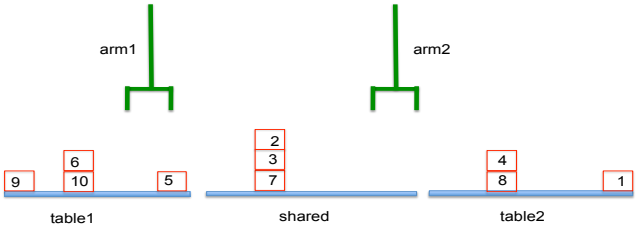

Fig. 4. Two Arm Multi Tower Building

that sub-goal has been aborted. The stable sub-goal concept was introduced by Benson and Nilsson in [2] for use by a `TR` multi-tasking scheduler that alternated evaluation of several tasks with no concurrent execution.

A stable sub-goal is one that will not be undone by another task when it acquires the released resources. For tasks building towers of different blocks, an un-stable sub-goal would be `holding(arm1,3)`. If the `arm1` resource is then acquired by another task, block `3` will be put down somewhere to free the arm, undoing the sub-goal `holding(arm1,3)`. A stable sub-goal would be `on(3,1)`. When achieved, the `arm1` resource may be released because no other task will have need to move block `3`.

To avoid *interference* between tasks, the agent programmer must ensure that only tasks with compatible goals are executed concurrently, and that every task atomic procedure has call goals that are stable.

Avoiding arm clashes The ability by both arms to reach over to the shared table means there is a risk that one concurrent task will try to use `arm1` to fetch a block from that table, at the same time as another task uses `arm2` to put down or pickup a block using the table. The arms may then clash. We can avoid this by making the shared table a resource that must be acquired before a task can access it, and by assuming that after putting a block down on the shared table an arm immediately and automatically swings back to its home table if not being used straight away to pick up a block from the shared table. For uniformity of programming we make all three tables resources, even though the home tables cannot be used independently of the arm for which they are the home table.

5 A `TeleoR` tower builder program for an agent controlling two independent robotic arms

The percepts will be facts recording which blocks are directly on a table and which blocks are directly on top of other blocks. We also need percepts recording that an arm is holding a block and the table above which it is currently positioned. We need a recursive `sub_tower` definition that holds when each block on a list of blocks is directly on the next block, except for the last block on the

list which is directly on a named table. A `tower` is then a `sub_tower` such that the first block is clear - has no block on top of it.

We will have durative `pickup`, `put_on_block` and `put_on_table` actions that name the arm and table resources that are to be used. For a `put_on_table(Arm,Tab)` action we assume there will always be a space on `Tab` to put down the held block.

5.1 The tower building ontology

The key definitions we need are given below with explanatory comments.

```
def block ::= 1..16  % blocks are labelled 1 to 16
def arm ::= arm1 | arm2
def table ::= table1 | table2 | shared
def resource == arm || table
% Union def. of resource type. Must be defined when multi-tasking using multiple
% robotic resources. Its values are used as arguments of actions and
% task_atom procedures to indicate the resources that they use.

percept on(block,block), holding(arm,block),
        on_table(block,table), over(arm,table)
def durative ::= pickup(arm,block,table) | put_on_table(arm,table) |
                 put_on_block(arm,block,table)

rel tower(list(block),?table)
tower([Block,..Blocks],Tab) <=
       not exists OnBlk on(OnBlk,Block) & % Block is not covered
       sub_tower([Block,..Blocks],Tab)

rel sub_tower(list(block),?table)
sub_tower([Block],Tab) <=  on_table(Block,Tab)
sub_tower([Block1,Block2,..Blocks],Tab) <=
       on(Block1,Block2) &
       sub_tower([Block2,..Blocks],Tab)

fun other(arm) -> arm
other(arm1) -> arm2
other(arm2) -> arm1

rel can_reach_block(arm,block,?table)
% arm can reach block if it is somewhere on table and arm can reach table.
can_reach_block(Arm,Block,Tab) <=
       somewhere_on(Block,Tab) & can_reach_table(Arm,Tab)

rel can_reach_table(?arm,?table)
can_reach_table(_Arm,shared)  % Either arm can reach shared table.
can_reach_table(arm1,table1)  % Each arm can reach its home table.
can_reach_table(arm2,table2)

rel somewhere_on(block,?table)
% block is either directly on table or is inside a tower on table.
somewhere_on(Block,Tab) <= on_table(Block,Tab)
somewhere_on(Block,Tab) <=
      on(Block,UBlock) & somewhere_on(UBlock,Tab)
```

The pattern `[Block1,Block2,..Blocks]` denotes a list with first two elements `Block1` and `Block2` with `Blocks` being the list of remaining elements. As `OnBlk` only appears in one condition, `not exists OnBlk on(OnBlk,Block)` can simplified to `not on(_,Block)` with anonymous variable _ implicitly existentially quantified inside the negation.

The five defined relations can all be used to check or to find the table argument. This test or generate flexibility is indicated by the `?table` moded type in their type declarations.

5.2 The `makeTower` procedure

The task start procedure is `makeTower` defined below. It has three arguments. The second is the list of blocks to be configured as a tower. The first is the primary arm to be used, and the third is that arm's home table on which the tower is to be built. The program assumes that only blocks that are located somewhere on one of the three tables will be configured as a tower. We also give three of the auxiliary procedures that are called.

```
task_start makeTower(arm,list(block),table)
makeTower(Arm,Blocks,Tab){

  tower(Blocks,Tab) ~> ()   % Call goal achieved, do nothing

  sub_tower(Blocks,Tab) & Blocks=[Blk,..Blks]
       until not holding(Arm,_) ~>
                 makeClear(Blk,Tab)
  % Blk, the first block of Blocks, must be covered. Make it clear. until condition
  % prevents firing of rule 1 until block directly on Blk has been put down on Tab

  Blocks=[Blk] ~> moveAcrossToTable(Arm,Blk,Tab)
  % Should eventually achieve guard of rule 1

  Blocks=[Blk1,Blk2,..Blks] & tower([Blk2,..Blks],Tab) ~>
                  moveAcrossToBlock(Arm,Blk1,Blk2,Tab)
  % Move of Blk1 to be on top of Blk2 should eventually achieve
  % guard of rule 1. Both arms and the shared table may need to be used.

  Blocks=[_,..Blks] ~> makeTower(Arm,Blks,Tab)
  % Recursive call action should eventually achieve guard of rule above.
  }
tel moveAcrossToTable(arm,block,table)
moveAcrossToTable(Arm,Blk,Tab){

  on_table(Blk,Tab) ~> ()

  % Two rules below are while rules as their guards will not be inferable
  % after Blk is picked up, but while condition  holding(Arm,Blk) will be.
  can_reach_block(Arm,Blk,BlkTab) & not over(other(Arm),BlkTab)
      while holding(Arm,Blk) ~>
                 oneArmMoveToTable(Arm,Blk,BlkTab,Tab)
  % BlkTab is Arm's home table or shared. The move to Tab can be
  % done task atomically using resources Arm, Tab and possibly shared.
```

```
  OArm=other(Arm) & can_reach_block(OArm,Blk,BlkTab) &
     not over(Arm,shared)
        while holding(OArm,Blk) ~>
                 oneArmMoveToTable(OArm,Blk,BlkTab,shared)
  % BlkTab is other(Arm)'s home table. Blk must first be task
  % atomically moved to shared using resources other(Arm), BlkTab, shared.

  true ~> ()
  % This rule will fire if the arm that will not be used to pick up Blk is perceived
  % as being over shared until it has automatically moved back to its home table.
  }
tel moveAcrossToBlock(arm,block,table,table)
% Like moveAcrossToTable using oneArmMoveToTable, oneArmMoveToBlock

task_atomic oneArmMoveToTable(arm,block,table,table)
% The arm and possibly two tables, the arm's home table and shared, need
% to be available resources for this task before the procedure is entered.
oneArmMoveToTable(Arm,Blk,BlkTab,Tab){

     on_table(Blk,Tab) ~> ()

     holding(Arm,Blk) ~> put_on_table(Arm,Tab)

     not on(_,Blk) ~> pickup(Arm,Blk,BlkTab)

     true ~> makeClear(Arm,Blk,BlkTab)
     }
tel makeClear(arm,block,table)
% makeClear and oneArmMoveToTable are mutually recursive.
makeClear(Arm,Blk,BlkTab){

      not on(_,Blk) ~> ()

      on(OthrBlk,Blk) until not holding(Arm,OthrBlk) ~>
            oneArmMoveToTable(Arm,OthrBlk,BlkTab,BlkTab)
      % Do not fire rule 1 until OthrBlk has been put down.
      }
task_atomic oneArmMoveToBlock(arm,block,table,block,table)
% Similar rules to the other task atomic procedure
```

The above `makeTower` procedure has the same number of rules as Nilsson's one arm tower builder given in [22]. The guards of the first, third and fourth rules identify the table `Tab` on which the tower or sub-tower is located.

Rules 3 and 4 have calls to procedures `moveAcrossToTable`, `moveAcrossToBlock` respectively, both of which may need to use both arms. If the block to be moved by a call to `moveAcrossToTable` is located on the other arm's home table, it will use two task atomic `oneArmMoveToTable` calls. The first is to transfer `Blk` from wherever it is located on `BlkTab` to `shared`, using `other(Arm)`. The second is to transfer it from `shared` to `Tab`, using `Arm`.

So, when `Blk` has been placed on `shared`, another task can acquire `shared` and/or `other(Arm)` to do some task atomic move needed for the construction of its tower. If this other task needs `shared` and `Arm`, and `other(Arm)` is not

acquired by a task, it will automatically move back to be over its home table. But `other(Arm)` could be acquired by a task just needing to move a block on `other(Arm)`'s home table. The result would be parallel use of the two arms.

We leave the reader to check that all the procedures are universal procedures for their goals. As with the mobile robot `TeleoR` procedures, these procedures will automatically recover from hindrance and take immediate advantage of help.

As an example of recovery from hindrance, suppose that a tower [2,7,3,1] is being built on `table1` and [7,3,1] has already been built on the table. Task `makeTower(arm1,[2,7,3,1],table1)` will call `moveAcrossToBlock(arm1,2,7,table1)` to move 2 from where ever it may be located to be on top of block 7 by firing its 4th rule. Suppose block 2 is located on `shared`. The next call will be `oneArmMoveToBlock(arm1,2,shared,7,table1)`, a task atomic call. The task may now have to suspend waiting for its turn to use the three resources `arm1`, `shared`, `table1`. Whilst it is suspended suppose that someone moves block 2 onto `table2`. Immediately the `moveAcrossToBlock(arm1,2,7,table1)` call will switch to firing its third rule and want to enter the call `oneArmMoveToTable(arm2,2,shared)`, a different task atomic call. As this requires different resources, `arm2` and `shared`, the task may be able to acquire them straight away to put block 2 back onto `shared`.

Regarding taking advantage of help, suppose that whilst waiting to enter the call `oneArmMoveToBlock(arm1,2,shared,7,table1)` an outside party picks up block 2 and puts it on top of 7. Immediately the next batch of percepts arrives recording the new position of block `2`, the initial `makeTower` call will fire its rule 1, task goal achieved.

`TeleoR` Software and Demo Programs The `QuLog`+`TeleoR` software, its documentation, demo programs and Python robot simulations can be downloaded from `http://staff.itee.uq.edu.au/pjr/HomePages/QulogHome.html`. Videos showing `TeleoR` being used to control Python simulated robot arms building block towers, and a Baxter robot using both its arms to build real block towers, are available at `https://www.doc.ic.ac.uk/~klc`. In each case the controlling agent is both helped and hindered. It uses the arms in parallel whenever this can be done without risk of the ams clashing.

6 Related Work

Benson and Nilsson [2] describe a multi-tasking architecture in which `TR` procedures are represented as trees with the regressions represented by branches in the tree. There is a fork in the tree when there are different ways of achieving the guard sub-goal at the fork. Tasks are run one at a time until they achieve a stable sub-goal of their task goal. There is no parallel use of resources.

A `TR` variant for control of a mid-sized robot for RoboCup competitions is described in [12]. Rules can have multiple robotic actions to be executed in parallel but no procedure call actions.

GRUE [11] is a TR architecture especially developed for programming characters in computer games. There is also the concept of a *resource*, although not in the sense that we use that term. A GRUE resource is not a game character, which would be a resource as we use the term, but an artefact such as money or food that can be acquired by a game character.

Choi [4] presents a concurrent extension of the logic based reactive skill description language Icarus [5]. It uses constraints to allocate the resources to tasks. Kinny [17] describes an abstract multi-tasking agent programming language with unordered event triggered rules with logic queries as guards. There is concurrent task execution but no independently useable resources.

GOAL[14] is an agent programming framework that can use a variety of logical representations for the beliefs and knowledge of the agent, although Prolog is normally used. A key component of the agent state is a set of goals that are conjunctions of beliefs that the agent should achieve. There is no concurrency but the achievement of several goals can be serially interleaved.

ConGolog [10] is a concurrent agent programming language based on the situation calculus. Execution can interleave inference selection of actions from a non-deterministic program with additional planing generation of actions. ReadyLog [9] is another variant of GoLog that has program constructs for real time reactive control. It has been used to control robocup soccer playing robots.

Soar is a general purpose agent architecture with its roots in cognitive psychology. It is a very mature system with many man years of development effort. It was the first cognitive agent architecture to be used for robotic control [13].

FLUX [28], LPS [18] and 2APL [8] are logic based approaches to programming single task software agents that can be used for robotic agents. None offer compile time guarantees of type and mode safe inference, and of type correct and ground actions. Others [26], [1] acknowledge the need for type safe agent programming languages. GRL [15] and FROB [23] are typed functional robot programming languages.

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

7 Future Work

Achieve goal actions and event triggered tasks The main planned future work is the incorporation of the concepts from the BDI concept language AgentSpeak(L)[25], and its implementation in Jason [3]. We will extend TeleoR rules so that they can have `achieve Goal` actions in addition to direct procedure calls and tuples of robotic actions. An extra non-deterministic top layer of *option* selection knowledge rules, perhaps of the form

```
for Goal try ProcCall <= BSQuery
```

can then used to find alternative TeleoR procedure calls that will normally, eventually achieve their call goal which is or implies instance Goal′ of Goal, where the fired TeleoR rule action is `achieve` Goal′. The corresponding instance BSQuery′ of

the rule's pre-condition is an extra *Belief Store* query that must also succeed in order for the instance `ProcCall''`, determined by the match againts `Goal'` and a successful `BSQuery'` evaluation, to be tried. An extended operational semantics for `TeleoR` could then allow failure of such `ProcCall`s with backtracking to see if an alternative procedure call might be used for the `achieve Goal'` action.

As in Jason, these same selection rules can be used when the agent is asked to achieve a goal by another agent. They enable inter-agent task requests at the level of a common descriptive ontology for the environment, and do not require other agents or humans to know the names and argument types of the task procedures that can be executed by each agent.

We will also add similar rules for starting tasks whenever a significant *Belief Store* update occurs. These rules generalise the Jason rules for responding to the addition of removal of a single belief, and might have the form

```
on Update do ProcCall <= BSQuery
```

`Update` is a list of `++p`, `--q` terms where `p` and `q` are names of *Belief Store* dynamic relations. They denote update events, `++p` being the event of remembering a new fact for `p`, `--q` being the event of forgetting a fact for `q`. `BSQuery` will be repeatedly tried immediately after *one or more* of these update events has occurred. If it succeeds, the corresponding instance of `ProcCall` will be launched as a new task.

Background threads and task scheduling We believe that adding background activity `QuLog` threads that can learn about the environment, perhaps to construct and/or update a topological map, generalise or abduce beliefs, or discover and repair belief inconsistencies, will enhance the cognitive ability of our robotic agents. One such thread could also respond to goal requests and *Belief Store* updates that trigger tasks.

We will also explore the usefulness of priority scheduling of tasks without task starvation, and the use of knowledge rules to determine when tasks should be suspended and resumed, and when they should be terminated.

References

1. M. Baldoni, C. Baroglio, and F. Capuzzimati. Typing Multi-Agent Systems via Commitments. In *Proc. of the 2nd Int. Workshop on Engineering Multi-Agent Systems (EMAS 2014)*, 2014.
2. S. Benson and N. Nilsson. Reacting planning and learning in an autonomous agent. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence 14*. Oxford University Press, 1995.
3. R. H. Bordini, J. F. Hubner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley-Interscience, 2007.
4. D. Choi. Concurrent execution in a cognitive architecture. In *Proceedings of the 31st Annual Meeting of the Cognitive Science Society. Amsterdam, Netherlands: Cognitive Science Society*, 2009.
5. D. Choi and P. Langley. The Icarus Cognitive Architecture. *Cognitive Systems Research*, 2017.

6. K. L. Clark and P. J. Robinson. Robotic Agent Programming in TeleoR. In *Proceedings of International Conference of Robotics and Automation*. IEEE, 2015.
7. K. L. Clark and P. J. Robinson. *Programming Communicating Robotic Agents: A Multi-tasking Teleo-Reactive Approach*. Springer, 2018. To appear, first 5 chapters on: teleoreactiveprograms.net.
8. M. Destani. 2APL: A practical agent programming language. *Autonomous Agents and Multi-agent Systems*, 16:214–248, 2008.
9. A. Ferrein and G. Lakemeyer. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems*, 56:980–991, 2008.
10. G. Giacomo, Y. Lesperance, and H. Levesque. `ConGolog`, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 1–2(121):109–169, 2000.
11. E. Gordon and B. Logan. A goal processing architecture for game agents. In *Proceedings of AAMAS*, 2003.
12. G. Gubisch, G. Steinbauer, M. Weiglhofer, and F. Wotawa. A teleo-reactive architecture for fast reactive and robust control of mobile robots. *New Frontiers in Applied Artificial Intelligence*, pages 541–550, 2008.
13. S. Hanford, O. Janrathitikarn, and L. N. Long. Control of mobile robots using the Soar cognitive architecture. *Journal of Aerospace Computing, Information, and Communication*, 6(2):69–91, 2009.
14. K. V. Hindriks. Programming Rational Agents in GOAL. In *Multi-Agent Programming: Languages and Tools and Applications*, pages 119–157. Springer, 2009.
15. I. Horswill. Functional programming of behavior-based systems. *Autonomous Robots*, 2000.
16. J. Jones and D. Roth. *Robot programming: a practical guide to behavior-based robotics*. McGraw-Hill, 2004.
17. D. Kinny. The ψ calculus: An algebraic agent language. In *Intelligent Agents VII*. Springer, 2002.
18. R. Kowalski and F. Sadri. Teleo-reactive abductive logic programs. In A. Artikis, R. Craven, N. Kesim, B. Sadighi, and K. Stathis, editors, *Festschrift for Marek Sergot*. Springer, 2012.
19. H. Levesque. *Thinking as Computation*. MIT Press, 2012.
20. M. J. Mataric. *The Robotics Primer*. MIT Press, 2007.
21. J. L. Morales, P. Sanchez, and D. Alonso. A systematic literature review of the Teleo-Reactive paradigm. *Artificial Intelligence Review*, 20(1), 2012.
22. N. J. Nilsson. Teleo-Reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5:99–110, 2001.
23. J. Peterson, G. Hager, and P. Hudak. Language for declarative robot programming. In *International Conference on Robotics and Automation*, 1999.
24. M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System, 2009. At:www.robotics.stanford.edu/∼ang/papers/icraoss09-ROS.pdf.
25. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Seventh European Workshop on Modelling Autonomous Agents in a Multi-AgentWorld*, LNAI, pages 42–55. Springer, 1996.
26. A. Ricci and A. Santi. Typing Multi-agent programs in simpAL. In *Promas*, volume 7837 of *LNAI*. Springer, 2013.
27. P. J. Robinson and K. L. Clark. Pedro: A publish/subscribe server using Prolog technology. *Software Practice and Experience*, 40(4):313–329, 2010.
28. M. Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Springer-Verlag, 2005.