Multi-tasking Robotic Agent Programming in TeleoR

Keith L. Clark
Department of Computing,
Imperial College London

Peter J. Robinson
School if ITEE,
University of Queensland, Brisbane

Abstract

We informally introduce the syntax, operational semantics and implementation aspects of a concurrent multi-tasking extension, `TeleoR`, of Nilsson's Teleo-Reactive (`TR`) rule based robotic agent programming language. 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. The *Action* is either a tuple of primitive actions of robotic resources, to be executed in parallel, or a single call to a `TR` procedure, which can be a recursive call.

`TeleoR`, extends `TR` in having: types and higher order features; extra forms of action rules that temporarily delay the firing of another rule in the same procedure call until some condition is inferable; automatic re-execution of ballistic actions that have not achieved their intended effect in the expected time; rules with belief store updates and message send actions; a flexibly typed fully integrated higher order LP/FP programming language, called `QuLog`, 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, with deadlock and starvation free guarantees.

The focus of this paper is on the multi-tasking features of TeleoR.

1 Introduction

Nilsson's Teleo-Reactive (`TR`) [Nilsson, 2001] agent programming language is a mid-level robotic agent programming language. It assumes lower level routines written in procedural programming languages such as C. Some will do sensor interpretation, particularly for vision, and others will 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 because doing so will achieve some sub-goal of a current task.

`TR` programs are *sequences* of
Guard ~>*Action*
rules clustered into parameterised procedures, each procedure being of the form:

$p(X_1,..,X_k)\{$
$\quad G_1 \sim> A_1$
$\quad .$
$\quad .$
$\quad G_n \sim> A_n$
$\quad \}$

A rule *Guard* is a conjunctive queries, possibly with negated conditions evaluated using the negation-as-failure inference rule [Clark, 1978], to a set of rapidly changing percept facts that are the agent's internal representation of sense data analysis. For example, `see(bottle,left,30)`, a percept recording simple bounding rectangle analysis of a camera image where bottles are all the same size and of fixed colours. The *Guard* may access the percepts via rules defining higher level 'interpretation' relations, and query fixed facts about the environment. A rule *Action* is: a tuple of robotic resource actions to be executed in parallel, for example `move(4.5), turn(left,0.5)`; or a single call to a `TR` procedure, which can be a recursive call.

In each called procedure there is a *current fired rule*. At the top of the call stack the fired rule always has robotic resource actions. They, and any previous executed actions, are used to generate control actions dispatched to the resources to effect changes in the agent's environment. There are both *discrete* and *durative actions*. The latter continue until modified (e.g. `move(4.5)` is modified to `move(4.0)`), or are terminated because the next tuple of actions does not include that action (e.g. `turn(left,0.5)` is terminated).

Typically, initially called `TR` procedures query the percepts facts through several levels of defined relations. Via procedure call actions, they usually cascade down to a `TR` procedure that directly queries the percept facts. For `TR` programs the interface between deliberation and reaction is procedure calling.

When each new batch of percepts arrives, perhaps via a ROS [Quigley *et al.*, 2009] 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 initially called procedure working up the call stack. As soon as a call is found that fires a different rule, or re-fires with different variable bindings the last fired rule, the rest of the call stack is discarded and a new extension built until a rule firing with robotic resource actions. These new actions are compared with the last determined tuple of actions and appropriate start, stop or modify actions are dispatched. There is no new percepts update until the new actions are determined, or it is determined that there is no new rule firing in any current procedure call.

A key desirable feature of the sequence of *Guard* ~>*Action* rules of a `TR` procedure call is that when a rule other than the first rule is fired, its action, whether a `TR` procedure call or a tuple of resource actions should *normally*, *eventually*, bring about a state of the resource environment such that a new batch of percepts coming from the sensors enable the guard of an earlier rule of the procedure call to be fired. Nilsson calls this the regression property. It means that the guards of a `TR` procedure can be put on a sub-goal tree routed at the guard of the first action rule. This guard is the goal of the procedure call. The action of the first rule is often the empty (do nothing action) but it is sometimes and action that maintains the rules guard. If in addition the guards together cover all the situations that can arise when the procedure is called, Nilsson calls this a *universal* procedure for its goal. When executing, there is always a rule that can be fired and its action will normally make progress towards the goal of the call.

1.1 New features 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. To this end `TeleoR` was made a typed language and the `TR` Prolog like *BeliefStore* inference language was made a flexibly typed higher order integrated logic and functional programming language, with modes of use specifications for the relation definitions. So, a `TeleoR` procedure must be given a type declaration of the form:

$\texttt{p:(t}_1\texttt{,..,t}_k\texttt{)}$ *% declaration of the argument types of* p

In addition, the predicates and argument types of the percepts must be declared, as well as the names and argument types of the discrete and durative actions. Every function has the types of its arguments and value declared, and every defined relation has each argument typed and moded.

For simple *BeliefStore* inference the primary modes are *ground input* (variable free term of specified type or a subtype), signalled by a `!` annotation on the argument type, and *ground output*, signalled by a `?` type annotation. The latter means the relation can be queried with a variable, or term of the required type containing variables, in that argument position but that the argument twill become a ground term when an instance of the relation call is found. Functions always return ground values. This typed and moded *BeliefStore* rule language is called `QuLog` [Clark and Robinson, 2015a]. Using this moded type information, the `TeleoR` compiler can check that every rule action will be ground when the rule is fired, and correctly typed. In particular, it can check that resource actions when dispatched will be fully instantiated and type correct. It can also ensure that primitives, such as arithmetic functions, will not generate run-time errors due to incorrectly typed or uninstantiated arguments, which is a quite possible with Prolog.

Message send and *BeliefStore updates* were added as auxiliary actions that can be linked with a rule firing, to be executed once when the primitive actions (eventually) determined by that firing are executed. The former enable the programming of collaborative robotic agent applications and make use of the Pedro publish/subscribe + addressed message router [Robinson and Clark, 2010]. The latter enable the remembering of which actions were executed and when.

Repeatable timed action sequences of the form

```
A1 for T1;...; An for Tn
```

where each `Ai` can be a tuple of resource actions or a `TeleoR` procedure call, were added to handle changes of behaviour triggered by a lapsed time rather than a sensor percept, such as random walking. The sequence is repeated while the rule remains a fired rule.

The last action extension was the *while/repeat* action, primarily for use with discrete resource actions. It has the form *Action* `wait` *T* `repeat` *N*, where *N* is a natural number between 1 and 5. `T` is the time by which `Action` should have succeeded and the *Guard* of the rule above, which is `Action`'s acheivement goal, should have become inferable. If not, the evaluator should re-do the action by sending a new control signal to the robotic resource. This *wait/repeat* cycle should be repeated *N* times. After a final wait of `T` seconds and `error(Action,wait T repeat N)` should be added to the agents *BeliefStore*. This adding of the `error` belief is like an *throw*. The catch can be done in the initial `TeleoR` procedure that is called for the robotic task. It has the form:

```
task_wrapper: trcall, pedro_handle
task_wrapper(ProcCall,Help){
  error(Action,wait T repeat N) ~>
    help(Action,wait T repeat N) to Help
  true ~> ProcCall
  }
```

Now, instead of calling a `TeleoR` procedure such as `collect_bottle` directly, the call `task_wrapper(collect_bottle,ag@bill_phone)` is executed. The first argument is a term of the system generated meta-type `trcall`, the second is a Pedro email style address for an agent process.

When the `task_wrapper` call is executed there will be no `error` belief in the *BeliefStore* so the second default rule will fire. However, if ever `Action wait T repeat N` should cause the error belief to be added, since all rule firings are reconsidered starting with that of the initial call, the first rule of `task_wrapper` will be fired sending a message to `ag@bill_phone)`.

Assume this is a personal agent on a cell phone and that Bill helps by fixing the resource, such as a gripper, used in `Action`. When fixed, via his agent process, Bill sends back a `retry(Action)` message to the agent executing the `task_wrapper` call which is handled by its message handler thread as depicted in Figure 1. This has to have been programmed to respond to such a message be removing the `error(Action,...)` belief. Immediately, the task procedure will be re-called using the second default rule. If the robot resource has not been moved the `Action` will be immediately re-tried.

All the `TeleoR` single task extensions of `TR` are described in [Clark and Robinson, 2015b]. Other extensions are new forms of rules that were added to facilitate the semantically clean programming of certain safety critical systems tasks. There is an `until` rule of the form

Guard `until` *UCond* ~>*Action*

that will inhibit the firing of an earlier rule in a procedure call *until UCond* is inferable. This is used to allow *Action* to over-achieve the guard of the earlier rule that can be fired, to prevent a too early need to re-achieve that guard. Its dual is the *while* of the form

Guard `while` *WCond* ~>*Action*.

This is used to inhibit the firing of rules *below*, which would normally be fired, depending on circumstances, to re-achieve *Guard* if *Guard* becomes non-inferable. This prolongs the continued execution of *Action* until *both Guard* and *WCond* are not inferable. The `while` and `until` conditions can be combined and further constrained my a `min T` condition that gives a minimum time that the inhibition should continue, after rule firing.

1.2 `TeleoR` procedures

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

```
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 components are dropped if they contain vacuous constraints such as a `WC` that is `true`, or a `min` time that is `0`. Here, each A_i is one of: a tuple of resource actions, a single call to a `TeleoR` procedure, a timed action sequence, a `wait/repeat` action.

The `TeleoR` single task agent architecture is naturally multi-threaded and 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.

This corresponds to an agent architecture as depicted in Figure 1. All incoming messages are handled by the message handler and both it and the percepts handler atomically update the agent's `it 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.

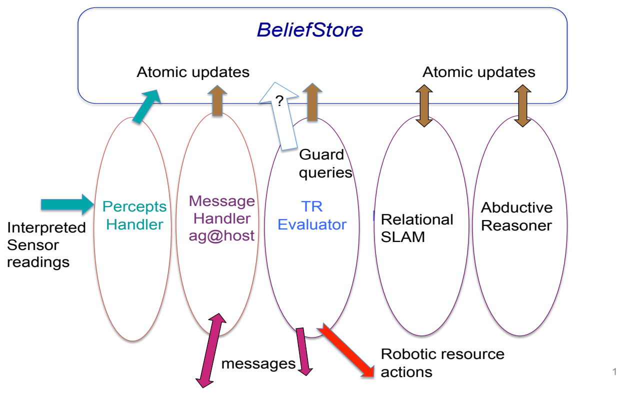

Figure 1: Multi-threaded TeleoR Agent Architecture

By using typed and moded `QuLog`, and by declaring the argument types for each `TeleoR` procedure and for the primitive resource actions, one can guarantee at compile time that each `TeleoR` rule action will be correctly typed and fully instantiated (ground) if the rule is fired. By exploring the defined relation dependences one can determine which percept updates are likely to change the outcome of a guard evaluation. This, and meta-information about how percept predicates have been updated, added to the *BeliefStore* by the percepts handler, enable us to skip reconsideration of rule firings for certain procedure calls. This is an important optimisation when initially called procedures have guards that do complex inference, but which depend upon percepts that change relatively slowly.

1.3 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 it is compiled to allow the relatively easy programming of multi-tasking agents dynamically sharing multiple robotic resources, in much the same way that an operating systems shares hardware resources between processes. This second phase is the primary subject of this paper.

The granularity of the interleaved sharing of resources is specified by the declaration of certain procedures as `task_atomic` and the declaration of which procedures can be the start procedures of a task. We leave the details to the later sections. We exemplify their use with a program for a multi-tasking block tower building robotic agent using two robot arm resources with blocks distributed over three tables. Each arm can only reach two tables and they must both be used to reach to the one table they can both reach to avoid a clash. The `TeleoR` program has a similar sub-goal structure to the one arm, one tower builder of [Nilsson, 2001], but is much more general.

We conclude by mentioning related work, and plans for further extensions. We assume familiarity with logic programming [Levesque, 2012], multi-agent systems [Wooldridge, 2009], and robot behavioural programming

[Jones and Roth, 2004],[Mataric, 2007].

2 Informal `TeleoR` Single Task Evaluation Algorithm

We can describe the evaluation cycle of a task executing a start `TeleoR` procedure call $TaskCall$ with the following 8 step informal algorithm.

$FrdRules$ is the set of indexed active 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. 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 $\boldsymbol{K} \sim> \boldsymbol{A}$, number $R$, with an inferable guard, with $\theta$ being the *first* returned answer substitution for variables of ***eK***.
   Add $(Index, Call, R, \theta)$ to $FrdRules$. No such rule: a **no-fireable-rule** failure.
4. $\boldsymbol{A}\theta$ is a procedure call.
   $Call := \boldsymbol{A}\theta$; $Index := Index + 1$; Go to step 2.
5. $\boldsymbol{A}\theta$ 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. Index:=1;
7. (Optimisation) 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 $\theta$, 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 $\boldsymbol{K} \sim> \boldsymbol{A}$, number $R'$, with an inferable guard, with $\theta'$ being the *first* returned answer substitution for variables of ***eK***.
   (a) No such rule: 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.
```

The call tuple with index $Dp+1$ is the offspring of the call tuple with index Dp and the $Call$ component of the $(Dp+1)^{th}$ tuple is the fully instantiated procedure call action of the rule fired in the Dp^{th} tuple. 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 $\#FrdRules$ when step 5 is executed will record the firing of a rule with primitive actions, control actions for which will be executed by step 5. $MaxDp$ is the maximum number of allowed active calls. $Acts$ is the last tuple of determined actions for $TaskCall$, initialised to $()$. Step 1, followed by an iteration of steps 2 to 4, generate the initial set of $FrdRules$ for $TaskCall$ for the initial state of the *BeliefStore*. This initial iterative generation terminates successfully when a rule is fired with a tuple of primitive actions $A\theta$. $LActs = \{\}$ and the set of actions of $A\theta$ are then used to generate the initial control actions $CActs$ that are executed in step 5. $LActs$ is updated to the new set of actions.

The algorithm then suspends at step 6 until the *BeliefStore* is updated. It then sets $Index$ to 1 and checks each tuple in $FrdRules$ one at a time, starting at the initial entry with $Dp = 1$, to see if a different rule, or a different instance of the same rule, should be fired. Step 7, occasionally augmented with step 8(b) when guards of rules need to be re-checked, is repeated until there is a change of rule firing. Step 7 on its own is an optimisation that uses information about which percepts have been changed as discussed in Section **??**. The 7, 8(b) iteration terminates when either $Index = \#FrdRules$, or step 8(c) finds that there should be another rule firing for some call, or for some call there is now no rule that can be fired (an error condition).

Note there is a new rule firing when a different rule is fired, or the same rule is fired with a different set of bindings θ' for the variables of the rule's e***K***, resulting in different action $\boldsymbol{A}\theta'$. The $Index$ entry of $FrdRules$ is then replaced and all entries on $FrdRules$ above this entry are discarded. The algorithm then switches into the steps that add new entries into $FrdRules$ when we have a sequence of new rule firings with procedure call actions (iteration of steps 4, 2, 3). This continues until a rule is fired with primitive actions (step 5) and new control actions are executed. It then re-suspends at step 6 until there is another *BeliefStore* update.

In other words, on each *BeliefStore* update either there is no change to $FrdRules$, or some entry is replaced by a new tuple which records the firing of a primitive actions rule, or a new sequence of tuples are added recording the firing of rules with procedure call actions until a last entry is added that fires a rule with primitive actions. Whenever there is a change of $FrdRules$ new control actions are computed and executed unless one of the two possible error conditions occurs.

We have step 7, that explicitly tests whether the previously fired rule of $Call$ must continue. Using the meta-beliefs added by the percepts handler about what has been changed, we are sometimes able to determine that this is the case without re-evaluating rule guards of its procedure call.

3 Multi-tasking with multiple resources

An agent that can concurrently execute several tasks using *multiple* resources has an architecture as depicted in Figure 2.

All the tasks threads are active and on each percepts update they reconsider all their fired rules.

The waiting tasks, which will each be waiting to enter some task atomic call with resource arguments, do this in case a changed rule firing somewhere in their call chain results in a task atomic procedure call with different resource needs. This may mean that the waiting task can acquire the resources it now needs. It will know which resources are being used because there are `resources_` facts recording which resources are being used by each running task.

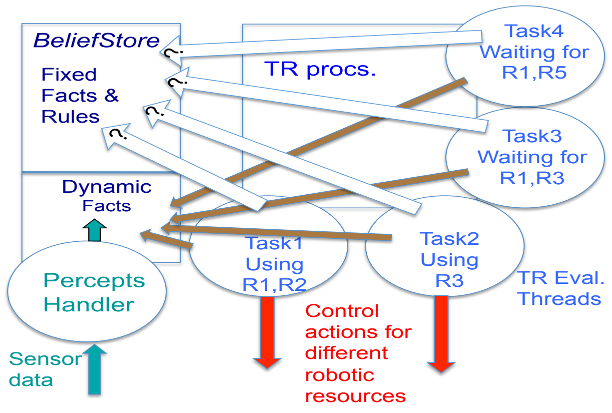

Figure 2: Multi-task `TeleoR` Agent Architecture

Similarly, running tasks - the tasks with acquired resources - may have a new rule firing resulting in an exit of their current initial task atomic call. If this happens, a running task will release its resources by forgetting its `running_` co-ordination fact. It will now want to re-enter some new initial task atomic call, unless it has achieved its task goal. So, normally the former running task will become the most recent waiting task with a `resources_` co-ordination fact in the *BeliefStore* recording the resources for which it is now waiting.

To avoid *deadlock*, the task atomic call that each waiting task wants to enter must have as a resource parameter *each* resource that *might* be used by a call to the procedure. This constraint is checked by the compiler. It means that a running task will only be suspended *after* its initial task atomic call has been terminated and the resources acquired for that call released. We cannot have two running tasks suspended, each with unreleased resources waiting, for a resource acquired by the other task - which is deadlock. It also means that, unless the task 'decides' to prematurely exit an initial task atomic procedure call before its goal is achieved, that acquired resources will be retained until the stable sub-goal of the task atomic call has been achieved.

To avoid *starvation*, a waiting task can only become a running task if none of the resources it needs is in use, or is needed by another waiting task that has been waiting for a longer time. In Figure 2, if we assume that `Task3` has been waiting for its resource needs longer than `Task4`, then even if `Task1` exits its initial task atomic procedure call and releases resource `R1`, `Task3` cannot acquire `R1` even if released by `Task1` as it is needed by `Task3`, that has priority. If, on the next percepts update, `Task3` wants to enter a different initial task atomic call with `R3` and `R6` as resource arguments, and `Task4`'s resource needs stay the same, it can then jump the queue and start using the free `R1` and `R5`.

To ensure a degree of *fairness*, after each percepts update we constrain the task threads so that no waiting task gets to respond to the update until all running tasks have responded, possible releasing resources. In addition, no waiting task can respond until all tasks that have been waiting longer have responded to the update. This means that waiting tasks get to change their minds about resources in wait queue order.

In the above scenario, `Task1` and `Task2` will respond in any order and `Task1` will release `R1` and `R2`. `Task3` will now respond and switch to needing `R3` and `R6`, updating its `resources_` co-ordination fact accordingly. Its wait start time is unchanged. `Task4` will next be able to check if there is a change in its needs. If there is no change it can acquire `R1` and `R5` as both are free and the only task ahead of it in the wait queue does not now need either resource.

`Task4` will 'know' that the other three tasks have responded to the latest percepts, and that it can now respond, as on each update the percepts handler updates a `time_` fact in the *BeliefStore* recording the time of the update. When a task `T` has responded to the new percepts it updates a `seen_` fact that records that `T` has seen the percepts added at the time recorded in the current `time_` fact.

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

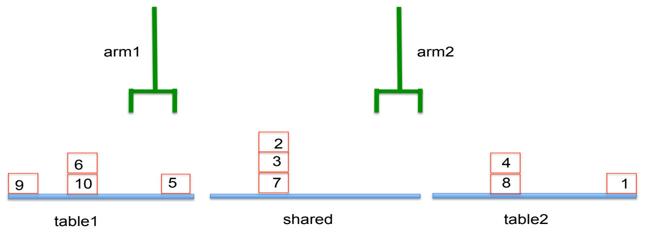

Figure 3: Two Arm Multi Tower Building

The tower building task is as depicted in Figure 3. We need to write a `TeleoR` program that can be used by several concurrent tasks building towers of different blocks either on `table1` or `table2`. These are the home tables of `arm1` and `arm2` respectively. An arm can reach its home table and the shared table. A task building a tower on `table1` will mostly use `arm1`, and a task building on `table2` will mostly use `arm2`. Actions of the two arms can often be executed in parallel. Occasionally, a task `T` building on `table1` will need a block on `table2`. In which case it must wait to acquire `arm2` to first transfer the block from `table2` to `shared` releasing `arm1` for use by another task only needing a block from `table1` or `shared`. When `T` has acquired `arm2` and transferred the block it needs to `shared`, it again waits to acquire `arm1` for the final move from `shared` to `table1`.

This ability by both arms to reach over to `shared` means there is a risk that one concurrent task will try to use `arm1` to fetch a block from shared at the same time as another task uses arm2 to put down or pickup a block using shared. The arms will then clash. We can avoid this by making shared 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 swings back to its home table. For uniformity of programming we will make all three tables resources.

The percepts will be facts telling us which blocks are directly on a table and which blocks are on top of other blocks located on some table. We also need percepts telling us if an arm is holding a block. 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 that is directly on a particular table. A tower is then a sub_tower that has a clear top block - a block with no block on top of it. We will have durative pickup and putdown that include the arm and table resources that are used - the source table for pickup and the destination table for putdown. So we will have actions such as putdown(arm1,2,3,table1) when table1 is the table resource location of block 2, but also putdown(arm2,6,shared,shared) when block 6 is to be putdown on location shared. The second occurrence of shared names the table resource needed, the first the destination location for block 6.

```
block ::= 1..10
arm::= arm1|arm2
tab ::= table1|table2|shared
loc ::= table||block
percept on:(block,loc),holding:(arm,block)
% Definitions of relations. can_reach_block:(arm,block),
% can_reach_table:(arm,table), some_where_on:(block,table)
% sub˙tower:([block],table), clear(block), tower:([block],table),
resource arm||tab   % resource is a reserved type name
durative pickup:(arm,block,tab)
durative putdown:(arm,loc,tab)

task_start makeTower:(arm,[block],table)
makeTower(Arm,[Blk,..Blks],TowerTab){
  tower([Blk,..Blks],TowerTab) ~> ()
  sub_tower([Blk,..Blks],TowerTab) ~>
            makeClear(Arm,Blk,TowerTab)
  Blks=[] ~> moveAcrossToLoc(Arm,
                     Blk,TowerTab,TowerTab)
  tower(Blks,TowerTab)&Blks=[TopBlk,..]~>
           moveAcrossToLoc(Arm,Blk,
                           TopBlk,TowerTab)
  true ~> makeTower(Arm,Blks,TowerTab)
  }
moveAcrossToLoc:(arm,block,loc,tab)
moveAcrossToLoc(Arm,Blk,Loc,LocTab){
  on(Blk,Loc) ~> ()
  can_reach_block(Arm,Blk,BlkTab)
     while holding(Arm,Blk) ~>
       moveToLoc(Arm,Blk,BlkTab,Loc,LocTab)

  can_reach_block(OArm,Blk,BlkTab)
     while holding(OArm,Blk) ~>
        moveToLoc(OArm,Blk,BlkTab,
                             shared,shared)
  true ~> ()
  }
task_atomic moveToLoc
moveToLoc:(arm,block,tab,loc,tab)
makeClear:(arm,block,tab)
```

The new makeTower procedure has the same number of rules as the one arm version. The guards of the first, third and fourth rules identify the table TowerTab on which the tower or sub-tower is located, the home table of the used Arm.

Instead of calls to the task atomic moveToLoc procedure rules 3 and 4 have calls to a procedure moveAcrossToLoc that will use the other arm if need be, and two task atomic moveToLoc calls, to transfer Blk from wherever it is located - on TowerTab, shared or the other home table, to put it onto its required destination location on TowerTab. For rule 3 this is TowerTab, for rule 4 it is TopBlock, the top block of tower Blks.

As always the rules of a call to moveAcrossToLoc will be tested in before/after order. The first rule is its goal achieved rule. The second covers the two cases of Blk being located on the home table of the Arm being used, or shared, as both a reachable by Arm. The guard can_reach_block(Arm,Blk,BlkTab) will check this and bind BlkTab to the table on which Blk is located. The action of the rule is a task atomic call to moveToLoc(Arm,Blk,BlkTab,Loc,LocTab). This will move Blk from wherever it is located on the Arm reachable BlkTab to the reachable Loc on LocTab. It will first make both Blk and Loc clear if need be. Before entry to this task atomic call the Arm resource, and the two table resources BlkTab and LocTab, must be acquired. The table resources may be the same table.

Notice that this rule has a **while** condition holding(Arm,Blk). This is because as soon as Blk is picked up, and new percepts have arrived, the guard of the rule will no longer be inferable. However, holding(Arm,Blk) should be inferable from the new batch of percepts. A **while** rule is one of our TeleoR single task extensions to TR. It allows one to give an alternative to the guard that may only be used *after* the rule has been fired. In this case, it allows the action of the second rule to continue and to complete the transfer *providing* the call to moveAcrossToLoc remains active, i.e. unless outside interference means that the task decides to no longer continue with the block transfer call. This will happen if rule 4 of makeTower has been fired and the TopBlk is removed from the top of the tower Blks. The grandparent makeTower call will take over, terminating the moveAcrossToLoc call and its parent makeTower call, to put back the removed top block of Blks.

The third rule of moveAcrossToLoc deals with the case of Blk being on the other arm's home table. Because the guard of rule 2 will not be inferable, the guard of rule 3 will only succeed with OArm\=Arm. It uses a moveToLoc call to move Blk to shared, to achieve the guard of rule 2. This call must wait to acquire OArm, BlkTab, shared.

5 Related and Future Work

A comprehensive survey of extensions and applications of the teleo-reactive paradigm is given in [Morales *et al.*, 2012].

[Benson and Nilsson, 1995] describes 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. They all use a single resource, or all the available resources - there is no parallel use of resources.

[Choi, 2009] describes an extension of the logic based reactive skill description language Icarus [Choi *et al.*, 2004] for a single task. The extension has concurrent execution of tasks with constraints used to allocate the resources to tasks. [Kinny, 2002] 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.

ConGolog [Giacomo *et al.*, 2000] 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.

[Thielscher, 2005], [Kowalski and Sadri, 2012], [Hindriks, 2009], [Destani, 2008], [Levesque and Pagnucco, 2000] present logic based approaches to programming single task software agents that either have been (the last two), or could be used for robotic agents with varying degrees of efficiency.

None of the above approaches appear to offer compile time guarantees of type and mode safe inference, and of type correct and ground actions. However others, [Ricci and Santi, 2013], [Baldoni *et al.*, 2014], see the need for type safe agent programming languages.

Future Work

The main planned future work is the incorporation of concepts from the BDI concept language AgentSpeak(L)[Rao, 1996] and its implementation in Jason [Bordini *et al.*, 2007]. 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` 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`.

References

[Baldoni *et al.*, 2014] 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.

[Benson and Nilsson, 1995] 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.

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

[Choi *et al.*, 2004] D. Choi, M. Kaufman, P. Langley, N. Nejati, and D. Shapiro. An architecture for persistent reactive behaviour. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-agent Systems*, volume 2, pages 988–995, 2004.

[Choi, 2009] 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.

[Clark and Robinson, 2015a] K. L. Clark and P. J. Robinson. *Engineering Agent Applications in QuLog*. Springer, 2015. To appear.

[Clark and Robinson, 2015b] K. L. Clark and P. J. Robinson. *Robotic Agent Programming in TeleoR*. Proceedings of International Conference of Robotics and Automation, 2015. To appear.

[Clark, 1978] K. L. Clark. Negation as failure. In J. Minker and H. Gallaire, editors, *Logic and Data Bases*. Plenum, 1978.

[Clark, 2014] K. L. Clark. Video of robotic agent controlling 2 arms concurrently building 4 block towers, 2014. Accessible via: teleoreactiveprograms.net.

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

[Giacomo *et al.*, 2000] 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.

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

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

[Kinny, 2002] D. Kinny. The ψ calculus: An algebraic agent language. In *Intelligent Agents VII*. Springer, 2002.

[Kowalski and Sadri, 2012] R. Kowalski and F. Sadri. Teleoreactive abductive logic programs. In Alexander Artikis, Robert Craven, Nihan Kesim, Babak Sadighi, and Kostas

Stathis, editors, *Festschrift for Marek Sergot*. Springer, 2012.

[Levesque and Pagnucco, 2000] H. Levesque and M. Pagnucco. Legolog: Inexpensive experiments in cognitive robotics. In *Cognitive Robotics Workshop, ECAI 2000*, 2000. At: http://www.cs.toronto.edu/cogrobo/Papers/crw00.pdf.

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

[Mataric, 2007] M. J. Mataric. *The Robotics Primer*. MIT Press, 2007.

[Morales *et al.*, 2012] J. L. Morales, P. Sanchez, and D. Alonso. A systematic literature review of the teleoreactive paradigm. *Artificial Intelligence Review*, 20(1), 2012.

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

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

[Rao, 1996] 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.

[Ricci and Santi, 2013] A. Ricci and A. Santi. Typing Multi-agent programs in simpAL. In *Promas*, volume 7837 of *LNAI*. Springer, 2013.

[Robinson and Clark, 2010] P. J. Robinson and K. L. Clark. Pedro: A publish/subscribe server using prolog technology. *Software Practice and Experience*, 40(4):313–329, 2010.

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

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