Robotic Agent Concurrent Task Programming in TeleoR

Keith L. Clark  
Department of Computing,  
Imperial College London

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

Abstract

We present a concurrent multi-tasking extension of Nilsson’s Teleo-Reactive rule based robotic agent programming language (hereafter TR). In this extension, TeleoR, the tasks alternate the use of sets of robotic resources with deadlock and starvation free guarantees. Tasks using disjoint sets of resources can use them in parallel. Our extension is typed. Resources are declared as a type.

The granularity of the interleaving of tasks using overlapping resources is controlled by the programmer by program structuring and declarations of certain procedures as task atomic. Task atomic procedures co-ordinate the use of resources themselves using the agent’s BeliefStore as a Linda co-ordination tuple store. This is transparent to the programmer. TeleoR has other extensions of TR facilitating more fine grained control of a single task, but they are not the focus of this paper. They, and the formal semantics of TR and TeleoR are the topic of other papers.

Introduction

Nilsson’s Teleo-Reactive (TR) (Nilsson 2001b) 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 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. The guards query, sometimes via rules defining ‘interpretation’ relations, a set of rapidly changing percept facts that are the agent’s internal representation of the lower level sense data analysis. A rule action is: one or more robotic resource actions to be executed in parallel; a call to a TR procedure, which can be a recursive call. In each called procedure there is a current fired rule. At the bottom of the call hierarchy the fired rule always has robotic resource actions that are dispatched to the resources to effect changes in the agent’s environment.

When there is a percepts update, the guards of the rules of every called procedure are re-evaluated starting with the top level procedure call working down the call chain. The re-evaluation in each call starts at the first rule working down the sequence of rules of the procedure until a rule R is found with a guard G such that an instance Gθ is inferable from the new percepts. If this is the same rule instance Rθ as was fired last time with a procedure call action PCallθ, this next call in the call chain has its rules re-evaluated. If the action is a tuple of resource actions they are allowed to continue.

If R is a different rule or θ is a different inferred instance of its guard, Rθ is fired and its instantiated action Aθ is executed. If this is a tuple of robotic actions these are used to update the tuple of robotic actions that were determined as response to the previous percepts update. This may involve modifying a durative action that is executing (e.g. changing a forward speed), or stopping an action and starting a new one. If Aθ is a TR procedure call, the partially instantiated rules of this call, where procedure parameters are replaced by the call arguments of Aθ, has its rule guards tested.

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

Figure 1: Double Tower TR Agent Architecture

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. There is no new percepts update until new actions are determined, or it is determined that there is no new rule firing in any current procedure call.

TR has a unique operation semantics. Procedure calls remain active even when the action of their fired rule was a procedure call. Procedures do not terminate themselves, they get terminated when an ancestor call fires a different rule, or a different instance of the same rule.

Each TR procedure achieves a goal usually represented as the guard of its first rule. TR procedures called as actions achieve subgoals of the calling procedure’s goal. The operational semantics means that the behaviour a TR encodes is robust and opportunistic. It automatically recovers from setbacks, redoing actions if need be. It skips actions, if helped. This makes TR well suited for human/robot and robot/robot collaborative applications.

In the rest of the paper we start by giving a slight modification of Nilsson’s block tower building program of (Nilsson 2001b) controlling a robotic arm. We give the program in the TR subset of typed TeleoR using the typed and moded BeliefStore rule language QuLog (Clark and Robinson 2015). You can download a Java applet (Nilsson 2001a) that simulates the behaviour of his program. You can help or hinder by moving blocks and it will respond appropriately. It is a recursive program and makes use of a recursively defined concept of a stack of blocks. The percepts record whether a block is directly on top of another block, or on the table.

By using typed and moded QuLog we 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 we can determine which percepts updates are likely to change the outcome of a guard evaluation. This, and meta-information about how percepts 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.

By adding two statements to this program, a task_start and a task_atomic statement, it becomes a program that can be used by an agent building several towers of different blocks, interleaving the use of the arm between the tasks. The agent will put one new block on each tower before switching to another task.

A task atomic assertion for a procedure in a TeleoR program file means that we compile the procedure in a different way so that the concurrent tasks decide for themselves, in a co-operative way, when they can enter a task atomic procedure call, and thus take over control of the robotic resource to try to achieve the goal of that call by resource actions. A task releases the resource when it exits the procedure should there be a waiting task.

This polite co-operative behaviour is achieved by having the tasks atomically query and update a set of special co-ordination facts in the agent’s BeliefStore in accordance with a co-ordination policy. There is no central coordinator.

We then give a resource parameterised version of the Nilsson’s program. It becomes a program executed by each tower building task of an agent using two arms and three tables as resources to interleave concurrent tower building tasks where, because two arms are used, we can have parallel building of towers. The individual tasks are still robust and opportunistic. They can be helped or hindered.

We conclude by mentioning related work, some single task programming extensions in TeleoR we have not illustrated, 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).

Nilsson’s Tower Building Program

The program assumes visual analysis routines that can recognise a block shape and read a unique number on the block, its identifier. The percepts that arrive at the agent as a result of this analysis are a set of on facts, such as on(6,1) indicating that block numbered 6 is directly on top of the one numbered 1, and on(1,table), indicating that block 1 sits directly on the table. A typical configuration is depicted in Figure 2. Another percept predicate, holding, is received if the arm is currently holding a block, e.g. holding(?). makeTower only uses defined percepts and procedure call actions. It is a deliberative procedure at the top of the action tower of this program. As in Prolog, names beginning with an upper case letter or underscore, are variables.

```
block::= 1..9 %Type block an integer between 1 and 9
tab::= table % Enumerated type just containing table
loc::= block||tab % Union type for block location

percepts holding:(block),on:(block,loc)
% Type declaration of dynamic percept facts

durative pickup:(block), putdown:(block)
% Declaration of the stoppable actions and their arg. types

stack:(!block) % A moded type declaration
% Mode annotation ! means list of blocks arg. must be given
stack([Blk1])<=on(Blk,table)
stack([Blk1,Blk2,...,Blks])<=
on(Blk1,Blk2) & stack([Blk2,...,Blks])
% Read <= as if and & as and

tower:(![block])
tower([Blk,...,Blks])<=
clear(Blk)& stack([Blk,...,Blks])
```

Figure 2: Typical configuration of blocks
A tower is a block stack with top block clear

\[ clear(\text{top}) \]

\[ clear(Blk) \iff \neg \text{holding}(Blk) \land \neg \text{on}(\_, Blk) \]

\[ \text{durative} \quad \text{pickup}:(\text{block}) \text{, putdown}:(\text{block}, \text{loc}) \quad \% \text{The stoppable arm actions} \]

\[ \text{makeTower}([\text{block}]) \quad \% \text{makeTower type decl.} - \text{argument a list of blocks} \]

\[ \text{makeTower}([\text{Blk}, \ldots \text{Blks}]) \{ \]

\[ \% \text{Argument pattern is for a list with first element Blk} \]

\[ \% \text{followed by list Blks} \]

\[ \text{tower}([\text{Blk}, \ldots \text{Blks}]) \Rightarrow () \]

\[ \% \text{List of blocks is configured as a tower. Do nothing} \]

\[ \text{stack}([\text{Blk}, \ldots \text{Blks}]) \Rightarrow \text{makeClear}(\text{Blk}) \]

\[ \% \text{[Blk..Blks] will be a tower if Blk is cleared} \]

\[ \text{Blks}=[] \Rightarrow \text{moveToLoc}(\text{Blk}, \text{table}) \]

\[ \% \text{Need to move Blk to table to make a one block tower} \]

\[ \text{tower}(\text{Blks}) \land \text{Blks}=[\text{TopBlk}, \ldots] \Rightarrow \]

\[ \text{moveToLoc}(\text{Blk}, \text{TopBlk}) \]

\[ \% \text{Move Blk onto clear TopBlk to complete tower} \]

\[ \text{true} \Rightarrow \text{makeTower}(\text{Blks}) \}

\[ \% \text{Else first recursively configure Blks as a tower} \]

\[ \text{makeClear}(\text{Blk}) \{ \% \text{Recursive, via moveToLoc} \]

\[ \text{clear}(\text{Blk}) \Rightarrow () \]

\[ \% \text{Goal is achieved} \]

\[ \text{on}(\text{OnBlk}, \text{Blk}) \Rightarrow \text{moveToLoc}(\text{OnBlk}, \text{table}) \}

\[ \% \text{TopBlk to complete tower} \]

\[ \text{moveToLoc}(\text{block}, \text{loc}) \]

\[ \text{moveToLoc}(\text{Blk}, \text{Loc}) \{ \% \text{Moves Blk from wherever} \]

\[ \text{on}(\text{Blk}, \text{Loc}) \Rightarrow () \]

\[ \% \text{it is to be on Loc} \]

\[ \text{holding}(\text{Blk}) \land \text{clear}(\text{Loc}) \Rightarrow \]

\[ \text{putdown}(\text{Blk}, \text{Loc}) \]

\[ \text{not} \text{holding}(\_ \land \text{clear}(\text{Loc}) \land \text{clear}(\text{Blk}) \Rightarrow \]

\[ \text{pickup}(\text{Blk}) \]

\[ \text{not} \text{holding}(\_ \land \text{clear}(\text{Loc}) \Rightarrow \]

\[ \text{makeClear}(\text{Blk}) \]

\[ \% \text{Need to clear Blk to pick it up} \]

\[ \text{not} \text{holding}(\_ \Rightarrow \]

\[ \text{makeClear}(\text{Loc}) \]

\[ \% \text{Loc must be an unclear block} \]

\[ \text{holding}(\text{B}) \Rightarrow \text{putdown}(\text{B}, \text{table}) \}

\[ \% \text{putdown any other block being held onto the table} \]

In the second rule of makeTower we do not have a condition that the top block Blk of the stack is not clear. It is implicit in that the TR evaluator will only test this stack guard if the tower guard of the first rule is not inferable. So we know that [Blk, .. Blks] is not a tower when the second guard is tested, hence that Blk cannot be clear if [Blk, .. Blks] is a stack.

The tower to be built is identified by a given list of integer block labels [Blk, .. Blks]. So a call makeTower([6, 4, 8]) will have to significantly re-configure the initial situation as depicted in Figure 2 to get 8 onto the table, 4 on 8, then 6 on 4.

For the configuration of Figure 2 the last rule of the procedure call will be fired which recursively calls makeTower([4, 8]). The last rule of this call will be fired giving the recursive call makeTower([8]). This time rule 3 of makeTower([8]) call will be fired and 8 moved to the table. The parent makeTower([4, 8]), which remember remains active, will now fire its 4th rule to move 4 to be on top of 8. This will put block 5 onto the table to make 4 clear. When clear, 4 can be put on top of 8 providing 8 is still clear. If an outside party has moved a block to be on top of 8, for example block 2, 2 must first be moved to the table to clear 8 by firing its 5th rule of moveToLoc, instead of its 4th rule. Finally, 6 will be made clear and put onto 4, providing 4 remains clear and on top of 8. If an outside party moves 4 to be on the table the parent makeTower([4, 8]) will take over and fire its 4th rule again to put 4 back on top of 8.

All the procedures satisfy Nilsson’s regression property. That is, the action of all but the first rule of each procedure is such that it will normally eventually bring about a state of the environment such that received percepts will enable an instance of the guard of an earlier rule of the procedure to become inferable. In addition, each procedure is such that there is always a rule that can be fired, their guards cover all situations. Nilsson calls procedures with both properties complete procedures for their goal, the guard of the first rule.

As an example, makeTower has a last default rule with guard the always inferable true. So there is always a rule to fire. The recursive call action of this last rule should eventually bring about a situation in which the guard of the 4th rule is inferable. Its action should achieve the procedure goal, the guard of the first rule. Similarly, rules 2 and 3 have actions that should eventually bring about the goal of the procedure.

The mutual recursion between makeClear and moveToLoc climbs up the sub-tower of blocks above the block to be cleared until the top block is reached and moved to the table. The parent moveToLoc calls then fire their rules 3 and 2 in turn to move each block to the table.

Building multiple towers

If an agent uses the above program to build two towers of different sets of blocks at the same time, by launching two evaluation threads, the tasks will interfere with one another even if the task threads are giving alternating time slices on a single processor.

Assume we start with blocks 1 to 5 all on the table with no blocks on top of them, and that the two tasks task_123, task_45 are to build towers [1,2,3] and [4,5] respectively. Suppose that task_123 gets the first time slice. It will do its first action pickup(2) and then suspend waiting for a percept that confirms it is holding(2). Immediately after, almost before the action has started, task_45 will respond to the first batch of percepts and send the action pickup(4) to the arm. The arm will switch to picking up 4.

task_45 has interfered with task_123 by preventing it from achieving the sub-goal holding(2), but more importantly, it has stopped it achieving the sub-goal on(2,3). If this on subgoal had been achieved by task_123 it would not have been undone by any actions of task_45, as they are using different blocks. on(2,3) is a stable sub-goal (Benson and Nilsson 1995) of the overall task goal of configuring blocks [1,2,3] as a tower. We need to make the execution of a call to move_to_loc atomic (not interruptible) with respect to the task inter-leavings. It needs to be a task atomic procedure.
When task_123 first enters this task atomic procedure it will remember (put into the BeliefStore) running,(task_123). Another task cannot enter the task atomic call if there is such a running belief. As all the arm actions are executed inside this procedure, or procedures it calls, this means no other task will try to control the arm. Other tasks will suspend and remember waiting,(task_45,WT), with WT the suspension time.

When task_123 exits its initial call to mov_to_loc, for whatever reason, and wants to enter a new initial call to the same procedure, it checks to see if there is a waiting task. If so it relinquishes control by forgetting its running belief. As there is no running belief with an earlier time stamp, it replaces its waiting belief by running,(task_45), and enters this initial task atomic call.

The tasks co-ordinate using the agent's BeliefStore in the manner of a Linda (Carriero and Gelernter 1989) tuple store, allowing the interleaving of any number of tasks.

**Procedure Declarations for Multi-Tasking**

We do not need to change any of the procedures we gave above to achieve the required fair and non-interfering interleaving of the tower building tasks. All we need do is add two statements to specify which procedures are task atomic and which procedures may be called to start a new task.

**Task atomic procedures** We declare that mov_to_loc should be used as described above, just by adding a task atomic mov_to_loc statement to our TeleoR program.

**Allowed Start Procedures** In addition to task atomic statements, the program file for a multi-tasking TeleoR agent must also include a declaration as to which TeleoR procedures are allowed to be called as the start procedures of a task. For example, the program file might include task_start makeTower telling the compiler that only calls to this procedures may be start calls of concurrent tasks of our tower building agent.

The task_start statement is used by the compiler to check that no rule with robotic resource actions appears in any procedure that might be called before a task atomic procedure. If not, resources might be used before they have been acquired, which happens only on entry to a task atomic procedure.

**Initial task atomic calls** Let us suppose that a task has fired a rule that wants to enter a task atomic procedure for the first time. It may have to wait until the arm is free and it is at the head of the queue of tasks waiting to use the arm. As soon as it has control of the arm it will start firing the primitive action rules of the procedure.

Suppose the moveToLoc procedure is being used when the block to be moved is not clear so makeClear is called to clear the block. This in turn will fire its second rule that calls moveToLoc where the destination is the table. This is another call to a task atomic procedure, but one called from inside an initial call to moveToLoc. It is still executing that initial procedure call, and will therefore still have control of the arm. It may enter the new moveToLoc call without any suspension. A task needs to be aware of whether or not it is already inside a task atomic procedure call, and when it is about to enter such a procedure call. This is achieved by compiling task atomic procedures in a special way, with changes to the task evaluator.

The querying and manipulation of the co-ordination facts is opaque to the TeleoR programmer. It is done by code generated by the compiler. The programmer chooses the atomicity of the interleaving by program structuring and choice of the task atomic procedures - there may be several. As an example, a slight restructuring of the above program results in the fine grained alternation of the use of the arm between tasks after each block move, for example a move of a block to the table in order to make another block clear.

**Multi-tasking with 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 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 3: Multi-task TeleoR Agent Architecture](image-url)
oordination 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 shorter time. In Figure 3, 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 coordinator 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.

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

The tower building task is as depicted in Figure 4. 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. For uniformity of programming we will make all three tables resources.

Our BeliefStore rules need some modification. We need stack and tower to have a second argument giving the table location of the stack or tower. The holding percept needs to have an extra argument giving the identity of the holding arm. We will also generalise pickup and putdown so that they 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.

```plaintext
block ::= 1..10
arm::= arm1|arm2
loc ::= table1|table2|shared
percept on:(block,loc),holding:(arm,block)
  % Definitions of relations. can_reach:loc1(loc2,arm,block)
  % can_reach:table(loc1,table,arm,block)
resource arm||tab % resource is a reserved type name
  % resource is a reserved type name
durative pickup:(arm,block,tab)
durative putdown:(arm,loc,tab)

makeTower(arm,[block],table)
makeTower(Arm,[Blk,..Blks],TowerTab)
  tower([Blk,..Blks],TowerTab) "~> ()
```

Figure 4: Two Arm Multi Tower Building
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 stack 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 \( \text{can}_\text{reach}_\text{block}(\text{Arm}, \text{Blk}, \text{BlkTab}) \) 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(\( \text{Arm}, \text{Blk}, \text{BlkTab}, \text{Loc}, \text{LocTab} \)). 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 \( \text{while} \) condition holding(\( \text{Arm}, \text{Blk} \)). This is because as soon as Blk is picked up, and new percepts have arrived, the guard of the rule will no longer be infeasible. However, holding(\( \text{Arm}, \text{Blk} \)) should be infeasible from the new batch of percepts. A \( \text{while} \) 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 \( \text{providing} \) 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 infeasible, the guard of rule 3 will only succeed with \( \text{OArm} = \text{Arm} \). It uses a moveToLoc call to move Blk to shared, to achieve the guard of rule 2. This call must wait to acquire \( \text{OArm}, \text{BlkTab}, \text{shared} \).

See (Clark 2014) for a video simulation of essentially this program concurrently building four towers. The program of the video actually has a concurrent task help another task if it can do so at no cost to itself. For example, when clearing a block on shared if a block B is picked up to be put down on shared, it is instead put down on the arm's home table if this is an acquired resource and the block is in the list of blocks to be configured as a tower on that home table by another task. The co-operative behaviour is possible because of task beliefs maintained by the agent recording the start calls of each task. If the arms were controlled by different agents such co-operation would only be possible if the agents told one another about their current tasks.

### Related and Future Work

A comprehensive survey of extensions and applications of the teleo-reactive paradigm is given in (Morales, Sanchez, and Alonso 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, Lesperance, and Levesque 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 planning 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, Baroglio, and Capuzzimati 2014), see the need for type safe agent programming languages.

TeleoR has several other extensions that we added to facilitate the semantically clean programming of a task. There is an until rule that is the dual of a while rule. It inhibits the firing of rules above until some condition is inferable. Actions can also be sequences of timed limited durative actions and procedure calls. The sequence is repeated while the rule remains a fired rule. Finally, resource actions can have linked BeliefStore update actions and message send actions to other agents. The latter enable the programming of collaborative robotic agent applications.

Future Work Our main planned future work is the incorporation of concepts from the BDI concept language AgentSpeak(L)(Rao 1996) and its implementation in Jason (Bordini, Hubner, and Wooldridge 2007). We will extend TeleoR rules so that they can have achieve Goal actions as well as direct procedure calls. An extra nondeterministic 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


