TR-Turtles: Simulating Teleo-Reactive Agents in NetLogo

Manual

Vasileios Apostolidis-Afentoulis and Ilias Sakellariou

February 1, 2023

If you come across any problems, feel free to contact us at:

vapostolidis@uom.edu.gr , iliass@uom.edu.gr

Have fun with NetLogo !

Contents

1 Introduction

Teleo-Reactive agents, are software entities that continuously sense the environment, and based on a set of reactive rules trigger actions, whose continuous execution eventually leads the system to satisfy a goal. The Teleo-Reactive approach provides a rather simple way of modelling model complex goal-driven agents.

The Teleo-Reactive approach was originally introduced by Nilsson [Nilsson, 1993], aiming to merge control theory and computer science notions, in order to offer an elegant way to encode agent behaviour in dynamic environments. Thus, a Teleo-Reactive sequence (TR) is an ordered list of production rules, guarded by environment conditions (K_i) that initiate actions (a_i).

$$\begin{gathered} K_1 \rightarrow a_1 \\ K_2 \rightarrow a_2 \\ \ldots \\ K_m \rightarrow a_m \end{gathered}$$

Rule ordering dictates a *top-down firing priority*, i.e. rules appearing *early* in the sequence have a higher priority. Thus, the condition K_m implicitly contains the negation of all previous rule conditions $\{K_j | j < m\}$.

Goal directed behaviour (*teleo*) is achieved by appropriately designing TR sequences to satisfy the *regression* property [Nilsson, 1993, Nilsson, 2001]: each lower priority rule action is expected to achieve the condition of a higher priority rule, gradually leading to the achievement of the top-level goal, appearing as a condition of the first rule.

TeleoR [Clark and Robinson, 2015] is an extension of the Nilsson's Teleo-Reactive paradigm, implemented in an extended logic programming language Qu-Prolog (Quantifer Prolog) [Staples et al., 1989], that introduces time sequenced actions, repeat/until rules etc.

An excellent source for more information can be found in https://teleoreactiveprograms.net.

1.1 The TR-Turtles Meta-Interpreter

The present manual documents the TR-Turtles syntax and meta-interpreter, that aims to allow the modeler to encode Teleo-Reactive agents in the NetLogo simulation platform.

The design goals that underline the TR-Turtles approach are as follows:

- offer modelers the ability to encode complex agent behaviour by embedding teleo-reactive rules in NetLogo syntax,
- low installation cost and an implementation approach that will survive the frequent evolutions of the NetLogo platform, with minimal to none changes,
- ease of use, small learning curve and compliance with the interactive style of model development in NetLogo and well as to its modeling approach.

To meet the above requirements TR-Turtles was implemented *entirely in the NetLogo language* and comes in the form of `*.nls` files to be included in the modeler's simulation code.

The overall architecture of TR-Turtles is presented in Fig. 1. Briefly, the modeler has to provide the simulation environment, including the agent sensors and effectors as NetLogo reporters and procedures. In order to specify the behaviour of a set of agents (breed in NetLogo terms) using TR-Turtles, the modeler:

1. defines the former in a NetLogo procedure using the syntax described in Fig. 3 (populating the *Rule Store* with the TR rules), and
2. provides the belief update "callback" function translating sensory input to beliefs to update the *Belief Store.*

During the simulation, the *execution of the agent* is then controlled by TR-Turtles: in each simulation cycle the meta-interpreter (*TR Execution Engine*) invokes the belief "callback" function, determines and executes the respective agent actions.

2 Getting Started

2.1 Obtaining TR-Turtles

Installation requirements were kept to a minimal, since the code of the meta-interpreter comes in the form of three `.nls` files:

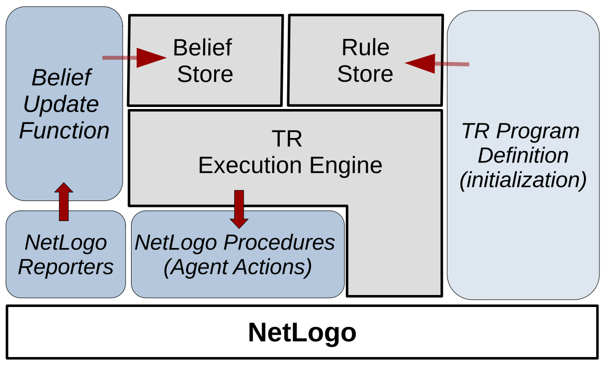


Figure 1: The overall TR-TURTLES architecture.

```
./tr/teleoTurtles.nls
./tr/teleoTBeliefs.nls
./tr/teleoTSyntax.nls
```

Thus, in order to "install" TR-TURTLES the modeler only needs to download the files above from the public GitHub repository [1] and include the corresponding folder (`tr`) in the development folder of their simulation.

For example, assuming that the top modeler folder is called "mySimulation" that contains an "agentSim.nlogo" NetLogo file, the structure of the folders and files should be like this:

```
mySimulation/agentSim.nlogo
mySimulation/tr/teleoTurtles.nls
mySimulation/tr/teleoTBeliefs.nls
mySimulation/tr/teleoTSyntax.nls
```

After the above "installation", the modeler must only *include*[2]: the `./tr/teleoTurtles.nls` file in their model while the other two files, (i.e. `teleoTBeliefs.nls` and `teleoTSyntax.nls`, are automatically included from the first. Thus the following line must be included in the simulation:

```
  __includes [ "./tr/teleoTurtles.nls" ]
```

In order for a breed of turtles (for instance `robots`) to be encoded as Teleo-Reactive agents, a breed specific variable under the name *teleor-store* must be defined as the example below:

```
  robots-own [teleor-store]
```

The `teleor-store` turtle variable holds the main data structure of the TR-TURTLES, i.e. compiled TR rules, necessary execution information and their `belief-store`, i.e. agent beliefs about the environment.

Agent behaviour (i.e. a TRProgram) is encoded in a user defined NetLogo procedure, whose form is the following:

```
⟨TRProgram⟩ ::= "tr-init" ⟨BeliefsD⟩ ⟨ActionD⟩ ⟨Procedures⟩
```

The above includes the initialization of the Teleo-Reactive specification ("tr-init"), the declaration of the *belief-update-function* (BeliefsD), the declaration of the *durative* and *discrete* actions (ActionD) and finally the (Procedures) that set the rules, goals and include the *Guarded Rules*, ensuring the achievement of a goal.

For instance, assuming that the modeler wants to define the TR code of the breed `robots`, the corresponding procedure will have the following form:

```
to tr-code-of-robots
  tr-init
  ;; Specify the user defined belief update function
  belief-update-function [[] -> update-robot-beliefs]
  ;; Specify the set of beliefs
  beliefs ["holding" "see-depot" ... ]
  ;; Specify the set of durative actions
  durative-actions ["move-forward" "rotate" .. ]
```

[1]https://github.com/isakellariou/teleoTurtles

[2]For more details about the `__includes` command see NetLogo manual.

```
  ;; Specify the set of discrete actions
  discrete-actions ["ungrasp" ... ]
  ;; Procedure Specification (i.e. a set of rules)
  procedure "clean-cans"
    # "holding" & "see-depot" & ... --> ["move-forward"]  .
    # "holding" --> "wander" .
    ...
    end-procedure
  ;; Another Procedure
  procedure "wander"
    ...
    # "true" --> "rotate" .
  end-procedure
  ;; setting the top level goal (active procedure).
  set-goal "clean-cans"
end
```

The common `setup` procedure of NetLogo simulations, must include TR-Turtles initialization. In this example, this is achieved by calling the above `tr-code-of-robots`, that is responsible performing any initialization on the TR structure, and "compile" the TR rules that constitute the TR-Program of the `robot` agent, in a form "executable" by the meta-interpreter. For instance, a simple `setup` procedure would look like:

```
    to setup
        ca
        reset-ticks
        create-robots 1 [
            tr-code-of-robots
        ]
```

The observant reader would notice that the `reset-ticks` NetLogo command is included in the above procedure. The TR-Turtles language depends on the NetLogo *ticks* to represent time, for instance in *time sequenced actions* (see Section 3.4), thus it is mandatory to enable ticks in the simulation. In the following, the terms *ticks* and *time units* are used interchangeably.

Execution of the Teleo-Reactive specification for the breed (i.e. code in the commonly named `run-experiment` NetLogo Procedure), is simply asking the breed to `execute-rules`, i.e. the top-level procedure of the meta-interpreter via the standard NetLogo primitive *ask*. For example:

```
    ask robots [execute-rules]
```

Belief update is handled by the `execute-rules` procedure, i.e. the latter calls the anonymous user defined procedure specified in the `belief-update` annotation of the TR-Program. The main purpose of the user defined procedure is to check the agent's latest percepts and store the *beliefs* of the Teleo-Reactive agents in the *teleor-store*.

The sections that follow describe in more detail the different parts of the TR-Program and informally present its semantics. An full version of the grammar in BNF can be found in Section B in Fig. 3.

3 Basic Language Syntax

In this section, we describe the syntax of the TR-Turtles language, explaining in greater detail how the different components of the language can be used to specify the TR agent behaviour.

3.1 Turtle Variables per Teleo-Reactive Agent

Each Teleo-Reactive agent breed must at least have the `teleor-store` variable defined and any other variable that the modeler wants. In the first example, the *robots* have only one variable, `teleor-store`, while in the second example, the *shepherds* have three, `carried-sheep`, `found-herd?` and `teleor-store`:

```
    robots-own
    [
        teleor-store
    ]
    shepherds-own
    [
      carried-sheep
      found-herd?
      teleor-store
    ]
```

3.2 Agent Beliefs, Updates and Perceptions

The handling of agent percepts must easily interface the agent with the NetLogo environment while supporting the semantics of the TR approach. Thus, the modeler has to provide a belief update command and include in the latter the detection of all the necessary environmental "events" that the agent should observe (Fig. 1). Both the *belief update function* and the set of beliefs (for type checking) have to be declared in the TR-Turtles program (Fig. 3).

The set of agent beliefs is declared by the following declaration:

```
     beliefs  <List of Strings Corresponding to Beliefs>
```

For instance:

```
     beliefs  ["can-move-ahead" "see-can" "touching"]
```

The *user defined* belief update function is specified in the corresponding declaration:

```
     belief-update-function <Anonymous NetLogo procedure>
```

It should be noted that the meta-interpreter expects *a NetLogo anonymous procedure* in the declaration, that will be stored in the appropriate internal meta-interpreter structure and called in each cycle *automatically.* For instance:

```
    belief-update-function [[] -> update-robot-beliefs]
```

The role of the anonymous procedure is the detection of such an environmental event, and in such a case the modeler has to update the store with the corresponding belief or inform the store that the specific belief should be removed. The modeler can achieve the latter through two commands that handle the belief addition/removal in the store and record whether a change occurred:

```
    add-belief <belief String Representation>
    no-belief <belief String Representation>
```

Both commands take as an argument a string representation of the belief, that must be declared previously in the corresponding declaration, or an error will be yield by the meta interpreter. This provides a minimum type checking functionality. It should be noted that currently beliefs carry no arguments (expected to be extended in a later version) and are simply true or false, with `add-belief` making the corresponding belief true and `no-belief` making it false. If any command changes the truth value of a belief, then a flag indicates *a change in the environment* that necessitates re-evaluation of the TR Program, possibly to find a new rule to be fired.

In the example below, the robot is checking if there is a `can` that can be seen within a radius of 1. If the condition is true, then it adds the *belief* (`"touching"` otherwise, the condition is false and the belief is removed.

```
to update-robot-beliefs
    ifelse any? cans in-radius 1
          [add-belief "touching"]
          [no-belief "touching"]
    ifelse ...
    ...
end
```

There are no limitations with respect to the NetLogo code included in the anonymous procedure; thus, the user is free to include any agent perception mechanisms.

3.3 Discrete and Durative Actions

Actions are string representations of NetLogo procedures, i.e., the effectors of the agent, possibly arranged for parallel or sequential execution. Given that the NetLogo simulation environment supports the notion of a time unit (*tick*), typically advancing in each simulation cycle, we assume that a *discrete* (or *ballistic*) action lasts for a single time tick, i.e., a single simulation circle. On the other hand, a *durative* action can span its execution over multiple cycles, i.e., creating a sequence of "discrete instances" of the action that are executed until the rule is deselected.

Thus, it is important that the modeler clearly define the type of the corresponding action. This is achieved by having *action mode declarations* in the TR agent specification, listing each set of durative and discrete actions (Fig. 3).

```
durative-actions <List of String Representations of Agent Actions>
discrete-actions <List of String Representations of Agent Actions>
```

For instance, the following declarations include the list of actions that the `robot` can execute:

```
durative-actions ["move-forward" "rotate"]
discrete-actions ["ungrasp" "grasp" "blink"]
```

A consequence of the distinction between durative and discrete actions, is that all agent actions must be user defined NetLogo procedures. For instance, even if the modeler wants to invoke a simple agent action, such as `fd 1`, he/she has to wrap it inside a procedure:

```
to move-forward
  fd 1
end
```

However, although this can seem to be rather non-intuitive, poses no limits as to what actions can be encoded, since a procedure can any NetLogo code.

3.4 Procedures and Guarded Action Rules

Probably the most important component of TR-Turtles is the guarded action rule, which has the following form:

# Condition --> Actions ++ [Side Effects] .

A TR rule starts with the `#` and ends with a `.` . Please note that both after the hash and before the full-stop a space is required. A *Condition* is a possible conjunction (&) of strings (beliefs), each representing a potential match to a *belief* that exists in the Belief Store. The "right hand side" (RHS) of the rule (*Actions*) can be, either:

- A single action
- A list of "parallel' actions
- A time sequenced action
- A *wait-repeat* action
- A single procedure call

Each of the above will be presented in more detail in the following.

3.4.1 Single Action

The simplest case is to have a *single action* in the RHS, i.e. a simple string. For instance, the following rule can be used to make the robot move forward.

```
# "can-move-ahead" --> "move-forward" .
```

3.4.2 Actions Executed in Parallel

A list of actions enclosed in "[" and "]" are executed in *parallel*, e.g., `["move-forward" "rotate"]`. Parallel in the case of a simulation, means that all actions are to be executed in a single time step (tick). Placing actions in such a list implies that the modeler considers that such actions can indeed be executed in parallel by the agent; this is a modeling choice. For instance, in the following rule, when the condition of the guarded rule `"can-move-ahead"` is true, the robot will `"move-forward"` and then `"rotate"`, in a single time step.

```
    # "can-move-ahead" --> ["move-forward" "rotate"] .
```

A list of parallel actions, can contain both discrete and durative actions. In the first execution of the corresponding rule, all actions get executed in a singe time step (thus in parallel), however, in subsequent executions only durative actions in the list are executed. For instance, consider the parallel action:

```
  "can-move-ahead" -->  ["move-forward" "blink" "rotate"]
```

where `"move-forward"` and `"rotate"` are durative actions and `"blink"` is a discrete action. In the first execution of the rule all the actions are executed, i.e.:

```
  ["move-forward" "blink" "rotate"]
```

However, if the rule remains active, then in the following steps, the action sequence contains only durative actions, i.e. only the latter persist their execution, thus:

```
  ["move-forward" "rotate"]
```

3.4.3 Time Sequenced Actions

A *Time sequenced action*, is a list of actions separated by the sequential operator `":"`, where each action is executed `for` N simulation steps (*ticks*) before moving on to the next action in the list (if the rule persists). For instance:

```
# "blink" : ["blink" "move-forward"] for 10 : "rotate" for 18 : "blink" : "move-forward"
↪  for 10 .
```

The absence of the `for` keyword implies that the action is executed only once (i.e equivalent to `for 1`). In the case that a parallel action is executed for multiple time steps, then the durative/discrete action semantics presented in Section 3.4.2 apply.

3.4.4 Wait Repeat Actions

A *wait repeat* action, is an action that is controllably repeated. The syntax is:

<Action> `wait-repeat` **<T> <Repetitions>**

If the execution of the action does not result in a change in the environment that causes a different rule to fire, will be executed *Repetition* times, with a T time units interval between tries. After that time has elapsed, TR-TURTLES will signal an user error.

For instance, in the following *wait repeat* action, when the conjunction of beliefs is true, the agent will execute `ungrasp` every 2 ticks for 10 times,

```
 # "holding" & "at-depot" --> "ungrasp" wait-repeat 2 10 .
```

Similarly, in this example, if the conjunction of the beliefs `"grasping"` and `"see-depot"` is true, then the robot agent will use the durative action `"move"`, print the message `"Moving towards a depot"` with the attached action and set the agent's color to yellow.

```
 # "grasping" & "see-depot" --> "move" ++ [[]-> show "Moving towards a depot" set color
 ↪  yellow] .
```

3.4.5 Procedure Calls

Finally, a rule RHS can be a *procedure call*. For instance the following rule calls a procedure "wander".

```
   # "holding" --> "wander" .
```

Procedures are explained in the section that follows (Section 3.5.

3.4.6 Side Effects

Side Effects are actions to be called at the end of the execution rule, and in the case of TR-Turtles, can be any anonymous NetLogo command, even adding a belief in the store, that the agent would like to assert when an action execution is completed. They can be added using the `++` operator at the end of the rule.

For example in the following rule, the *side effect* `show "moving"` (i.e. a message displayed to the NetLogo console) is executed after the `"move-forward"` action takes place.

```
# "see-can" & "can-move-ahead" --> "move-forward" ++ [[] -> show "moving" ] .
```

3.5 TR-Turtles Procedures

Procedures in TR-Turtles are collections of guarded action rules under a name (string) that can be called, as a result of a rule firing. Their role is to provide code (rules) organization. The rules of a procedure are included in the keywords `procedure <name>` and `end-procedure`, with the former declaring also the name of the procedure. For example:

```
procedure "clean-cans"
    ...
    # "holding" --> "wander" .
    ...
end-procedure

procedure "wander"
    ...
    # "true" --> ["move-forward" "rotate"] .
    ...
end-procedure
```

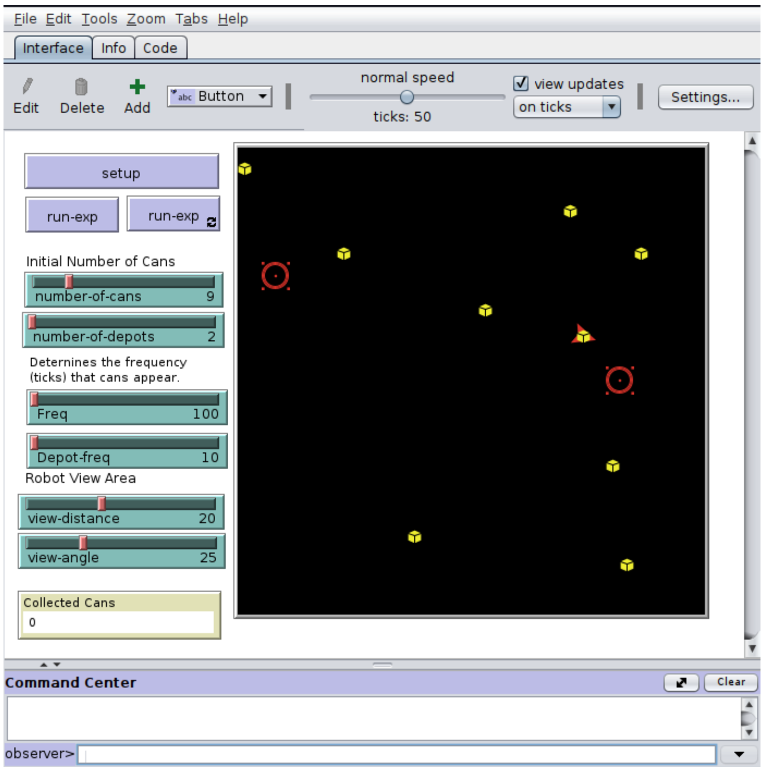
The modeler has to state the initial procedure to be activated when the agent starts its execution via the `set-goal` keyword.

It should be noted that TR (and thus TR-Turtles) procedures differ from ordinary procedures in the sense that they do not terminate or return a value: once the environment changes, rules are re-evaluated from the top-level goal to determine the next rule to fire.

A Teleo-Reactive Example

A full example is provided below, presenting a fully functional simulation file that makes use of the TR-Turtles meta-interpreter's capabilities. The following section explains every part of the code, and experienced NetLogo programmers might find some bits rather obvious. Our aim was to be as complete as possible in order to target not-so-experienced programmers.

The example presents an agent that collects cans in a room to deposit them in a number of bins (depots). In order to demonstrate the reactive features of the TR approach, depots move randomly inside the room and switch with a specified frequency between two states: those of accepting and not-accepting cans. However, this state is *invisible* to the robotic agent, i.e., the agent has no sensor to detect the state of the can, resulting in some cases in a failed action, i.e., one that does not change the environment. Robotic sensors for detecting the location of cans and depots have a limited range, implying that the robot has to engage in wandering behavior in order to find the respective positions. Thus, a robotic agent has to locate a can, pick it up, reach a depot, possibly follow it in case the depot moves away and try (multiple times) to drop the can. A snapshot of the simulation example is provided in Fig. 2.



normal speed
ticks: 50
view updates ✓
on ticks

Initial Number of Cans
number-of-cans 9
number-of-depots 2
Determines the frequency (ticks) that cans appear.
Freq 100
Depot-freq 10
Robot View Area
view-distance 20
view-angle 25
Collected Cans
0

Figure 2: A snapshot of the simulation example prototypeModel.nlogo.

In the code presented next, the TR-Turtles file is included; breeds and globals are declared, as well as the fact that robots own the `[teleor-store]` to store, update and recall the values of their perceptions.

```
__includes [ "./tr/teleoTurtles.nls" ]

breed [robots robot]
breed [cans can]
breed [depots depot]

robots-own [teleor-store]
globals [collected-cans]
```

Within the "setup" procedure, after the usual NetLogo "clear everything" and reset of the simulation ticks (which is necessary), the "setupEnv" procedure is called, 1 robot agent is created and then the "tr-code-of-robots" procedure is called that is responsible for the initialization of the Teleo-Reactive robot agent code.

```
to setup
  ca
  reset-ticks
  setupEnv
  create-robots 1 [
    set size 2
    set color red
    tr-code-of-robots
  ]
end
```

The "run-exp" top-level procedure is responsible for executing the simulation. First, it is asking the agentset of "cans" to run a set of commands "cans-code" (see below) Then the code for depots follows, which changes the state of the depots with a specified frequency and moves them slowly in space, turning left and right in a random direction at an angle of 45 degrees. The depot becomes green if the remainder of the module is smaller than the "Depot-freq" and red otherwise.

After that, "robots" are asked to run the `"execute-rules"` command that are responsible for the TR meta-interpreter execution, that is included in the "teleoTurtles.nls" file. The full behaviour including perception is encoded in the TR program, thus nothing more needs to be "asked" from the TR agents regarding their execution.

Finally, the last part controls the introduction of new cans in the environment, based on a can generation frequency and the existing number of cans. This is a process that occurs continuously.

```
to run-exp
  ask cans [cans-code]
  ask depots [
    ifelse ticks mod Depot-freq < (Depot-freq / 2) [set color green] [set color red]
    ifelse can-move? 1
       [fd 0.01]
       [rt random 45 lt random 45]
    ]
  ask robots [execute-rules]
  if (ticks mod Freq = 0 and count cans < number-of-cans) [create-cans 1 [place-can]]
  tick
end
```

The "setupEnv" procedure, which was mentioned earlier during the setup of the simulation, is responsible for setting up the environment of the simulation. The "create-depots" command creates breeds of depot agents with the specific attributes of having the shape of a depot, color green and size 2, as well as setting its coordinates to a random patch where it doesn't have any other turtles in a radius of 4 patches. The "create-cans" command creates breeds of cans agents and places them randomly in the environment.

```
to setupEnv
  create-depots number-of-depots
    [ set shape "depot"
      set color  green
      set size 2
      move-to one-of patches with [not any? turtles in-radius 4]
    ]
   create-cans number-of-cans
    [ place-can
    ]
end
```

When each "can" is created it is placed in a random spot in the environment without any other turtles within a radius of 4 patches.

```
to place-can
   set shape "box"
   set color  yellow
   move-to one-of patches with [not any? turtles in-radius 4]
end
```

The "tr-code-of-robots" procedure starts with the "tr-init" command that initializes the TR-Turtles meta-interpreter data structure. Then it defines the "belief-update-function" with an anonymous NetLogo procedure call, "[[] -> update-robot-beliefs]". All the "beliefs" that the TR-agents can have are declared. The declaration of "durative actions" and "discrete actions" is then set.

In order to understand the TR approach, the condition part of the rules are subgoals to be achieved, by lower priority rules, i.e. rules stated lower in the procedure. For instance the rule `# "touching" --> "grasp" .`, via its action `"grasp"` achieves the condition `"holding"` of the rule `# "holding" --> "wander" .`.

```
to tr-code-of-robots
  tr-init
  belief-update-function [[] -> update-robot-beliefs]
  beliefs ["holding" "at-depot" "see-depot" "see-can" "touching" "can-move-ahead"]
  durative-actions ["move-forward" "rotate"]
  discrete-actions ["ungrasp" "grasp" "blink"]
  procedure "clean-cans"
    # "holding" & "at-depot" --> "ungrasp" wait-repeat 2 10  ++ [[]-> show "At-deport -
     ↪  Delivered" set color red] .
    # "holding" & "see-depot" & "can-move-ahead" --> ["blink" "move-forward"]  .
    # "holding" --> "wander" .
```

```
     # "touching" --> "grasp" .
     # "see-can" & "can-move-ahead" --> "move-forward" .
     # "true" --> "wander" .
    end-procedure

   procedure "wander"
    # "can-move-ahead" --> "move-forward" for 2 : "rotate" for 1 .
    # "true" --> "rotate".
   end-procedure

   set-goal "clean-cans"
 end
```

The procedure "update-robot-beliefs", implements the perception of the agent. The procedure uses standard NetLogo primitives to scan the environment in the proximity of the agent for cans and depots. For instance, the first `ifelse` command simulates a sensor detecting depots in a cone, defined by `view-distance` and `view-angle` simulation parameters (sliders). In the case that a depot is "in sight" of this field of view, the agent adds the corresponding belief; in the case that it is not, the belief is removed (`no-belief`, i.e. its value is set to false, indicating no detection of a depot.

As it can be seen from the procedure, a number of such sensors can be implemented with no restriction in what can be encoded. For instance, when an agent "grasps" a can, it creates a NetLogo link with it, thus the existence of such a link determines the truth value of the belief *"holding"* in the code.

```
to update-robot-beliefs
 ifelse any? depots in-cone view-distance view-angle
   [add-belief "see-depot"]
   [no-belief "see-depot"]
 ifelse any? depots-here  [add-belief "at-depot"]  [no-belief "at-depot"]
 ifelse any? cans in-cone view-distance view-angle [add-belief "see-can"] [no-belief
 ↪  "see-can"]
 ifelse any? cans in-radius 1 [add-belief "touching"] [no-belief "touching"]
 ifelse any? my-out-links [add-belief "holding"] [no-belief "holding"]
 ifelse can-move? 0.2 [add-belief "can-move-ahead"] [no-belief "can-move-ahead"]

end
```

The procedure `"cans-code"` implements the behaviour of cans. A can to disappear (i.e. dropped in the depot) must be close to the depot and must not have any links (i.e. a connection to a robot). If both conditions apply, then the can is removed from the simulation (`"die"`).

```
to cans-code
  if any? depots in-radius 1 and not any? my-in-links
     [set collected-cans collected-cans + 1
      die
      ]
end
```

The following is a list of agent actions. Grasping a can is creating a NetLogo link with it. Thus the robot moves to the same location as a nearby can and creates a link with the can. The `tie` makes the can follow the movement of the robot.

```
;;; Crearting a link.
to grasp
  move-to one-of cans in-radius 1
  create-link-to one-of cans-here [tie]
end
```

The action "ungrasp" destroys the link between the robot and the can, however for the action to be successful the depot must be available, i.e. have a green color. If not, then the action simply does not change the environment (i.e., the can is still in the robot) and the directed link is not affected.

```
to ungrasp
  if [color = green] of one-of depots-here
   [ask my-out-links [die] ]
end
```

The action ”move-forward” will move the robot forward 0.2 patches. This is a case, that although an agent action is a simple NetLogo turtle action, it is “wrapped” inside a user defined procedure, so that it can be declared as durative or discrete (see Section 3.3).

```
to move-forward
  fd 0.2
end
```

The action ”blink” sets the color of the robot to green, and is simply in the simulation to demonstrate discrete actions.

```
to blink
  set color green
end
```

Finally, the action ”rotate” makes the robot rotate randomly by 8 degrees.

```
to rotate
  rt random 8
end
```

B TR-Turtles Grammar

This section provides the grammar of TR-Turtles in BNF form. It is important to mention that each ⟨GuardedRule⟩ must start with the condition operator `"# "` and end with the rule reporter `" ."` (including the spaces before and after). Last but not least, it should be noted that an anonymous NetLogo procedure, ⟨NetLogoProc⟩, is going to appear in the standard NetLogo syntax, i.e. `[[]->...]` in the actual code of the meta-interpreter.

The Backus–Naur form is followed to express the TR-Turtles grammar. Non terminals are enclosed between a pair of `<>` brackets, while terminals appear in double quotes. The terminal `"string"` stands for any valid NetLogo string. The Kleene `"*"` operator has the usual meaning, i.e. that a symbol sequence can be ”repeated 0 or more times”, and the same applies for the positive Kleene operator `"+"` (”repeated 1 or more times”). The operator `"?"` means that a sequence can be repeated ”0 or 1 time”. Symbol sequences separated by a vertical bar `|` indicates choice.

The TR-Turtles grammar is provided in Fig. 3.

⟨TRProgram⟩	::=	`"tr-init"` ⟨BeliefsD⟩ ⟨ActionD⟩ ⟨Procedures⟩
⟨BeliefsD⟩	::=	`"belief-update-function"` ⟨NetLogoProc⟩
		`"beliefs"` `"["` ⟨Belief⟩* `"]"`
⟨Belief⟩	::=	`"string"`
⟨ActionD⟩	::=	`"durative-actions"` `"["`⟨ActionDescr⟩*`"]"`
		`"discrete-actions"` `"["`⟨ActionDescr⟩*`"]"`
⟨Procedures⟩	::=	⟨Procedure⟩$^+$ `"set-goal"` ⟨PName⟩
⟨Procedure⟩	::=	`"procedure"` ⟨PName⟩ ⟨GuardedRule⟩* `"end-procedure"`
⟨PName⟩	::=	`"string"`
⟨GuardedRule⟩	::=	⟨Guard⟩ `"--->"` ⟨Actions⟩ (`"++"` ⟨NetLogoProc⟩)? `"."`
⟨Guard⟩	::=	⟨Belief⟩ (`"&"` ⟨Belief⟩)* | `"true"`
⟨Actions⟩	::=	⟨AnotAction⟩ | ⟨PName⟩
⟨AnotAction⟩	::=	⟨Action⟩
		| ⟨Action⟩ `"wait-repeat"` W R
		| ⟨Action⟩ `"for"` N (`":"`⟨Action⟩(`"for"` N)?)*
⟨Action⟩	::=	⟨ActionDescr⟩ | `"["` ⟨ActionDescr⟩$^+$ `"]"`
⟨ActionDescr⟩	::=	`"string"`
⟨NetLogoProc⟩	::=	*Anonymous NetLogo Procedure Call*

Figure 3: TR-Turtles grammar. Note that W, R and N are integers.

References

[Clark and Robinson, 2015] Clark, K. L. and Robinson, P. J. (2015). Robotic agent programming in teleor. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5040–5047.

[Nilsson, 1993] Nilsson, N. (1993). Teleo-reactive programs for agent control. *Journal of artificial intelligence research*, 1:139–158.

[Nilsson, 2001] Nilsson, N. J. (2001). Teleo-reactive programs and the triple-tower architecture. *Electron. Trans. Artif. Intell.*, 5(B):99–110.

[Staples et al., 1989] Staples, J., Robinson, P. J., Paterson, R. A., Hagen, R. A., Craddock, A. J., and Wallis, P. C. (1989). Qu-prolog: An extended prolog for meta level programming. In *Meta-Programming in Logic Programming*, page 435–452. MIT Press, Cambridge, MA, USA.