TSTATES Library in Netlogo

Manual

Ilias Sakellariou

November 2011, Revised Jun 2013

1 Introduction

The present document is a draft of the manual for the TSTATES DSL for NetLogo. The TSTATES allows specification of agent behaviour using states machines. This is an on-going work and the library is under heavy development. The current document concerns Version 2 of the library as described in [1] and [2].

2 Installation and Requirements

2.1 Installation

TSTATES come in the form of a nls file (*stateMachines.nls)*, i.e. a file that can be included in a netlogo file, by issuing the corresponding **`__includes`** command (please see the NetLogo manual for more information). Thus, the only "installation" requirement is to place the *stateMachines.nls* file in your local disk and include the corresponding **`__includes`** command at the top of your model.

The library is built using the standard NetLogo programing language and thus allows taking advantage of all the features of the NetLogo language as explained below.

Using the library requires **NetLogo version 5.0** and above since its implementation heavily depends on *tasks* a feature included in recent NetLogo releases.

2.2 Model Requirements

There are a few requirements for using the library in a NetLogo model:

- ALL complex agents, i.e. agents whose specification is defined by state machines, MUST have three *declared -own variables*:
  - **`active-states`**
  - **`active-states-code`**
  - **`active-machines`**
  - **`active-machine-names`**
- These are the variables to which current states and their respective code are stored. So, in your model if there is a breed of turtles which you wish to model as State-Machine agents, then you should have a **`BREED-own [active-states active-states-`**

`code active-machines]` declaration (along with any other variables that you wish to include in your model).

- MAKE SURE that when you create/setup your turtles you run once during initialization the "**`initialise-state-machine`**" library command.
- You also must have **`ticks`** (see NetLogo manual) in your model or timeout facilities described below will not function.

A more detailed discussion and an example model is provided below.

Please note that the manual assumes knowledge of the NetLogo platform.

Please report bugs to either iliass@uom.gr or iliass@uom.edu.gr

3 State Machines in NetLogo (TSTATES LIB)

A state machine in NetLogo is a collection of state definitions under a name. A state machine definition is implemented as a *NetLogo reporter* that returns a list of *state definitions*. The first state in this list is considered to be the initial state of the machine. The name of the reporter depends on the type of the state machine and always includes the prefix **`state-def-of-<state-machine-name>`**. Currently, there are two kinds of state machines that can be defined:

- *NetLogo breed* specific state machines
- State machines that can be *invoked (called)* from another state machine.

NetLogo breed specific state machines

There is only one such machine specified for each ***breed*** of turtles that exist in a simulation. The name of the reporter consists of the prefix **`state-def-of`** followed by the *name* of the breed. For instance if you have a breed collectors, then the corresponding reporter name is **`state-def-of-collectors`**. The definition of NetLogo breed specific machines is compulsory for each state based agent. These machines are initalised for each turtle (of the specific breed) by calling the **`initialise-state-machine`** turtle procedure at turtle creation/setup.

Example

Assuming that we have defined a breed called **`collectors`**, their behaviour is specified by a state machine that is defined as follows:

```
to-report state-def-of-collectors
  report (list
    state "patrol"
    … (state transitions)
    end-state
    state "move-to-base"
    … (state transitions)
    end-state
    )
end
```

During initialization/creation of the turtles of breed collectors the procedure **`initialise-state-machine`** should be called:

```
to setup-collectors
   create-collectors no-of-collectors [
      set shape "bulldozer top"
      set color green
      initialise-state-machine
      ...
      ]
end
```

State machines that can be invoked (called) from another state machine.

Their name is formed by the prefix **state-def-of-<name>** followed by the *name* of machine. The latter is used to *invoke* the specific machine through a transition as discussed later.

Example

Assume that there is the need to define a machine named **get-away-from-obstacle**, the reporter providing the specification to that machine would be:

```
to-report state-def-of-get-away-from-obstacle
   report (list
    state "finding-a-clear-space"
    ...
    end-state
    state "check-space-cleared"
    ...
    end-state
    )
end
```

3.1 State Definition

Each state definition inside a list as those shown above, is included within the keywords **state <StateName>** and **end-state**. **<StateName>** can be any NetLogo string. Each state definition consists of a number of transitions, each having the following form:

```
# on <condition> do <Action> goto <stateName>
```

OR

```
# on <condition> do <Action> activate-machine <MachineName>
```

where

- **<Condition>** is a string representation of either a NetLogo Boolean condition, or a Boolean reporter or any valid Boolean NetLogo expression,
- **<Action>** is a string representation of a NetLogo procedure(s) and
- **<StateName>** is a state name as defined above.

For instance the following is transition from the current state to the state patrol, when the agent detects that it is at the base (**at-base** boolean reporter returns true).

```
 # when "at-base" do "drop-samples" goto "patrol"
```

There are a few special conditions that allow encoding of complex behaviours

- **for-n-ticks <n>** will be true for n ticks after last entering the state (using a transition from a different state),
- **after-n-ticks <n>** will become true after n ticks the state was last entered
- **otherwise** is always true, meaning that this transition will always be triggered.

- **`invoked-from <state>, previous-active-state <state>, on-failure <MachineName>`** and **`on-success <MachineName>`** are special conditions discussed later.

TSTATES has the keyword "**`nothing`**" to indicate a transition with no action. For instance

```
# otherwise do "nothing" goto "wait-approval"
```

The standard **`goto`** command allows indicating a state to move to, however this must be a state within the same machine. Since there can exist multiple machines defined in a NetLogo file, a transition can move to (or better pass control to) another state machine, through the **`activate-machine <StateMachine>`** transition. This is the *callable states* feature supported by TSTATES discussed in the next section. For instance:

```
# when "detect-obstacle" do "nothing" activate-machine "get-away-from-obstacle"
```

Finally the third part can be either a **success** or **failure** pseudostate, as discussed later.

Condition evaluation occurs bottom-up using the order that transitions occur in the state definition. Thus, the first transition to be evaluated is the first one on that list that has its conditions satisfied.

Example

The complete definition of the collectors state machine is shown below:

```
to-report state-def-of-collectors
 report (list
  state "patrol"
  # when "detect-obstacle" do "nothing" activate-machine
                                        "get-away-from-obstacle"
   # when "detect-samples" do "pick-samples" goto "move-to-base"
   # otherwise do "move-randomly" goto "patrol"
  end-state

  state "move-to-base"
  # when "detect-obstacle" do "nothing" activate-machine
                                      "get-away-from-obstacle"
   # when "at-base" do "drop-samples" goto "patrol"
   # otherwise do "move-towards-base" goto "move-to-base"
  end-state
    )
end
```

3.2 Invoking State Machines

The library supports the concept of **callable state machines**, i.e. state machines that can be invoked by a transition from any state and terminate returning a boolean result. The concept is similar to nested functions, in the sense that when such a machine terminates, "control" returns to the state that invoked the machine. Each such callable state machine, has to include at least a **success** or a **failure** pseudostate to terminate its execution.

For example, in the following code, the last transition incluides a **success** preudostate:

```
to-report state-def-of-get-away-from-obstacle
```

```
 report (list
  state "finding-a-clear-space"
  ...
  end-state
  state "check-space-cleared"
  # when "obstacle-around" do "nothing"
                            goto "finding-a-clear-space"
   # otherwise do "nothing" success
  end-state )
end
```

Upon termination of execution, the calling state can optionally activate transitions on the result returned by the invoked machine, by employing the special **on-success <MachineName>** and **on-failure <MachineName>** transitions conditions. Machines are invoked using the **activate-machine <MachineName>** special action of the library and the implementation allows sets no limit to the number of machines that can be invoked by a single state. However, just as ordinary programming functions, nested invocations for machines can reach any level (permitted by the memory limitations of the NetLogo platform itself). For instance, the following transition invokes a machine named "**get-away-from obstacle**":

```
# when "detect-obstacle" do "nothing" activate-machine "get-away-from-obstacle"
```

As mentioned above, in order to support callable machines, the condition part is extended by a few primitives. The calling agent can include in its conditions the

- **on-success <MachineName>** , that succeeds if the machine **<MachineName>** that was invoked terminated in a success state, and
- **on-failure <MachineName>** , that succeeds if the machine invoked terminated in a failure condition.

It should be noted that before the invocation of the callable machine both these conditions evaluate to false.

The invoked machine can include in its conditions the following

- **invoked-from <state>**, which evaluates to true if the state that invoked the current state is that stated in the parameter.
- **previous-active-state <state>**, which evaluates to true if the state is active. An actiove state is a state that has not terminated and has invoked directly or indirectly the current running machine.

3.3 Executing State machine Agents.

Executing the specified machine for each turtle is done by calling in each simulation cycle the **initialise-state-machine** turtle specific primitive. Thus a "*run-experiment*" would look like the following

```
to run-experiment
  ask collectors [execute-state-machine]
  tick
end
```

3.4 TSTATES Available Procedures and Reporters

initialise-state-machine
(procedure)

The procedure "loads" the default state machine for the agent of breed B (machine state-def-of-B) and initializes the current state to be the first state on that list. It should be called once, preferably during the initialisation of the agent.

execute-state-machine
(procedure)

Executes the state code (condition evaluation → action on transition → state change) of the machine. This is a *turtle specific primitive* that should be called in each discrete step of the execution in the simulation experiment.

3.5 Skeleton of a Model Using TSTATES

This section provides a skeleton of an example model with the major points highlighted in order to simplify the use of the DSL in an existing model.

```
_includes ["stateMachines.nls"]

… (Other breeds in your experiment)
breed [collectors collector]

collectors-own [ …(other values) active-states active-states-code
  active-machines active-machine-names ]

… (Other code)

to setup-collectors
   create-collectors robots [
       ...
       initialise-state-machine
       ]
end

to run-experiment
  ask collectors [execute-state-machine]
  tick
end

to-report state-def-of-collectors
 report (list
  state "patrol"
   # when "detect-obstacle" do "nothing" activate-machine "get-away-from-obstacle"
   # when "detect-samples" do "pick-samples" goto "move-to-base"
   # otherwise do "move-randomly" goto "patrol"
  end-state
  state "move-to-base"
   # when "detect-obstacle" do "nothing" activate-machine "get-away-from-obstacle"
   # when "at-base" do "drop-samples" goto "patrol"
   # otherwise do "move-towards-base" goto "move-to-base"
  end-state)
end

to-report state-def-of-get-away-from-obstacle
 report (list
  state "finding-a-clear-space"
   ...
  state "check-space-cleared"
   # when "obstacle-around" do "nothing" goto "finding-a-clear-space"
   # otherwise do "nothing" success
  end-state )
end
```

Inclusion of nls file

Necessary breed-own variables

Initialization of State Machine

State machine "execution"

State machine Definition for breed "collectors"

State machine Definition for callable state machine

3.6 The Termites Model

The termites model provided the initial motivation for the TSTATES library. Below a comparison between the implementation of the model in TSTATES and the original implementation in the Netlogo library. Figure 1 presents the state machine of the termites model.

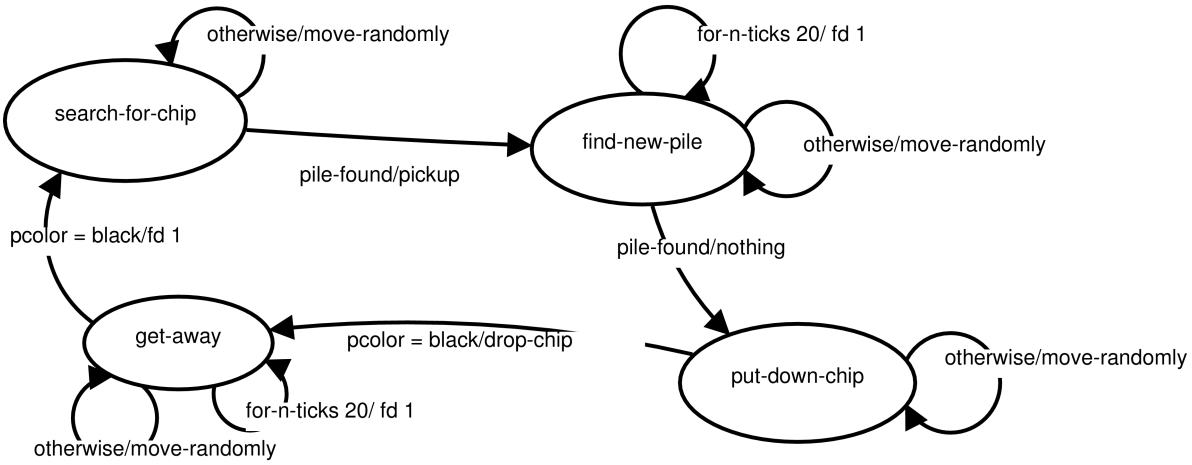

Figure 1: The State Machine of the Termites Model

TSTATE Implementation	NetLogo Lib Implementation

`to-report state-def-of-turtles`
` report (list`
`  state  "search-for-chip"`
`   # when "pile-found" do "pick-up"`
`     goto "find-new-pile"`
`   # otherwise do "move-randomly"`
`     goto "search-for-chip"`
`  end-state`
`  state "find-new-pile"`
`   # for-n-ticks 20 do "fd 1"`
`     goto "find-new-pile"`
`   # when "pile-found" do "nothing"`
`     goto "put-down-chip"`
`   # otherwise do "move-randomly"`
`     goto "find-new-pile"`
`  end-state`
`  state "put-down-chip"`
`   # when "pcolor = black" do "drop-chip"`
`     goto "get-away"`
`   # otherwise do "move-randomly"`
`     goto "put-down-chip"`
`  end-state`
`  state "get-away"`
`   # for-n-ticks 20 do "fd 1"`
`     goto "get-away"`
`   # when "pcolor = black" do "fd 1"`
`     goto "search-for-chip"`
`   # otherwise do "move-randomly"`
`     goto "get-away"`
`  end-state )`
`end`
	`to go`
	`  ask turtles`
	`    [ ifelse steps > 0`
	`        [ set steps steps - 1 ]`
	`        [ run next-task`
	`          wiggle ]`
	`      fd 1 ]`
	`  tick`
	`end`
	
	`to wiggle  ;; turtle procedure`
	`  rt random 50`
	`  lt random 50`
	`end`
	
	`;; "picks up chip" by turning orange`
	`to search-for-chip`
	`  if pcolor = yellow`
	`    [ set pcolor black`
	`      set color orange`
	`      set steps 20`
	`      set next-task task find-new-pile ]`
	`end`
	
	`;;  look for yellow patches`
	`to find-new-pile`
	`  if pcolor = yellow`
	`    [ set next-task task put-down-chip ]`
	`end`
	
	`;; finds empty spot & drops chip`
	`to put-down-chip`
	`  if pcolor = black`
	`   [set pcolor yellow`
	`    set color white`
	`    set steps 20`
	`    set next-task task get-away ]`
	`end`
	`;; get out of yellow pile`
	`to get-away`
	`  if pcolor = black`
	`   [ set next-task task search-for-chip ]`
	`end`

4 Grammar

What follows if the BNF definition of the TSTATES library DSL.

```
    Machine  = "state-def-of-" MachineName "report (list" State+ ")"
      State  = "state" StateName Transition+ "end-state"
 Transition  = "#" Condition "do" Action StateChange
  Condition  = "when" ReporterExp | "otherwise"
               |"for-n-ticks" N | "after-n-ticks" N
               |"invoked-from" StateName |"previous-active-state" StateName
               | "on-failure" MachineName | "on-success" MachineName
     Action  = Procedures | "nothing"
 StateChange = "goto" StateName | "activate-machine" MachineName
               |"success" | "failure"
          N  = <INTEGER>
  StateName  = <STRING>
MachineName  = <STRING>
 Procedures  = <STRING>
ReporterExp  = <STRING>
```

5 Obtaining TSTATES

This manual, examples, and the DSL can be obtained by the following site:

http://users.uom.gr/~iliass (follow the link “Turtles as State Machines”)

6 References

[1] Sakellariou, I. (2012) Agent Based Modelling and Simulation using State Machines, SIMULTECH 2012, Rome Italy.

[2] Sakellariou, I. (2012). Turtles as state machines - agent programming in netlogo using state machines. ICAART 2012 February, Algarve, Portugal.

Please report any bugs to iliass@uom.gr

Have fun with NetLogo.