X-Machines Simulation In NetLogo: Turtles as X-Machines

Ilias Sakellariou

January 9, 2015

Abstract

This document describes the TXStates Domain Specific Language that allows specifying and executing X-Machine specification in NetLogo. The document acts as a manual and a presentation for TXStates in order to allow people to understand and use TXStates as toll for developing complex agents. The current document assumes basic knowledge of the NetLogo platform.

1 Introduction

The main idea behind this work is to allow agent behaviour to be specified as an X-machine. For completeness, an X-Machine is defined as:

Definition 1. *A stream X-machine [Holcombe and Ipate, 1998] is an 8-tuple*

$$\mathcal{X} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$$

where:

- Σ *and* Γ *are the input and output alphabets, respectively.*
- Q *is the finite set of states.*
- M *is the (possibly) infinite set called memory.*
- Φ *is a set of partial functions* φ*; each such function maps an input and a memory value to an output and a possibly different memory value,* $\varphi : \Sigma \times M \rightarrow \Gamma \times M$.
- F *is the next state partial function,* $F : Q \times \Phi \rightarrow Q$*, which given a state and a function from the type* Φ *determines the next state.* F *is often described as a state transition diagram.*
- q_0 *and* m_0 *are the initial state and initial memory respectively.*

Definition 2. *A computation state is defined as the tuple (q, m), with $q \in Q$ and $m \in M$. The computation step is defined as $(q, m) \overset{\varphi}{\vdash} (q', m')$ with $q, q' \in Q$ and $m, m' \in M$ such that $\varphi(\sigma, m) = (\gamma, m')$ and $F(q, \varphi) = q'$. The computation is the series of computation steps when all inputs are applied to the initial computation state (q_0, m_0).*

2 Rational

Informally, an agent is an entity that maps its current percepts and state to an action. Thus, in order to model the behaviour of an agent using X-Machines, a mapping of the concepts of the former to the latter is necessary. However, due to the structure of the X-Machines this mapping is clear and straightforward:

- Agent percepts form the input alphabet Σ, and are updated in each simulation cycle.
- Agents hold their simulation state (different that the X-Machine State) and all parameters that affect their behaviour in memory M. For instance, agent gender and speed, can be modelled as elements of memory M.
- Agent behaviour is modelled as a set of functions Φ, and obviously the transition diagram F.
- Finally, agent actions are mapped to the output Γ. A delicate issue regarding modelling appears here. Since, according to the X-Machine model, output *cannot* change the memory, actions should be implemented carefully, so that they do to change the X-Machine state of the calling agent, but *only* the simulation environment.

The TXStates provides support for easily encoding all the above in NetLogo. Briefly:

- Memory M is a *NetLogo table* and is stored in a *turtle-own variable* as described in section 3.1.
- The set of states Q and the transition digram F are encoded using the primitives of the TXStates DSL as described in section 3.5.
- Functions of the set Φ are encoded as NetLogo reporters, that return results in a specific format, the latter being processed by the TXStates interpreter.
- the output Γ contains an action represented as *NetLogo tasks* and contains the (set of) NetLogo procedures, applied to the simulation environment by the interpreter. The DSL supports encoding such tasks as part of the X-Machine function specification.

- Finally, input is provided by the simulation environment, through an appropriate turtle variable. The DSL provides appropriate primitives for encoding input management.

The rest of this document describes the DSL. In the following terms agent and turtle refer to the same entity. We will use the term turtle, when describing issues that are more implementation oriented (reporters, functions etc.).

3 TXStates Domain Specific Language

3.1 Turtle Variables

Each agent (turtle in NetLogo) must carry its own state and memory information. To do so, TXStates requires a number of agent own (turtle-own) variables to be defined in the NetLogo model. These can be categorised in two classes. The first class consists of the variables that the library uses *internally* and should not be changed in any way by the developer of the model, sine they hold information regarding state code, state transitions, etc. These are:

active-states active-states-code active-machines active-machine-names

The second class consists of those variables that the model under development should update/change in each execution cycle:

- *memory*: A variable holding the memory of the X-machine. Usually, the memory consists of attribute - value pairs and the DSL provides special care for its management to facilitate model development.
- *percept*: A variable that holds the percepts of the agent and is updated in each execution cycle by the environment. Appropriately encoding percepts and linking the latter to the model under development is the responsibility of the model developer. However, the DSL provides a set of primitives to allow the user clearly define percepts.
- *emotion*: A variable that holds emotion values, if the latter are specified in the model.

Thus, for specifying the behaviour of an agent using X-Machines the corresponding turtle in NetLogo should define the above mentioned variables. For instance, if we are to create turtles of the breed "persons" driven by an X-machine, the NetLogo code is:

```
persons-own [active-states active-states-code active-machines
             active-machine-names memory percept emotion]
```

3.2 X-Machine Memory Management

As mentioned the `memory` turtle variable holds the X-machine memory structure. During agent creation, the variable must be initialised and appropriate memory positions (attributes/variables) must be created. These are achieved by the following DSL commands:

- `x-init-memory`: Initialises memory to and empty structure, and is required to be invoked only once.
- `x-mem-initial-var <varName> <Value>`, Adds a new X-machine variable value pair (tuple) to the memory. The first argument `<varName>` is a string representing the name of the memory position and the second argument (`<Value>`) is its initial value. The latter can be any valid NetLogo value (integer, float, string, etc) or valid expression/NetLogo reporter. If `<varName>` has been added before, its value is replaced by the new value appearing in the command.

For instance, if it is desired to initialise X-Machine memory and create variable value pairs, with the variables "leader" set to false, "turns" set to 0 and "speed" being a random value from the list (2,3 4,5), the following code should be included in the code creating the agent:

```
...
init-x-memory
x-mem-initial-var "leader" false
x-mem-initial-var "turns" 0
x-mem-initial-var "speed" one-of [2 3 4 5]
...
```

Accessing memory variables is achieved by calling the function (NetLogo reporter) `x-mem-value <varName>` anywhere inside the code of the agent. For instance a call `x-mem-value "turns"` anywhere after its initialisation through `x-mem-intitial-val` will return the value 0.

Updating memory variables can only occur from an X-function (see subsection 3.4) and thus during X-Machine execution. The DSL provides the command `x-mem-set <varName> <Value>`, for such destructive updates. For instance, the following code updates variable "leader" to true and increases the value of "turns" by one.

```
...
   x-mem-set "leader" true
   x-mem-set "turns" x-mem-value "turns" + 1
...
```

It is stressed once more that the `x-mem-set` command can appear only as a part of an X-Machine function, since according to the model, only X functions can change memory values. If the user uses `x-mem-set` in any other place than as a "return" value of an X-function, the results will not be as expected.

3.3 Input

Encoding the X-Machine input, i.e. agent percepts is facilitated by a set of primitives. Each input is modelled as an attribute-value pair (tuple, `<P> <Val>` in the following), and there can be multiple values for an attribute, i.e. multiple tuples. The list of primitives is the following:

- `x-add-percept <P>`, adds a percept `<P>` with no value. The percept in encoded as a tuple carrying the dummy value `x-nil`.
- `x-percept-add-value <P> <Val>`, adds a percept `<P>` with value `<Val>`.
- `x-has-percept? <P>`, returns true if there is a percept `<P>` in in input.
- `x-percept-value <P>`, returns the value `<Val>` of the percept `<P>`. In the case of multiple tuples for the same percept, the first value is returned.
- `x-oneof-percept-value <P>`, returns a random value `<Val>` of the percept `<P>`, in the case that there are multiple tuples for `<P>`.
- `x-all-percept-values <P>`, returns all values of `<P>` in a list.

3.4 X-Functions

X-functions are encoded as NetLogo reporters (NetLogo jargon for functions). There are no arguments in such functions and consequently to the corresponding NetLogo reporters, since by X-Machine definition, functions operate on input and memory and produce output and memory updates. Thus, it is assumed that each function always has access to the former and produces output to the latter. X-functions must return (report in NetLogo terms) either a *success token* followed by output and memory updates or a special *failure token.* These return values will be used by the TXStates meta-interpreter to determine possible transitions, according to the implemented semantics of the DSL and X-Machines.

Thus, each such NetLogo reporter should return either:

- `x-false`, a keyword handled by the meta-interpreter, indicating that the function is not applicable,
- `x-true <xmOutput> <xmMemUpdates>`, indicating an applicable function that will produce `<xmOutput>` output and change memory according to the `<xmMemUpdates>`.

Other values return will produce an error or unexpected behaviour.

The first "argument" of `x-true` is a list of actions that the agent has to perform. These actions must correspond to NetLogo procedures, i.e. represent the effects of the execution to the environment. Each such action is a *NetLogo task* annotated by the keyword `x-action`, that gets to be executed if the function is selected by the interpreter. Delimiters `#<` and `>#` mark the start and the end of the list of actions.

Thus `<xmOutput>` has the form:

```
to-report reachExitSafa
  ifelse has-percept-type "exitReached"
   [report x-true
    #< x-action task [fd 1] >#
    #< x-mem-set "turns" (x-mem-value "turns" + x-mem-value "halfCycle")
       x-mem-set "halfCycle" 0 >#]
    [report x-false ]
end
```

Figure 1: Example of a X Function

```
 #< x-action task [...]
    x-action task [...] #>
```

The second "argument" is a list of memory updates, i.e. invocation of `x-mem-set` command described in 3.2, again delimited by `#<` and `>#`. Thus, `<xmMemUpdates>` has the following form:

```
 #< x-mem-set ...
    x-mem-set ... #>
```

Empty `<xmOutput>` and `<xmMemUpdates>` are denoted as `#< >#`. It should be mentioned that the above are *lists*, and not sets, i.e. the changes described either as environment effects or memory updates will be performed in the order they appear.

There are no limitations regarding the code that a X-function can include, as long a it respects the X-Machine simulation semantics. Special care should be taken so that these functions do not contain side-effects in any other place than the `<xmOutput>` part of the function. The meta-interpreter evaluates (runs-tries-executes) all functions, producing possible memory and output results and then decides which function to apply, the presence of side-effects (for instance changes in the simulation environment) would produce unexpected behaviour.

A example X-function encoded as a NetLogo reporter is given in figure 1. The function checks whether the agent has certain percepts, executes the NetLogo procedure `fd 1` and updates the memory variables "turns" and "halfCycle". Note that since memory updates are a list, the first x-mem-set will use the previous value of "halfCycle" and then set it to 0.

3.5 State and Transition Diagram Specification

Probably the most important aspect of the TXStates DSL is the ease by which states and transitions between states are encoded, since it allows directly encoding X-Machines in NetLogo as described in this section.

Information regarding a single state and the related transitions is encoded as:

```
state <StateName>
```

```
  # x-func <XMachineFunction 1>  goto <StateName 1>
  ...
  # x-func <XMachineFunction n>  goto <StateName n>
end-state
```

where `<XMachineFunction>` is a NetLogo X Function (reporter) of the special "type" reported in 3.4 and `<StateName>` is the name of the target state of the transition (a simple string). There is a special transition function `otherwise` which always results to adopting the corresponding transition, however, a similar effect could be achieved by an X Function with empty guard conditions.

An X-machine that consists (as usual) of multiple states is in fact a NetLogo list of such state definitions:

```
x-diagram
 state <StateNameA>
  # x-func <XMachineFunction A1>  goto <StateName A1>
  ...
  # x-func <XMachineFunction An>  goto <StateName An>
 end-state

 state <StateNameK>
  # x-func <XMachineFunction K1>  goto <StateName K1>
  ...
  # x-func <XMachineFunction Kn>  goto <StateName Kn>
 end-state
end-diagram
```

In such a specification, the first state that appears in the list is considered to be the *initial state* q_0.

In order to relate an X-Machine definition to a specific breed of turtles to the execution environment, the list of state definitions given above, is placed inside a NetLogo reporter the name of which is formed by appending the string "state-def-of-" to the breed name of the turtles. For instance, if the breed is called "persons" the X-Machine controlling the behaviour of persons will be given be a reporter named"state-def-of-persons". Figure 2 presents a part of an X- Machine encoded in the TXStates DSL.

Figure 3 presents the grammar for specifying transitions in TXStates.

3.6 X-Machine Execution

Executing the agent specifications presented in the previous section is the responsibility of the TXStates *meta-interpreter*. The latter is invoked by calling the `execute-state-machines` command, usually in each simulation cycle. Before invocation, the user must ensure that the agent percepts been updated, through appropriate calls of the corresponding primitives in 3.3.

The *meta-interpreter* is responsible for handling state transitions and action execution and implements the computation described in definition 2, with each invocation of the `execute-state-machines` command corresponding to a single *computation step* of Definition 2. Thus at each cycle, the meta-interpreter:

```
to-report state-def-of-persons
  x-diagram
    state "Entering"
    # x-func "follower?" goto "MoveToLeader"
    # x-func "moveInsideTurn" goto "Turning"
    # x-func "moveInsideEntrance" goto "Entering"
    # x-func "insideCorridor" goto "Walking"
    # otherwise do "nothing" goto "Entering"
    end-state

    state "Walking"
    # x-func "reachExitSafa" goto "AtExit"
    # x-func "reachExitMargah" goto "AtExit"
    # x-func "leaderFar" goto "MoveToLeader"
    # x-func "leaderFarTurn" goto "MoveToLeader"
    # x-func "followerFarMove" goto "WaitingForGroup"
    # x-func "followersFar" goto "WaitingForGroup"
    # x-func "moveToExit" goto "Walking"
    # otherwise do "nothing" goto "Walking"
    end-state

   (... more states)

    state "AtExit"
    # x-func "walksDone" goto "Exiting"
    # x-func "MoveInsideTurn" goto "Turning"
    # x-func "MoveInsideExit" goto "AtExit"
    # otherwise do "nothing" goto "AtExit"
    end-state


   state "Exiting"
   # x-func "reachedEnd" goto "Exiting"
   # x-func "leavingCorridor" goto "Exiting"
   end-state
 end-diagram
end
```

Figure 2: TXStates State Transition Diagram Definition

$$\begin{array}{rcl}
\textit{Machine} & \rightarrow & \textbf{state-def-of-}\ \textit{TurtleBreed}\ \textbf{report}\ \textit{X-Diagram} \\
\textit{X-Diagram} & \rightarrow & \textbf{x-diagram}\ \textit{State+}\ \textbf{end-diagram} \\
\textit{State} & \rightarrow & \textbf{state}\ \textit{StateName}\ \textit{Transition+}\ \textbf{end-state} \\
\textit{Transition} & \rightarrow & \textbf{\# x-func}\ \textit{X-Function}\ \textbf{goto}\ \textit{StateName} \\
\text{StateName} & \rightarrow & \langle\ \text{STRING}\rangle \\
\text{X-Function} & \rightarrow & \langle\ \text{STRING}\rangle \mid \langle\ \text{REPORTER-TASK}\rangle
\end{array}$$

Figure 3: TXStates DSL grammar for specifying transitions. Please note that an *X-Function* is a string representation or a Netlogo reporter, or a NetLogo reporter task, that returns special values.

1. Forms the list of functions Φ_{state}, that guard transitions in the current SXM state q, i.e. $\Phi_{state} = \phi \in \Phi : (q, \phi, q'') \in F$, in the order they appear in the agent specification.
2. Form the list Φ_{trig} that contains all functions from Φ_{state} whose guards are satisfied.
3. Select the *first* function ϕ_i from the trigger list Φ_{trig} .
4. Execute actions specified by ϕ_i.
5. Apply memory updates specified by ϕ_i.
6. Perform a transition to state q' that corresponds to function $(q, \phi_i, q') \in F$.

In order to simplify the encoding of guards, an ordering is imposed to the function application; currently the selection function chooses the *first function* in the state definition that triggers in step 3. This imposes a *priority ordering* on the transitions in a state, with the transitions that appear higher in the state definition having a larger priority.

The TXStates DSL is provided as a NetLogo library that users can include in their models and specify behaviour. The major advantage of using TXStates is that model developers can develop models in an iterative fashion, modifying the X-Machine model quite easily and viewing directly the results of their changes. Thus, complex model development can be greatly facilitated.

4 Conclusions

The TXStates is a domain specific language that allows easy encoding and execution of X Machine specifications for agents in the NetLogo environment. Although, the current version is stable, it could benefit from a number of extensions, such as percept management, and debugging.

References

[Holcombe and Ipate, 1998] Holcombe, M. and Ipate, F. (1998). *Correct Systems: Building a Business Process Solution*. Springer, London.