sxl: NetLogo state machine engine (sxl-stn)

brief

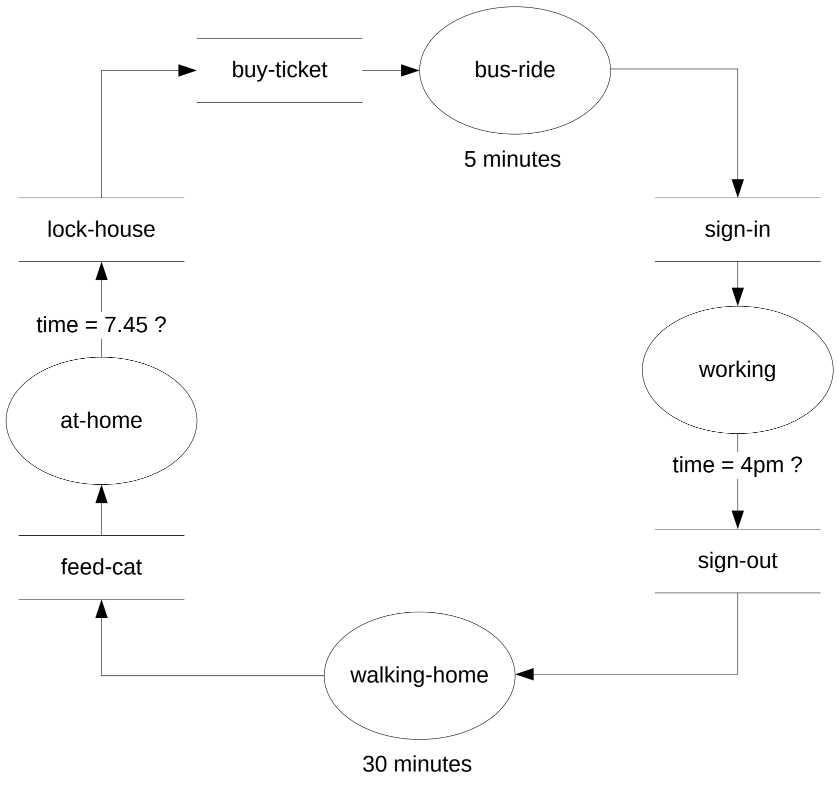
This document provides an overview of the sxl state machine evaluation engine provided for NetLogo. Finite State Machines (FSMs), State Transition Networks (STNs), etc. are not described here, check the wider computing literature for details & examples.

outline

For this documentation we use a simple STN example. This example describes a "working day". It contains 4 states {at-home, in-bus-to-work, working, walking-home} which are simply connected in 1-to-1 links, ie: the transition of one state connects only to one other state (note that sxl-stn also allows 1-to-many, many-to-many, etc). Narratively, for some agent "Bill", the example STN describes the following...

- Bill leaves home at 7.45am & locks his house.
- He buys a bus ticket & takes the bus for 5 minutes then gets off the bus.
- He signs-in & starts work.
- At 4pm Bill signs-out of work and spends 30 minutes walking home.
- Reaching home, he feeds his cat.

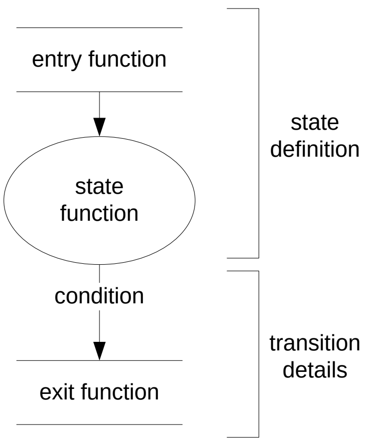
This STN can also be represented graphically...



STN features

sxl-stn allows STN states & transitions to represent the following...

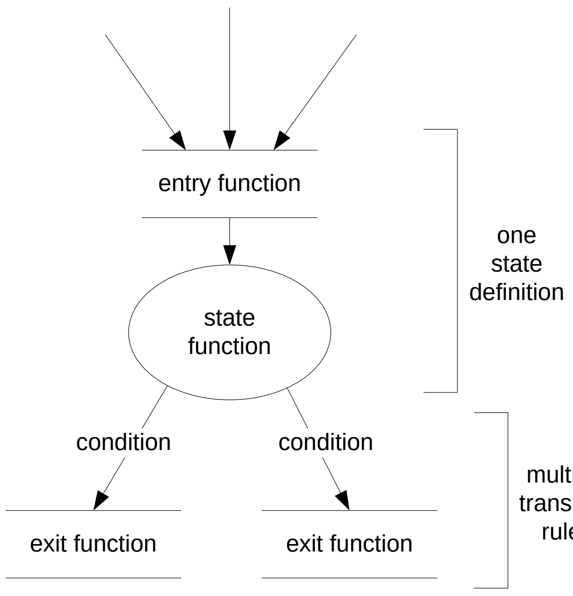
- state functions – runs each tick while an agent is in the given state;
- entry functions – runs once when an agent changes to a new state;
- exit functions – runs once when an agent leaves a state by a particular transition rule;
- conditions – used to determine if a transition takes place.



In practice states can have multiple entry routes (connections from many other states) and multiple transitions (connections to many other states) but only one entry function and one state function.

scl-stn packages up (i) the entry function and the state function as part of the state definition (ii) conditions and exit functions as part of transition rules. Transition rules also contain the name of the new state the rule transitions to.

Each state may have only one state definition but may have many transition rules.



defstate

The state definition for "working" state for Bill (first diagram) is defined using defstate which has the following syntax...

```
defstate  state-name  entry-function  state-function
```

...so, assuming procedures for sign-in and do-work-stuff are defined, Bill's working state would be something like...

```
defstate "working"  (task sign-in)  (task do-work-stuff)
```

Alternatively (if you do not like the task syntax) the functions can be named instead of being wrapped as tasks (but note that models defined like this may not run as quickly)...

```
defstate "working"  "sign-in"  "do-work-stuff"
```

initialisation

sxl-stn is a NetLogo include file. To use it (i) put the relevant __includes statement at the start of your model code (ii) before running your STN initialise it with `initialise-stn`

```
__includes [ "sxl-stn.nls" ]

to setup
  initialise-stn

  ; now do defstate's and transitions's
  defstate ....
  transition ....
end
```

transition

Transition rules are needed for each connection in the STN. The syntax for transition rules is...

```
       transition  from-state  to-state  condition  exit-function
```

Assuming sign-out is defined and there is a suitable daytime variable, Bill's transition rule from working to walking could be...

```
transition "working" "walking" (task [daytime = 1600]) (task sign-out)
```

As before (task sign-out) could be replaced with "sign-out", similarly for "`daytime = 1600`".

null-task

To specify a null task (as an entry, exit or state function) use null-task...

```
; nothing happens while sleeping...

defstate "sleeping"  (task go-to-bed)  null-task
```

always

To ensure that a state transition always occurs (ie: its condition is always true) use always...

```
  transition  "something-brief"  "next"  always  null-task
```

timing transitions

scl-stn provides primitives for timing state transitions. If you want a state to cycle for a given amount of time (a number of ticks) then transit on time out, you need to set a clock condition. Eg: specifying a 30 tick walk home...

```
  transition "walking"  "at-home"  (task [clock= 30])  null-task
```

Note that if you can use clock= in your own procedures...

```
to setup-stn
  initialise-stn
  transition "walking"  "at-home"  (task end-walking?)  null-task
end

to-report end-walking?
  report (clock= 30) or some-other-test
end
```

forcing state changes

To cleanly force a state change to some new state use *state-change*. This changes the state and runs the appropriate entry function. It does not run an exit function.

```
  state-change "new-state"
```

Note: turtles own a variable $state but it is better not to manipulate this directly.

probabilistic transitions

The simplest way to organise a transition based on probability values is as shown in the example below. This assumes two things...

1. prob% is some variable (global or breed owned) which holds a suitable probability value;
2. trigger is defined as in sxl utils...

```
;; from sxl-utils
to-report trigger [#prob]
  report (random 100) < #prob
end
```

probabilistic state transition example...

```
  transition "playing"  "eating"   (task [trigger prob%])  null-task
```

Bill's working day STN rules

A complete set of rules for Bill's working day is as follows...

```
to setup-stn
  initialise-stn

  defstate "at-home"   (task feed-cat)        (task do-home-stuff)
  defstate "at-work"   (task sign-in)         (task do-work-stuff)
  defstate "walking"   null-task              (task [forward 1])
  defstate "on-bus"    (task [buy-ticket])    null-task

  transition "at-home"  "on-bus"   (task [daytime = 7.45])
                                   (task [set house "locked"])

  transition "working"  "walking"  (task [daytime = 16.00])  (task sign-out)
  transition "on-bus"   "at-work"  (task [clock= 5])         null-task
  transition "walking"  "at-home"  (task [clock= 30])        null-task
end
```

Running an STN

STNs are run from the turtle context using run-stn, typically within some go procedure, eg...

```
to go
  ask turtles [ run-stn ]
  tick
end
```

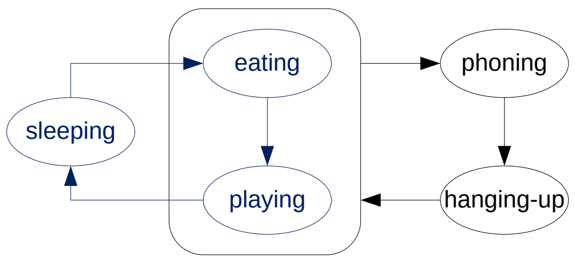
Transitions happen automatically and clock time advances automatically.

Multiple Transition Networks

There are two ways to handle multiple transition networks (i) using interrupts and (ii) using additional state variables.

using interrupts

Sometimes it is useful to have two or more STNs that are interconnected. Consider for example a simple `{eating -> playing -> sleeping ->...}` cycle. Then add in the behaviour that if the phone rings during eating or playing time, the phone will temporarily interrupt eating/playing while a new STN `{-> phoning -> hanging-up ->}` runs (where phoning is talking on the phone & hanging-up is some activity like saying goodbye, putting the phone down, noting who had called, etc. (see diagram).



The {eating -> playing -> sleeping ->...} cycle is specified as usual…

```
defstate "eating"   (task [set color red])    (task [right 45])
defstate "sleeping" (task [set color green])  null-task
defstate "playing"  (task [set color blue])   (task play)

transition "eating"   "sleeping" (task [clock= 10])  null-task
transition "sleeping" "playing"  (task [clock= 15])  null-task
transition "playing"  "eating"   (task [clock= 20])  null-task
```

…and we use start-interrupt to interrupt the current STN and save its state until later…

```
transition  "eating"   "phoning"  (task phone-rings?)  start-interrupt
transition  "playing"  "phoning"  (task phone-rings?)  start-interrupt
```

The specification of phoning & hanging-up are as usual…

```
defstate "phoning"      (task [set shape "circle"])    null-task
defstate "hanging-up"   (task [set shape "x"])         null-task

transition  "phoning"  "hanging-up"  (task [clock= 10])  null-task
```

…but when hanging-up finishes & we wish to return to whatever was happening before the earlier STN was interrupted we transit briefly to a *stop-interrupt* state restores earlier activity…

```
transition "hanging-up"  "stop-interrupt" (task [clock=  5])
                                          (task [set shape "default"])
```

Note that it is possible to have multi-level interrupts (ie: interrupts that interrupt interrupts, etc) – they are stacked and unstacked as entered/exited.

additional state variables

Using the example {eating -> playing -> sleeping ->...} cycle (above) we use a modified version of the phoning cycle {-> phoning -> hanging-up ->} which uses a system defined idle-state (which does nothing). We specify these as two independent STNs as follows...

```
; { eating -> sleeping -> playing } cycle
defstate "eating"   (task [set color red])    (task [right 45])
defstate "sleeping" (task [set color green])  null-task
defstate "playing"  (task [set color blue])   (task play)

transition "eating"   "sleeping" (task [clock=  5])  null-task
transition "sleeping" "playing"  (task [clock= 10])  null-task
transition "playing"  "eating"   (task [clock= 15])  null-task


; { idle -> phoning -> hanging-up -> idle } cycle
defstate "phoning"      (task [set shape "circle"])    null-task
defstate "hanging-up"   (task [set shape "x"])         null-task

transition "idle-state"   "phoning"     (task phone-rings?)  null-task
transition "phoning"      "hanging-up"  (task [clock= 10])   null-task
transition "hanging-up"   "idle-state"  (task [clock= 15])
                                        (task [set shape "default"])
```

To run these STNs so the phoning cycle can take precedence over the playing cycle we allocate turtles a new variable for the phoning cycle (in this example we use a variable called "stn2") and assign it to a new state value (normally during setup). NB: if you need to run the entry function use state-change-with after make-state.

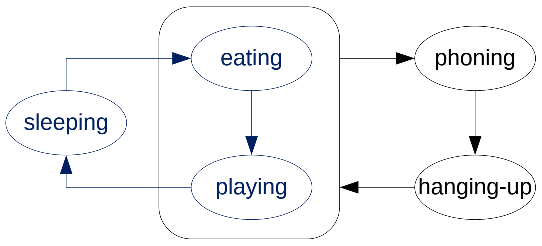
```
turtles-own [ stn2 ]      ;; turtles own an stn for the phoning cycle

to setup
  ... do other setup operations then...

  ask turtles
  [ set stn2 (make-state "idle-state")    ;; assign the stn variable
  ]

  ... finalise setup...
end
```

Finally to run the two STNs *sympathetically* while still giving priority to the phoning cycle we use run-state-with to check and update the new STN variable...

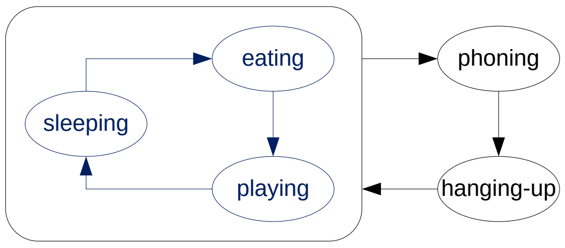
```
to go
  ask turtles
  [ set stn2 (run-stn-with stn2)
    if ($state-of stn2) = "idle"
    [ run-stn ]
  ]
end
```

Some important notes:

1. $state-of reports the state label of a state variable;
2. with this approach (using multiple state variables) you cannot selectively overlap another STN, in this case (unlike the example using interrupts) the phone-cycle takes control even when sleeping. Modifying this behaviour is possible but requires code to be surrounded by "if" conditions (or something similar).



using interrupts



using multiple state variables

Other State Transitions

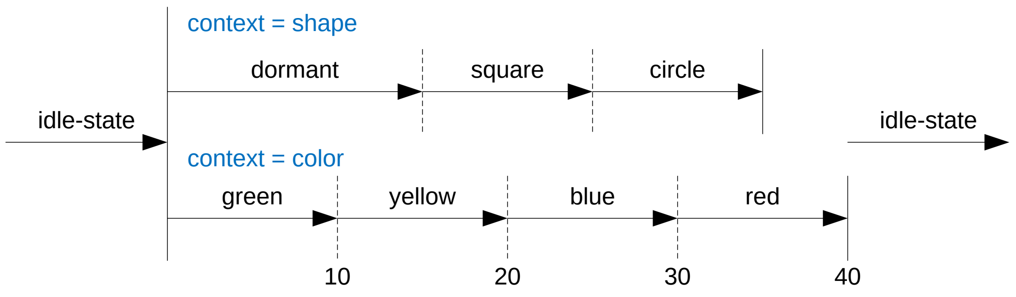
agent death

This example uses the `{eating -> playing -> sleeping ->...}` cycle above but (for some unknown reason) sleeping agents die if the phone rings. For this we specify a transition from "sleeping" to "agent-death" to fire when the phone rings. "agent-death" is an internally defined state.

```
to setup-stn-dying
  initialise-stn
  defstate "eating"    (task [set color red])      (task [right 45])
  defstate "sleeping"  (task [set color green])    null-task
  defstate "playing"   (task [set color blue])     (task play)

  transition "eating"   "sleeping" (task [clock=  5])  null-task
  transition "sleeping" "playing"  (task [clock= 10])  null-task
  transition "playing"  "eating"   (task [clock=  5])  null-task

  transition "sleeping" "agent-death" (task phone-rings?)  (task [set shape "x"])
end
```

coding timing diagrams

It is often useful (as in some of the earlier examples) to have timings on states/transitions. In some cases it is useful to simultaneously operate over multiple time lines. As an example consider the diagram below where agents change shape and color.



In this example we have two independent time lines with a named context for each. All transitions occur on clock-elapsed events with the exception of the move from the idle-state onto the time line – this happens when the phone rings.

The STN setup is fairly standard but note the use of the context variable to control transitions from the idle-state onto each of the shape & color timelines.

```
to setup-stn-timed
  initialise-stn

  ; context = shape
  defstate "dormant"  null-task  null-task
  defstate "square"   (task [set shape "square"])  null-task
  defstate "circle"   (task [set shape "circle"])  null-task

  transition "idle-state" "dormant"
               (task [context = "shape" and bell-rings?])  null-task
  transition "dormant"  "square"      (task [clock= 15])   null-task
  transition "square"   "circle"      (task [clock= 10])   null-task
  transition "circle"   "idle-state"  (task [clock= 10])
                                      (task [set shape "default"])

  ; context = color
  defstate "green"  (task [set color green])   null-task
  defstate "yellow" (task [set color yellow])  null-task
  defstate "blue"   (task [set color blue])    null-task
  defstate "red"    (task [set color red])     null-task

  transition "idle-state" "green"
               (task [context = "color" and bell-rings?])  null-task
  transition "green"    "yellow"      (task [clock= 10])   null-task
  transition "yellow"   "blue"        (task [clock= 10])   null-task
  transition "blue"     "red"         (task [clock= 10])   null-task
  transition "red"      "idle-state"  (task [clock= 10])
                                      (task [set color grey])
end
```

To set everything up we use two named STNs & start them in an "idle-state"...

```
turtles-own [ shape-stn color-stn ]

to setup
  clear-all
  setup-stn-timed

  create-turtles no.turtles
  [ set shape-stn (make-state "idle-state")    ;; assign the shape-stn
    set color-stn (make-state "idle-state")    ;; assign the color-stn
  ]
  reset-ticks
end
```

Running the timelines (the STNs) requires some checking of context...

```
to go
  ask turtles
  [ set context "shape"
    run-stn-with "shape-stn"
    set context "color"
    run-stn-with "color-stn"
  ]
end
```

time periods & non-deterministic time transitions

In addition to using fixed time transitions with clock= it is possible to specify that transitions occur across a given time period, ie: between start and end times. This mechanism ensures that a transition is equally likely to occur at any clock period between start & end times.

These types of transition are defined using *period*, see following example…

```
to setup-stn-v6
  ;
  ; states go: {eating -> sleeping -> playing ->... }
  ;
  initialise-stn
  defstate "red"     (task [set color red])      null-task
  defstate "green"   (task [set color green])    null-task
  defstate "yellow"  (task [set color yellow])   null-task
  defstate "blue"    (task [set color blue])     null-task

  transition "red"     "green"   (task [period 5  20])  null-task
  transition "red"     "yellow"  (task [period 10 25])  null-task
  transition "green"   "blue"    (task [period 0  10])  null-task
  transition "yellow"  "blue"    (task [period 0  20])  null-task
  transition "blue"    "red"     (task [period 0  10])  null-task
end
```

summary

declarations

defstate

define a new state

use: `defstate state-name entry-function state-function`

transition

define a state transition

use: `transition from-state to-state condition exit-function`

common tasks

null-task

do nothing

use: `null-task`

interrupts

start-interrupt

initiates an interrupting STN

use: `start-interrupt`

stop-interrupt

this is a pre-defined state

use: `transition from-state "stop-interrupt" condition exit-fn`

running STNs

initialisation

initialise all transition networks

use: `initialise-stn`

run-stn

run default STN in an agent context

use: `ask turtles [run-stn]`

run-stn-with

run a specific STN in an agent context

run default STN in an agent context

use: `ask turtles [run-stn-with` *`stn-variable`*`]`

changing states

change-state

force a state change & run its state entry function
use: `change-state` *`new-state-name`*

change-state-with

force a state change with a specified STN & run its state entry function
use: `change-state-with stn new-state-name`

make-state

make a new state
use: `set stn (make-state “some-state”)`

always

this is a condition used within transitions that always reports true

clock handling

clock=

a condition used to check a clock time for transitions
use:
`transition from-state to-state (task [clock= n]) exit-fn`

period

a non-deterministic time-out condition. Period specifies the earliest time & latest time for a transition. A *flat* probabilistic profile operates between these times – ie: all clock times have an equal probability of causing a transition.

use: `transition from-state to-state (task [period from to]) exit-fn`

predefined states

"agent-death"

causes the agent to die on the next cycle

"idle-state"

a predefined state with no state function or entry function. You may define transitions from and to “idle-state” but should not defstate “idle-state”.

"stop-interrupt"

causes the agent to return from an interrupt in the next cycle.