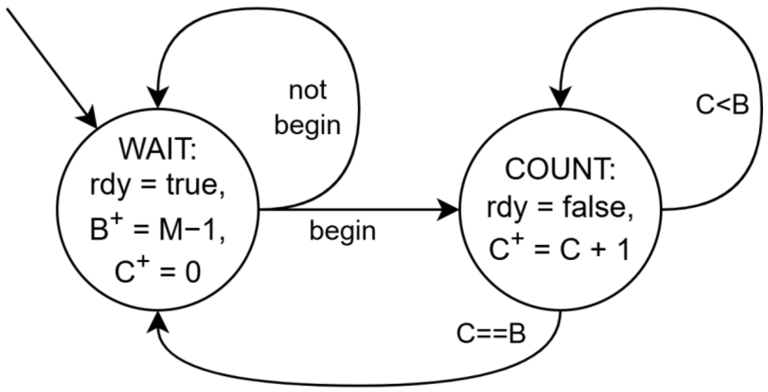
Extended Finite-State-Machines (EFSM)

EFSM design and implementation

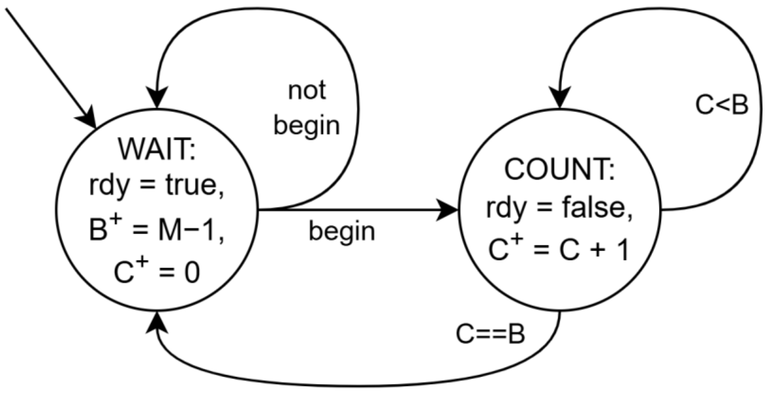
Extended Finite-State Machines (EFSMs)

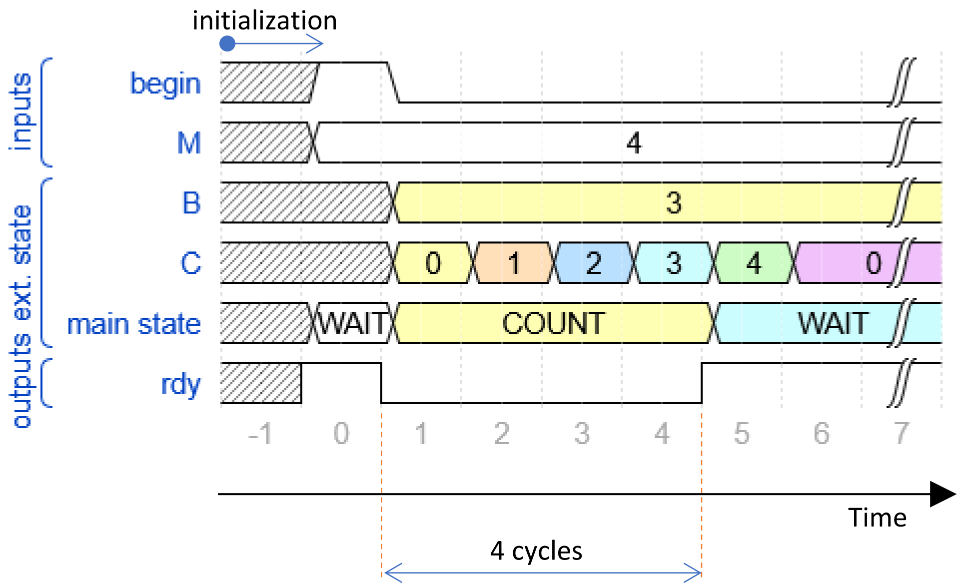
- Finite-state machines (FSM)
  - Inputs: 2^m binary encoded (i.e. $\{0, 1\}^m$) symbols
  - Outputs: 2^n symbols
  - Boolean computations
- EFSM = FSM + non-binary data + additional *state variables*
  - Full state: main state variable (*state*) + other state variables (*variables*)
    - Next values for variables are indicated by a superscript plus sign$^+$
    - Variables are updated at the same instant than the state
  - Example: Counter up to *M*: 0, 1, 2, … *M*-1
    - Variables *B* (boundary) and *C* (counter)



Timing diagrams

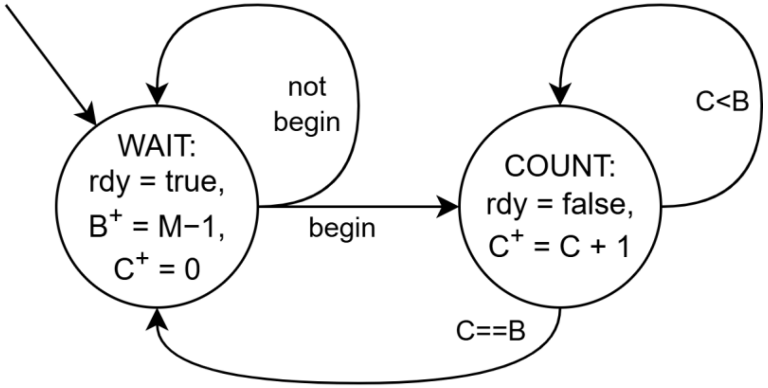
- A *timing diagram* shows how extended state and outputs vary over time
- Example:





Timing diagrams, tabular form

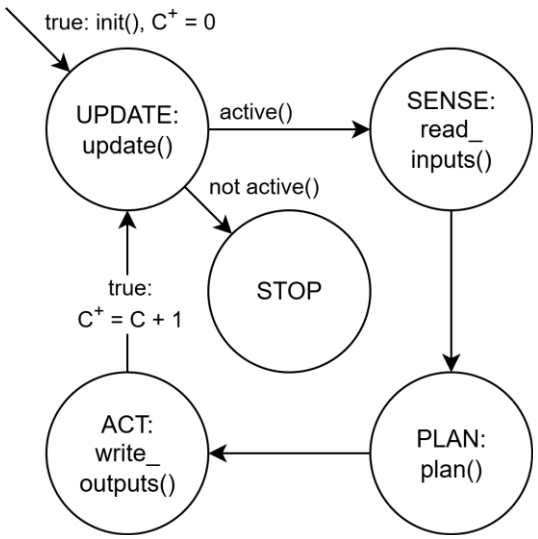
- Table filling method
  - Compute immediate outputs from extended state (and inputs)
    - Put the corresponding values into the same column, i.e., at the current state
  - Use current extended state and inputs to determine next values of state and variables
    - Put values into the next column at the right



Cycle	i+0	i+1	i+2	i+3	i+4	i+5	i+6
M	5	5	5	4	4	4	4
begin	false	false	false	true	true	false	false
state	COUNT	COUNT	WAIT	WAIT	COUNT	COUNT	COUNT
B	4	4	4	4	3	3	3
C	3	4	5	0	0	1	2
rdy	false	false	true	true	false	false	false

Implementation of EFSM

- State-based programming
  - Programs that simulate the behavior of EFSMs



  - Example in Lua →

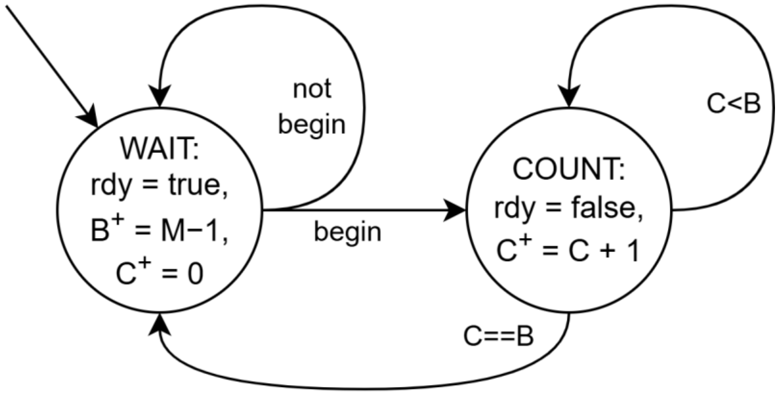
```lua
-- SIMULATION ENGINE

Counter:init()
C = 0 -- Cycle counter
while Counter:active() do
  io.write(string.format("Cycle %d:\n", C))
  Counter:monitor()
  Counter:read_inputs()
  Counter:plan()
  Counter:write_outputs()
  C = C + 1 -- next cycle
  Counter:update()
  io.write("-----------------------------------\n")
end -- while

io.write( "Program exited!\n" )
```

Implementation of EFSM, setup

- **`Counter:init()`**
  - Sets the initial values of the extended state
- **`Counter:update()`**
  - Sets the next state as the current, and the values of immediate outputs



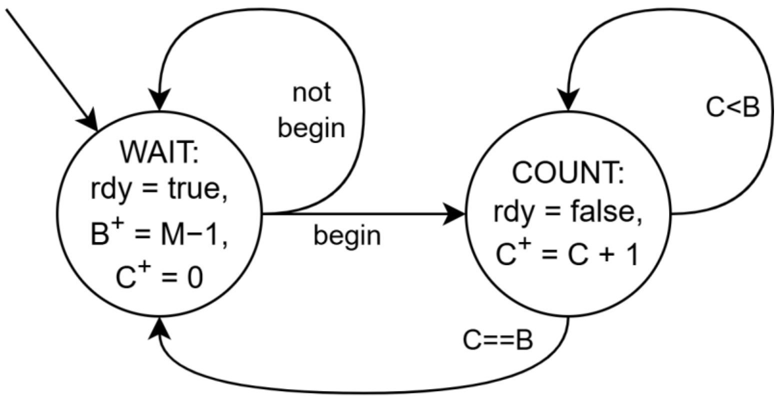

```lua
Counter = {state = {}, B ={}, C = {}}
-- Counter = {}; Counter["state"] = {}; ...

function Counter:init() -- INITIAL ARROW
-- Counter["init"] = function(self)
  self.state.next = "WAIT"
  self.C.next = -1
  self.B.next = -1
  self:update() -- self["update"](self)
end -- function

function Counter:update()
-- CURRENT STATE CHANGE AND OUTPUT FUNCTION
  self.state.curr = self.state.next
  self.B.curr = self.B.next
  self.C.curr = self.C.next
  self.rdy = self.state.curr=="WAIT"
end -- function
```

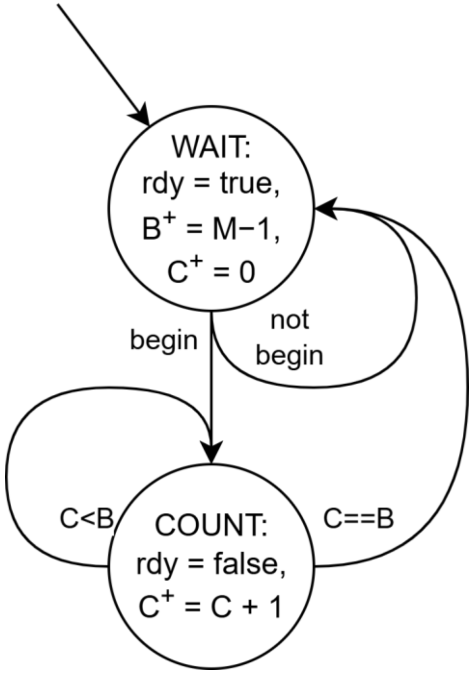
Implementation of EFSM, next state

- **`Counter:plan()`**
  - Computes next extended state




```lua
function Counter:plan()
-- TRANSITION FUNCTION
  if self.begin==nil then
    self.state.curr="STOP"
  end -- if
  if self.state.curr=="WAIT" then
    self.C.next = self.M-1
    self.C.next = 0
    if self.begin then
      self.state.next = "COUNT"
    end -- if
  elseif self.state.curr=="COUNT" then
    self.C.next = self.C.curr + 1
    if self.C.curr==self.B.curr then
      self.state.next = "WAIT"
    end -- if
  else -- stop state or error
    self.state.next = "STOP"
  end -- if..elseif
end -- function
```

Implementation of EFSM, input/output




```lua
function Counter:read_inputs()
  io.write("begin [0/1], M [whitespace=keep] = ")
  local line = io.read()
  --[[ ... --]]
  self.begin, self.M = begin, M
end -- function

function Counter:write_outputs()
  io.write(string.format("rdy = %s\n", self.rdy and "true" or "false"))
end -- function

function Counter:monitor()
  io.write(string.format("%s B = %d C = %d\n",
      self.state.curr, self.B.curr, self.C.curr))
end -- function

function Counter:active()
  return self.state.curr=="WAIT" or self.state.curr=="COUNT"
end -- function
```

Example: Characterization of a mobile robot

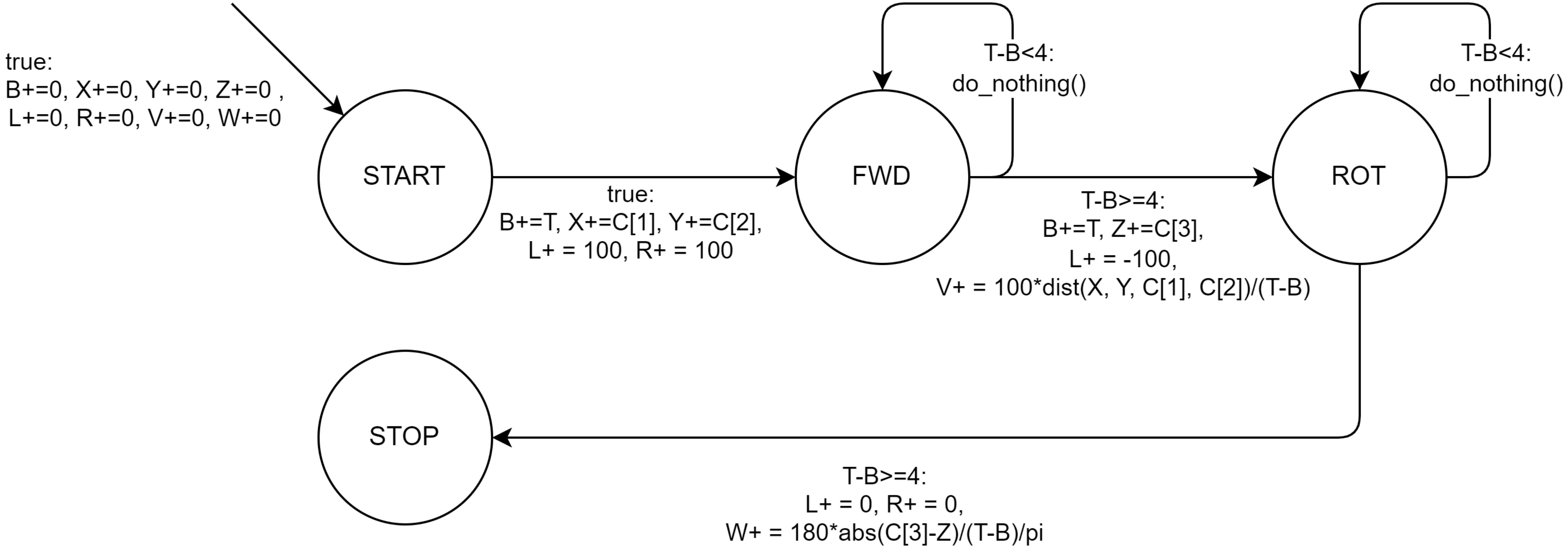
- Characterization refers to parameter identification
- For a simple mobile robot, like the UABotet, that is limited to either moving forward or turning in place, but not both at the same time, there are two main parameters:
    - Linear speed in terms of control input, $v(u_{linear})$
    - Rotation speed, $w(u_{angular})$
    - Notice that there are other factors (battery charge, floor and wheel conditions, …) that affect *v* and *w*
- In an on/off control mode, only two values are required v_{max} and w_{max}
- Characterization of UABotet implies identifying the values of v_{max} and w_{max}

Example: Characterization of UABotet, 1

- Procedure
  - Make it move for a while and compute v_{max} = distance increment / elapsed time
  - Make it rotate for a while and compute w_{max} = angle variation / elapsed time
- Controller's inputs and outputs
  - Inputs
    - C : Coordinates/pose table
      - C[1] : X position [m]
      - C[2] : Y position [m]
      - C[3] : Orientation w.r.t. Z [rad]
    - T : Time [s]
  - Outputs
    - L, R : Control values of left and right motors [−100, 100]
    - V, W: Values of linear [cm/s] and rotational [deg/s] speeds

Example: Characterization of UABotet, 2

- EFSM diagram



Notice that it is a Moore machine and that all output signals come from variables

Example: Characterization of UABotet, 3

```lua
Idify = {
  state = {}, -- Main state
  B = {},      -- Begin time
  X = {},      -- X position
  Y = {},      -- Y position
  Z = {},      -- orientation
  L = {},      -- DC value for left motor
  R = {},      -- DC value for right motor
  V = {},      -- Linear speed
  W = {}       -- Angular speed
}
function Idify:init()
  self.robot = nil -- Object handle
  if sim then self.robot = sim.getObject("..") end
  self.C = {}  -- Coordinates (X, Y, angle wrt. Z)
  self.T = 0   -- Time
  self.state.next = "START"; self.B.next = 0
  self.X.next = 0; self.Y.next = 0; self.Z.next = 0
  self.L.next = 0; self.R.next = 0
  self.V.next = 0; self.W.next = 0
  self:update()
end -- function
function Idify:update()
  self.state.curr = self.state.next
  self.B.curr = self.B.next
  --[[ omitted --]]
end -- function
```

```lua
function Idify:plan()
  if not sim and self.C[1]==nil then
    self.state.curr = "STOP"
  end  -- if
  if self.state.curr=="START" then
    self.B.next = self.T
    self.X.next = self.C[1]; self.Y.next = self.C[2]
    self.L.next = 100; self.R.next = 100
    self.state.next = "FWD"
  elseif self.state.curr=="FWD" then
    --[[ omitted --]]
  elseif self.state.curr=="ROT" then
    local delay = self.T-self.B.curr
    if delay>4 then
      local dA = math.abs(self.C[3]-self.Z.curr)
      self.W.next = 180*dA/delay/math.pi
      self.L.next = 0; self.R.next = 0
      self.state.next = "STOP"
    end -- if
  else -- STOP or error
    self.state.next = "STOP"
  end -- if..ifelse
end -- function
function Idify:active()
  return self.state.curr=="START"
      or self.state.curr=="FWD"
      or self.state.curr=="ROT"
end -- function
```

Example: Characterization of UABotet, 4

- The program changes I/O upon being linked to a simulator

```
function Idify:read_inputs()
  if sim then
    self.T = sim.getSimulationTime()
    local position =
sim.getObjectPosition(self.robot, sim.handle_world)
    local eulerAngles =
sim.getObjectOrientation(self.robot, sim.handle_world)
    self.C[1] = position[1]
    self.C[2] = position[2]
    self.C[3] = eulerAngles[3]
  else -- console input
    self.T = self.T + 0.05
    io.write(string.format(
      "> T = %.4fs or ...", self.T))
    local newT = tonumber(io.read())
    if newT then self.T = newT end
    io.write(string.format("> X = "))
    self.C[1] = tonumber(io.read())
    if self.C[1] then
      io.write(string.format("> Y = "))
      self.C[2] = tonumber(io.read())
      --[[ omitted --]]
    end -- if
  end -- if
end -- function
```

```
function Idify:write_outputs()
  print(string.format(
    "< L= %i, R= %i, V= %.2fcm/s, W= %.2fdeg/s\n",
    self.L.curr, self.R.curr, self.V.curr, self.W.curr))
  if sim then
    sim.setInt32Signal("DC_left", self.L.curr)
    sim.setInt32Signal("DC_right", self.R.curr)
  end -- if
end -- function
```

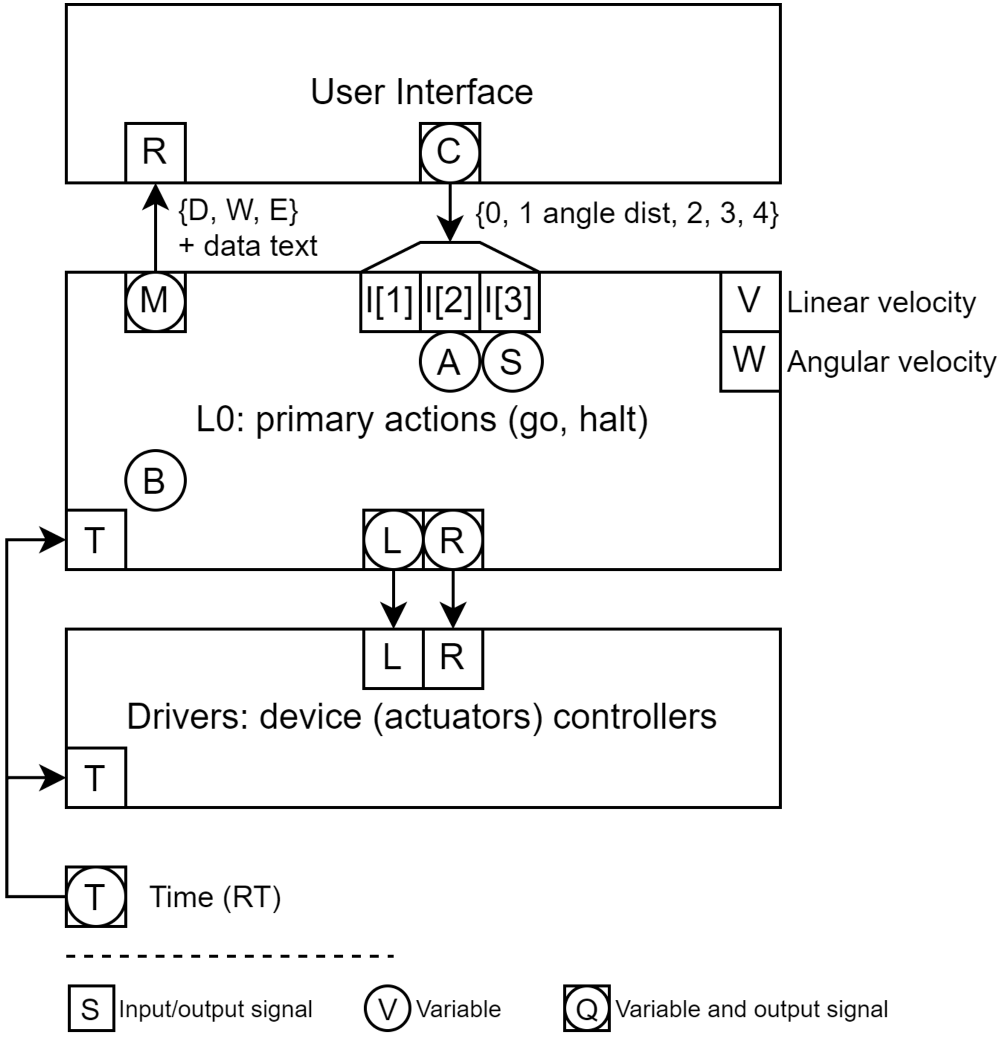
Example: Characterization of UABotet, 5

- If not associated with a sim, then simulates EFSM itself

```
if not sim then -- LOCAL SIMULATION ENGINE
  Idify:init()
  C = 0 -- Cycle counter
  while Idify:active() do
    io.write(string.format("Cycle %d:\n", C))
    --Idify:monitor()
    Idify:read_inputs()
    Idify:plan()
    Idify:write_outputs()
    C = C + 1 -- next cycle
    Idify:update()
    io.write("----------------------------------------\n")
  end -- while
  io.write(string.format("Cycle %d:\n", C))
  Idify:write_outputs() -- outputs at error or STOP state
  io.write("Program exited!\n")
end -- if
```

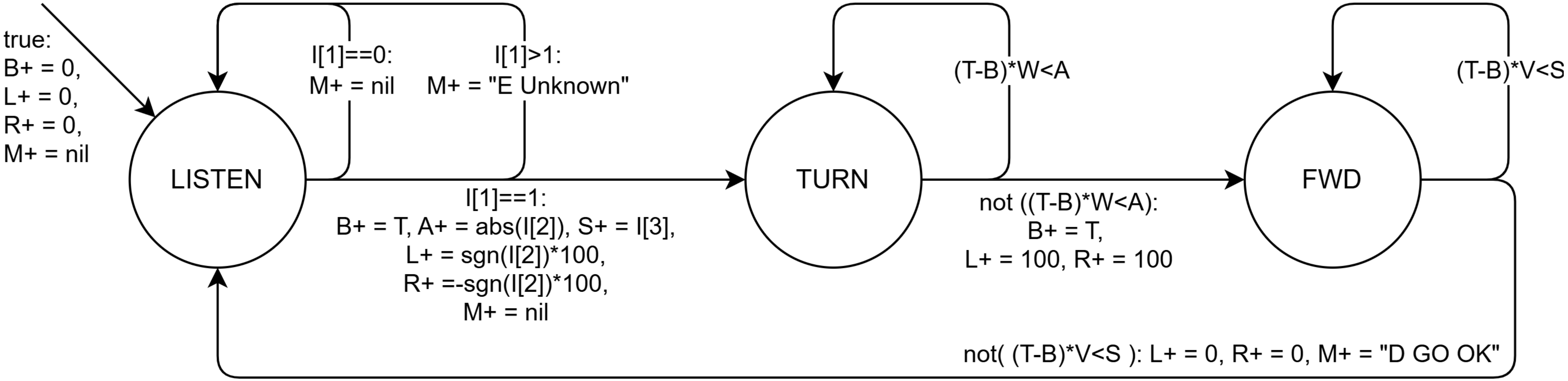
Exercise 2: Motion control of a mobile robot

- Specification
  - Move the robot to the specified polar coordinates (angle, distance) any time instruction command is 1 ("go") and stop
- Controller's inputs and outputs
  - Inputs
    - I : Instruction table
      - I[1] : Instruction command = 0 (none), 1 (go)
      - I[2] : Angle to turn [deg], from −90 to +90
      - I[3] : Distance to move [cm], from 0 to 255
    - T : Time [s]
    - V, W: Values of linear and rotational speeds (constant)
  - Outputs
    - L, R : Control values of left and right motors
    - M : Reply message, nil or string



Exercise 2: Motion control of a mobile robot

- Controller's EFSM basic model

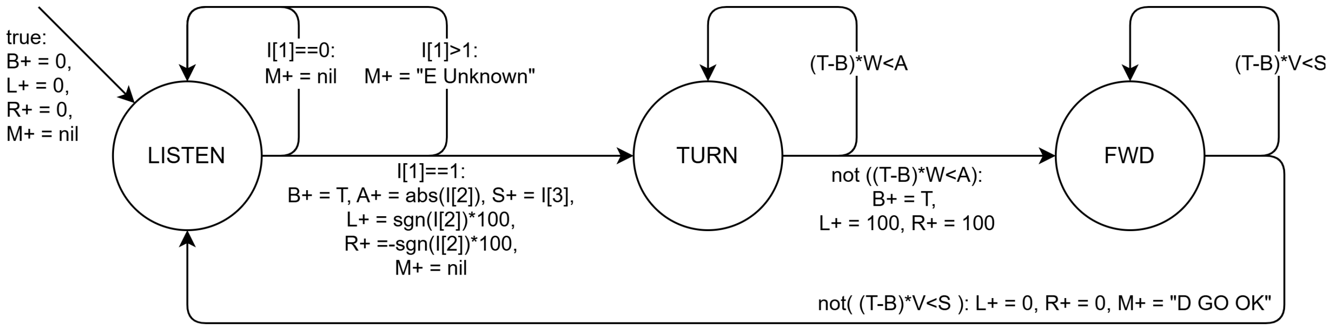


sgn(*a*) returns -1, 0, or +1 for *a*<0, *a*==0, and *a*>0, respectively

Exercise 2: Motion control of mobile robot

- Complete the timing diagram table below
  - Assume V = 10cm/s, W = 10deg/s

Cycle	i+0	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8
I	{4, nil, nil}	{1, 90, 100}	{4, nil, nil}	{0, nil, nil}	{0, nil, nil}	{0, nil, nil}	{1, 10, 0}	{0, nil, nil}	{0, nil, nil}
T	1003.5	1008.3	1014.7	1018.4	1023.8	1029.5	1033.6	1038.1	1043.4
state	LISTEN								
A	-30								
S	20								
B	904.3								
M	nil								
L	0								
R	0								



Exercise 2+: Motion control of a mobile robot

- Is there any problem when instruction is “1 0 10”?, and “1 15 0”?

- Program the EFSM for the controller in Lua
  - Use the characterization program [M02_characterization.lua] as example
  - Complete `L0Main:init()`, `L0Main:update()` and `L0Main:plan()` in M02_simple_GO.lua