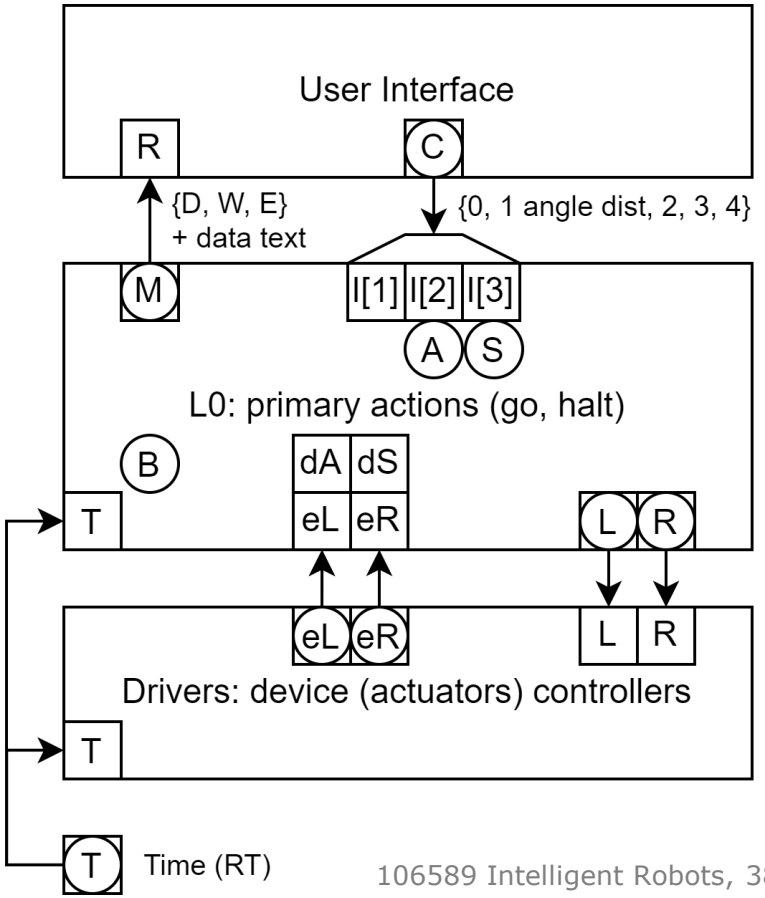
Controllers’ architecture

Concurrent EFSM

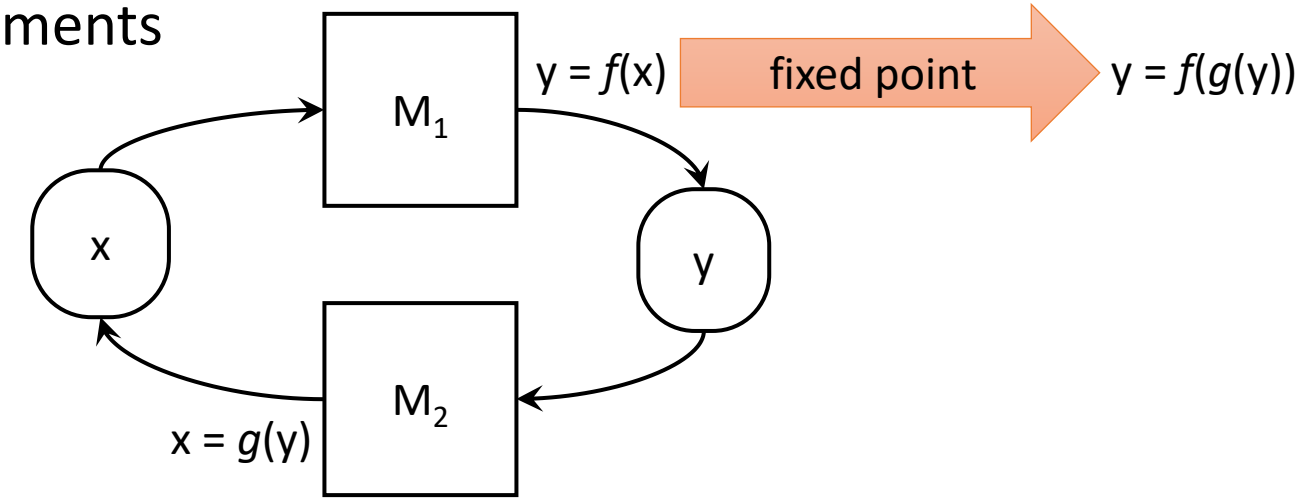
Concurrent EFSM

- Systems combine different elements which work independently
- In the robot controller, there are three different components
  - Application's user interface
  - Main controller (application's logic)
  - Device controllers (drivers)
- Single-EFSM models are OK for simple behaviors but dealing with complexity
  implies diving behaviors into different tasks,
  each one simpler than their composition



Concurrent EFSM, immediate assignments problem

- Assignments can be
  - Immediate, i.e. made at the beginning of the cycle, in no time
    - V = value (result of an expression evaluation, including constants)
  - Delayed, i.e. made before the beginning of the next cycle, after some time has passed
    - V^+ = value (result of an expression evaluation, including constants)
- Variables updated in delayed assignments
- Output signals come from
  - Present values of model variables
  - Immediate assignments
    What if immediate assignments depend on inputs?

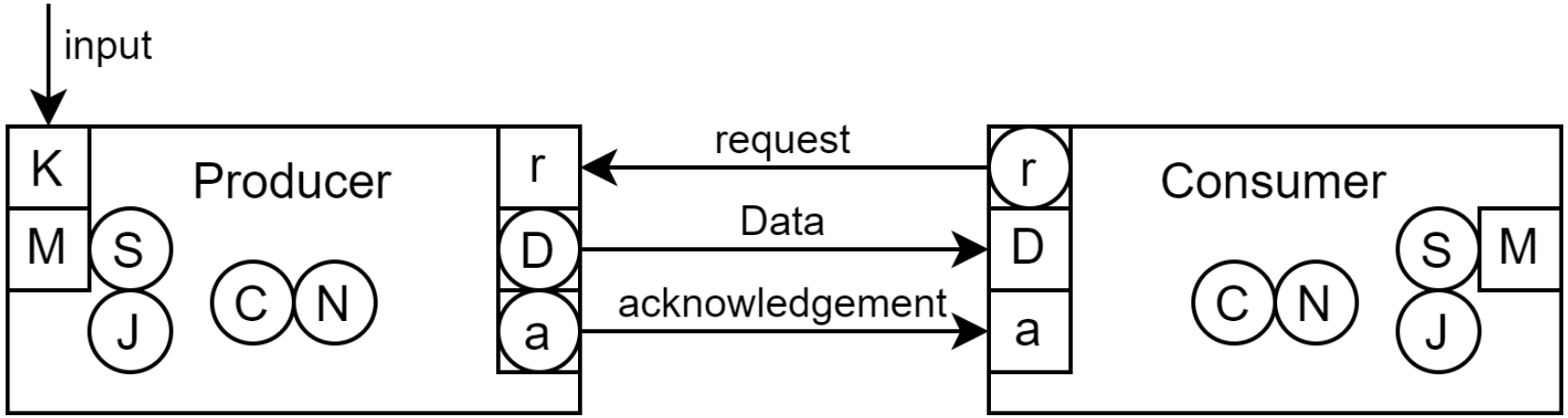


Concurrent EFSM, immediate assignments solution

- Immediate assignments to output signals could lead to circular dependencies, e.g. y = $f(g(y))$, which must be detected and solved!
  - Solution of circular dependencies imply
    1. Computing their fixed points, i.e. values of y that satisfy equation y = $f(g(y)$
    2. Eventually replace y by the desired values, if they exist, thus **breaking the cyclic dependency**
- Prevent circular dependencies from taking place by…
  - **Not using inputs in immediate assignments to output signals**
  - The SW synthesis pattern does not allow these type of assignments

Concurrent EFSM, producer-consumer

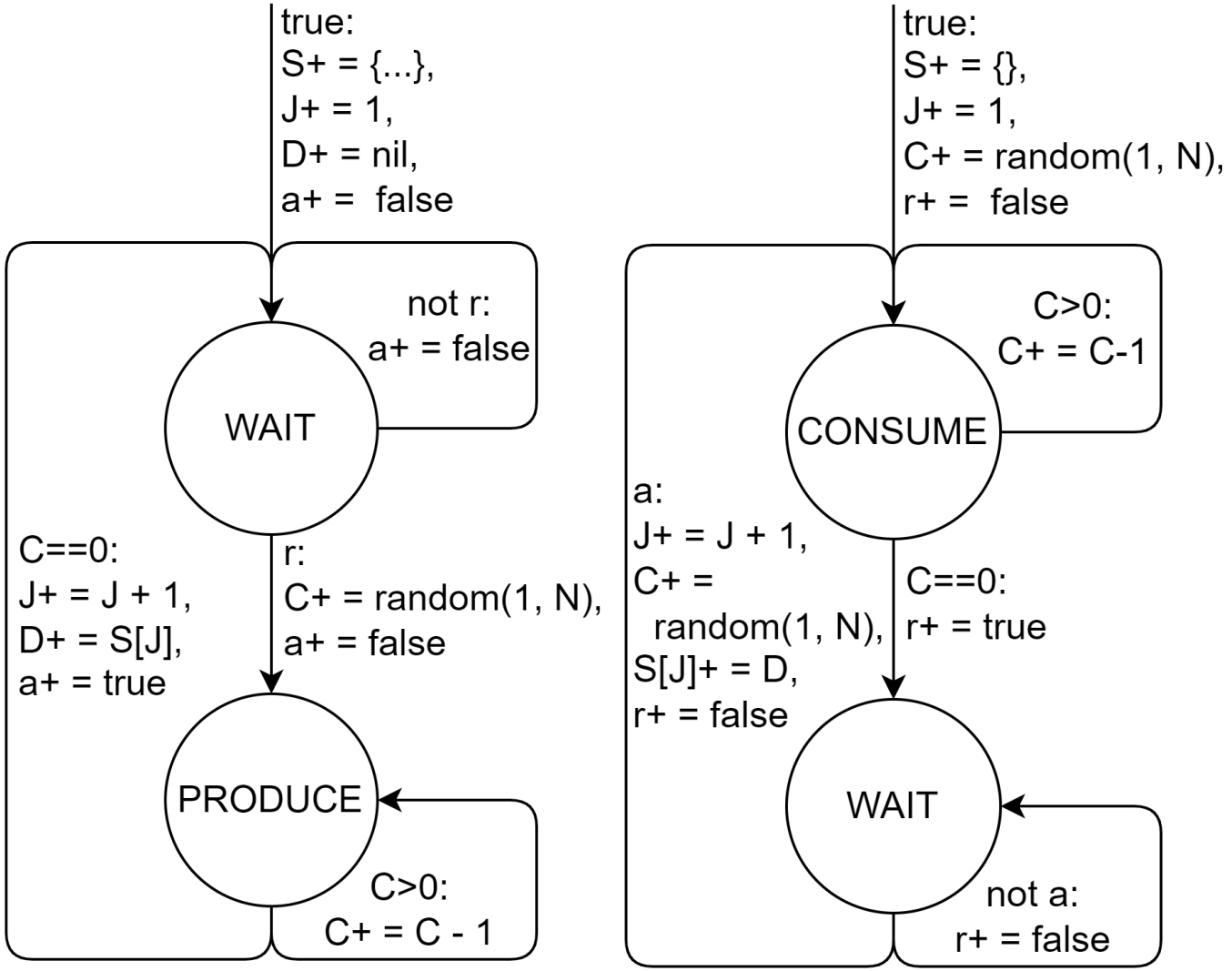
- Organizing a complex behavior into components that model simpler behaviors often lead to producer-consumer situations
(e.g. sensor data collecting and data-processing tasks)
- The producer-consumer example:



- Data transmission from producer to consumer
  - Synchronous: both modules know the cycles when data are available
  - Asynchronous: a *handshake protocol* is required to determine at which cycles data are available

Concurrent EFSM, producer-consumer

- EFSM models of generic producers and consumers
- Signals
  - **S**: internal storage with maximum capacity M not dealt with in the models
  - **J**: index of S
  - **D**: data signal
  - **a**: acknowledgment signal
  - **r**: request signal
  - **C**: counter to model random cycles when data is available or requested



Concurrent EFSM, producer-consumer

- Software syntheses
  - Modules at independent tables (objects)
  - Additional **get_**<output_signal> methods
    - Function **read_inputs** with parameters from other modules
- Set up and main loop
  - Example from producer-consumer system:

```
--SETUP
Producer:init()
Consumer:init()

-- MAIN LOOP
cycle = 0
while Producer:active() and Consumer:active() and cycle<100 do
  io.write(string.format("Cycle = %04i:\n", cycle))
  Producer:monitor(); Consumer:monitor()
  Producer:write_outputs(); Consumer:write_outputs()
  Producer:read_inputs(Consumer:get_req());
  Consumer:read_inputs(Producer:get_ack(), Producer:get_Data())
  Producer:step(); Consumer:step()
  Producer:forward(); Consumer:forward()
  cycle = cycle + 1
end -- while
```

Concurrent EFSM, producer-consumer

- Producer

```
Producer = {
  S = {"H", "e", "l", "l", "o", ",",
       " ", "w", "o", "r", "l", "d"},
  M = 0,        -- storage size
  N = 4,        -- random cycles
  r = nil,      -- input request
  state = {},   -- state
  J = {},       -- data index
  C = {},       -- internal counter
  D = {},       -- output data
  a = {},       -- output acknowledgement
```

```
 init = function(self)
    self.M = #self.S
    self.state.next = "WAIT"
    self.J.next = 1
    self.D.next = nil
    self.a.next = false
    math.randomseed(os.time())
    self:forward()
 end -- function init()
 ,
 forward = function(self)
    self.state.curr = self.state.next
    self.J.curr = self.J.next
    self.C.curr = self.C.next
    self.D.curr = self.D.next
    self.a.curr = self.a.next
 end -- function init()
 ,
```

Concurrent EFSM, producer-consumer

- Producer

```
read_inputs = function(self, req)
  self.r = req
end -- function read_inputs()
,
write_outputs = function(self)
  local D = self.D.curr or "nil"
  local a = "false"
  if self.a.curr then a = "true" end
  io.write(string.format(
   "< Producer: a = %s D = %s\n", a, D))
end -- function write_outputs()
,
get_ack = function(self)
  return self.a.curr
end
,
get_Data = function(self)
  return self.D.curr
end
,
```

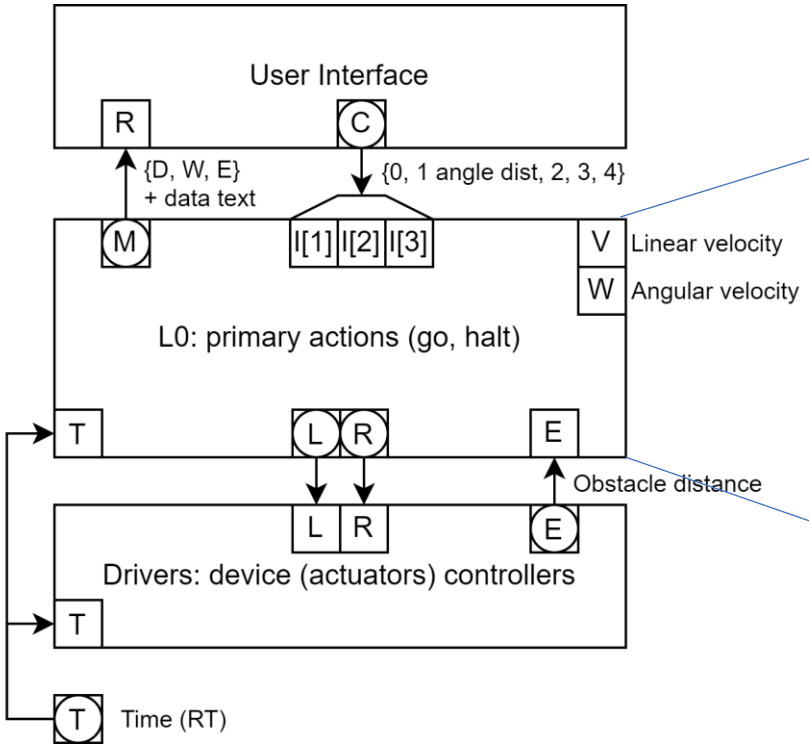
```
step = function(self)
  if self.state.curr=="WAIT" then
    if self.r then
      self.C.next = math.random(1, self.N)
      self.a.next = false
      self.state.next = "PRODUCE"
    else
      self.a.next = false
    end -- if
  elseif self.state.curr=="PRODUCE" then
    if self.C.curr>0 then
      self.C.next = self.C.curr - 1
    else
      self.J.next = self.J.curr%self.M + 1
      self.D.next = self.S[self.J.curr]
      self.a.next = true
      self.state.next = "WAIT"
    end -- if
  else -- Stop state or error
  end -- if chain
end -- function step()
,
active = function(self)
  return self.state.curr=="WAIT" or
    self.state.curr=="PRODUCE"
end, -- function active()
```

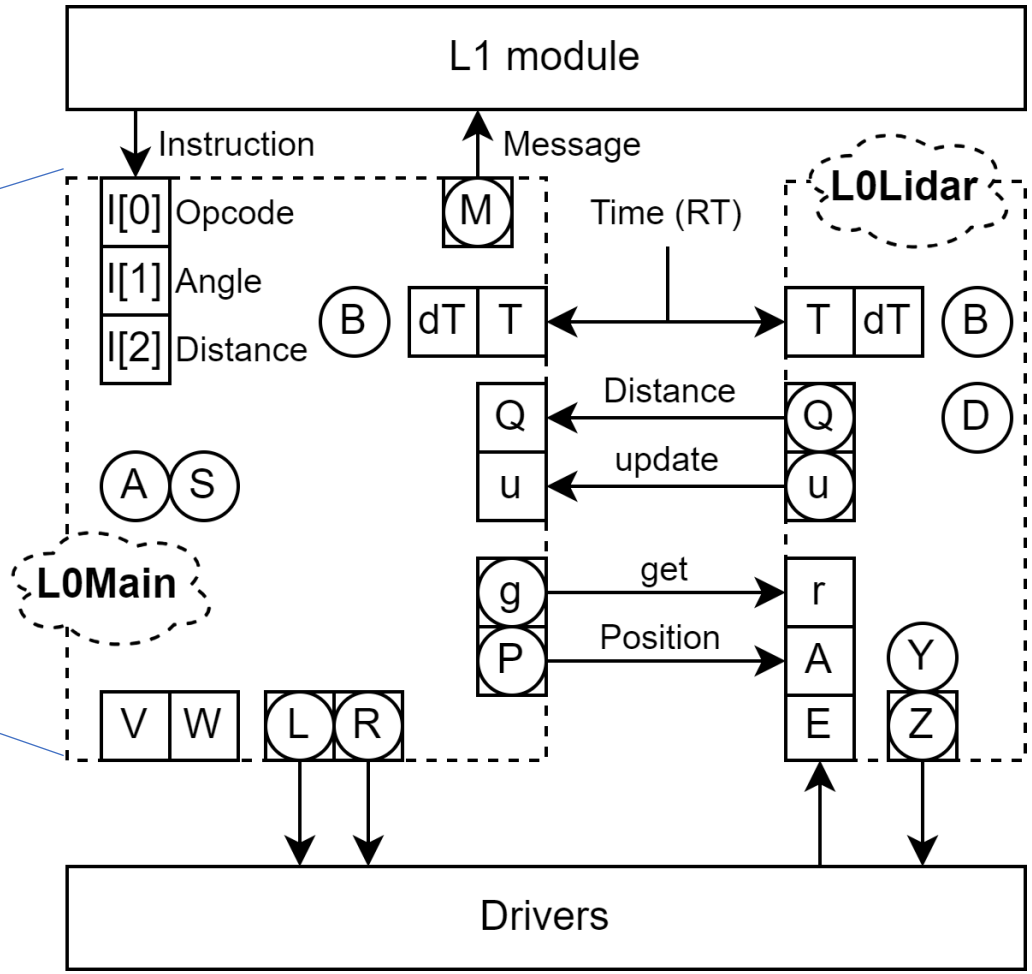
Concurrent EFSM, producer-consumer

- Activity
  - Complete the code of the producer-consumer system [M04_producer-consumer.lua] in accordance with the given EFSM
  - Note that attention to communication protocol signals is paid only in the WAIT states and, consequently, positive pulses in other states will be lost. How will you solve this?

Concurrent EFSM, L0 main + lidar controllers

- L0 Level with two controllers





Exercise 4: Reverse engineering L0 lidar

- Use code to obtain the corresponding EFSM graph
  - Code in L0_Lidar.lua