Cyber-Physical Systems

Laura Nenzi

Università degli Studi di Trieste
II Semestre 2018

Lecture 3: Concurrent Modeling

[Many Slides due to J. Deshmukh, Toyota]

Functional Components

1. *Classical* model of computation: Functional or Transformational Programs

   - Start from a given input,
   - Produce a certain output and then **terminate**
   - Desired functionality can be described by a mathematical function
   - Emphasis is on data computation

Reactive Components

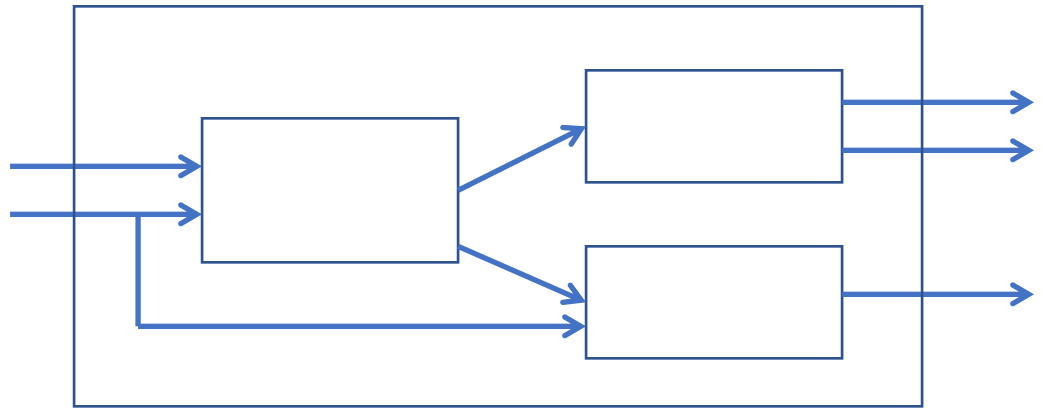
2. Reactive Programs:

- It maintains and internal state
- Continuously interact with the environment at a rate decided by the environment
- Emphasis is on system-environment interaction; e.g. airline autopilot, mail-servers, etc.

Synchronous Models

- All components execute in a sequence of rounds in lock-step

- Example:
  - Components in a digital hardware circuit with a central global clock
  - Fixed-step Simulation Models of Discrete Components in Simulink

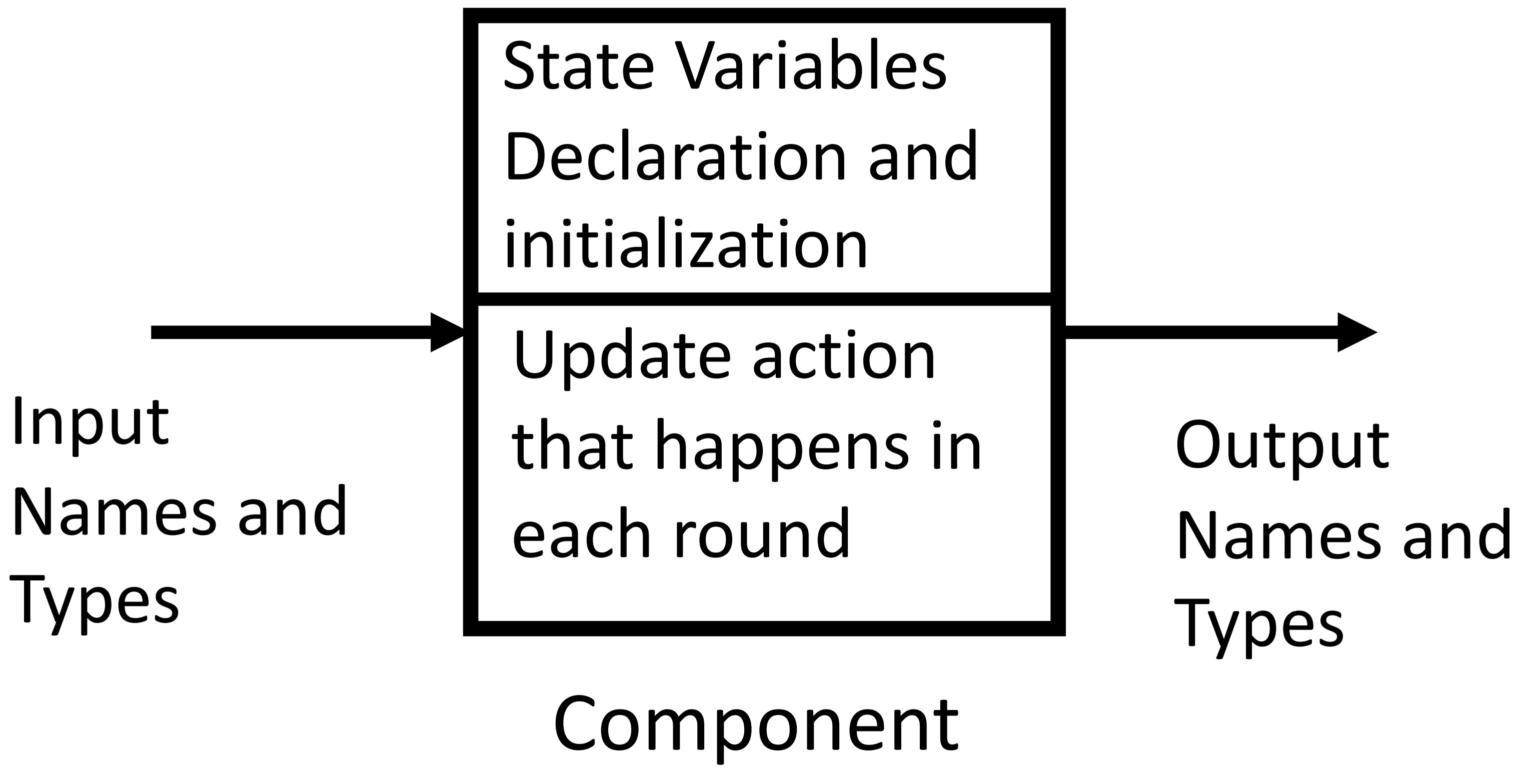


Synchronous languages

- Benefit: system design is simpler if we use a simple round-based computation

- Challenge: How do we ensure synchronous execution when components may execute on different hardware?

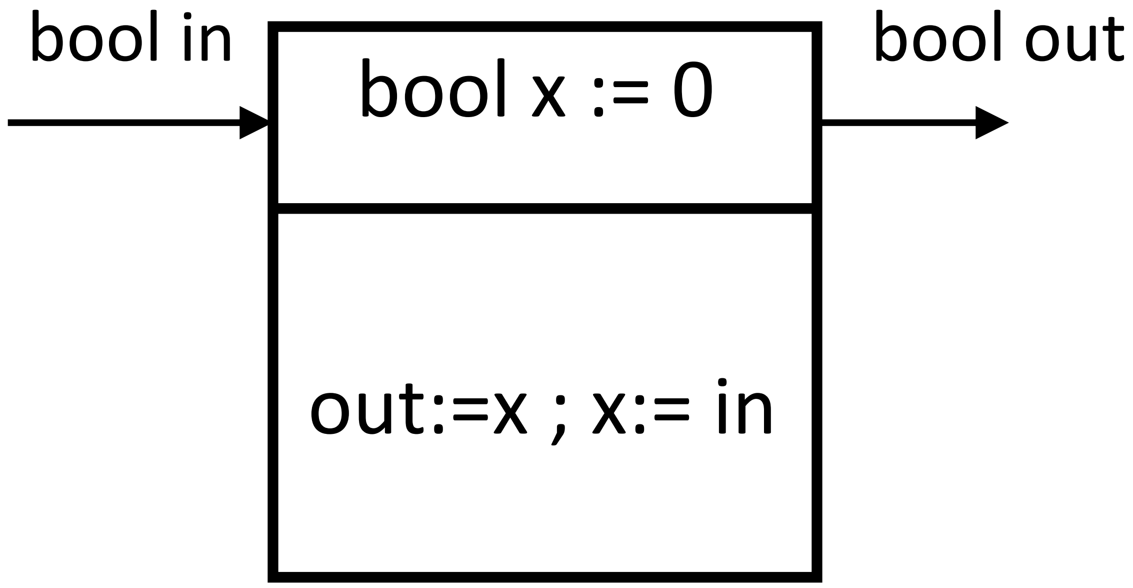
Simple Representation of a Synchronous Component


State Variables Declaration and initialization
Update action that happens in each round
Input Names and Types
Output Names and Types
Component

Simplest synchronous component: delay

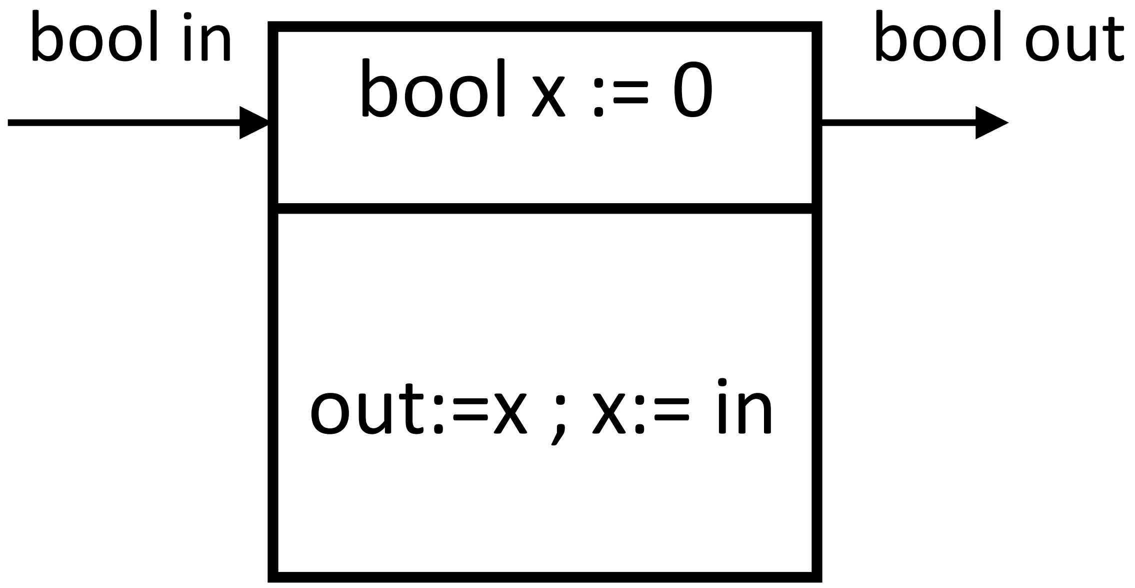
(Boolean = { 0, 1})

- Input variable: in of type Boolean
- Output variable: out of type Boolean
- State variable: x of type Boolean, initialized to 0
- In each round, component updates output from the state and state from input



Execution of “Delay”

- Initialize state to 0
- Repeatedly execute rounds
- In each round:
  - Choose value for input (provided from environment, e.g. by user)
  - Execute update code



$$\begin{tikzcd} 0 \arrow[r, "1/0"] & 1 \arrow[r, "1/1"] & 1 \arrow[r, "0/1"] & 0 \arrow[r, "0/0"] & 0 \arrow[r, "1/0"] & 1 \arrow[r, dotted] & {} \end{tikzcd}$$

Synchrony hypothesis

- Time needed to execute update is negligible compared to arrival times between consecutive inputs

- Synchronous execution is a *logical abstraction*
  - *Execution time of update code is 0*
  - *Production of outputs, updates to state and arrival of inputs happen instantaneously*

- With multiple components, assume all execute synchronously and simultaneously

- Burden on design-time to validate hypothesis

Let’s Formalize an SRC

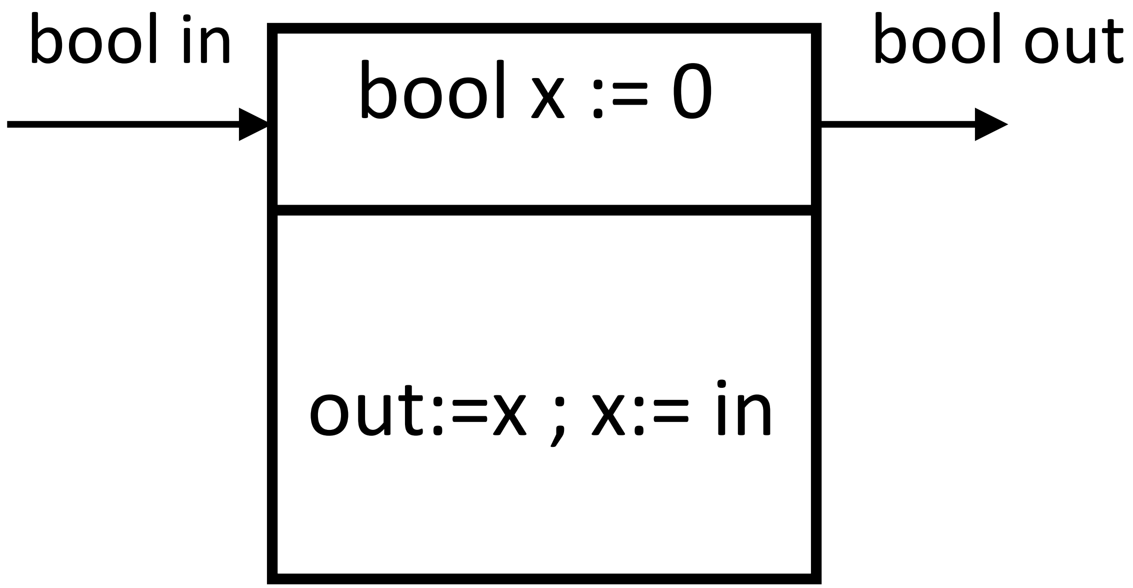
- SRC is defined as a tuple: (I, O, X, U), where:

Symbol	Designation	Examples
I	Set of Inputs	$\{ in_1, in_2\}$ $\{\}$
X	Set of State Variables	$\{x_1, x_2, x_3\}$ $\{\}$
O	Set of Outputs	$\{out_1, out_2\}$ $\{\}$
U	Set of Updates	$x \coloneqq in_1 \land in_2 \,; out_1 \coloneqq x$ $x_1 \coloneqq in_1 + in_2; x_2 \coloneqq in_3; out: = x_1$

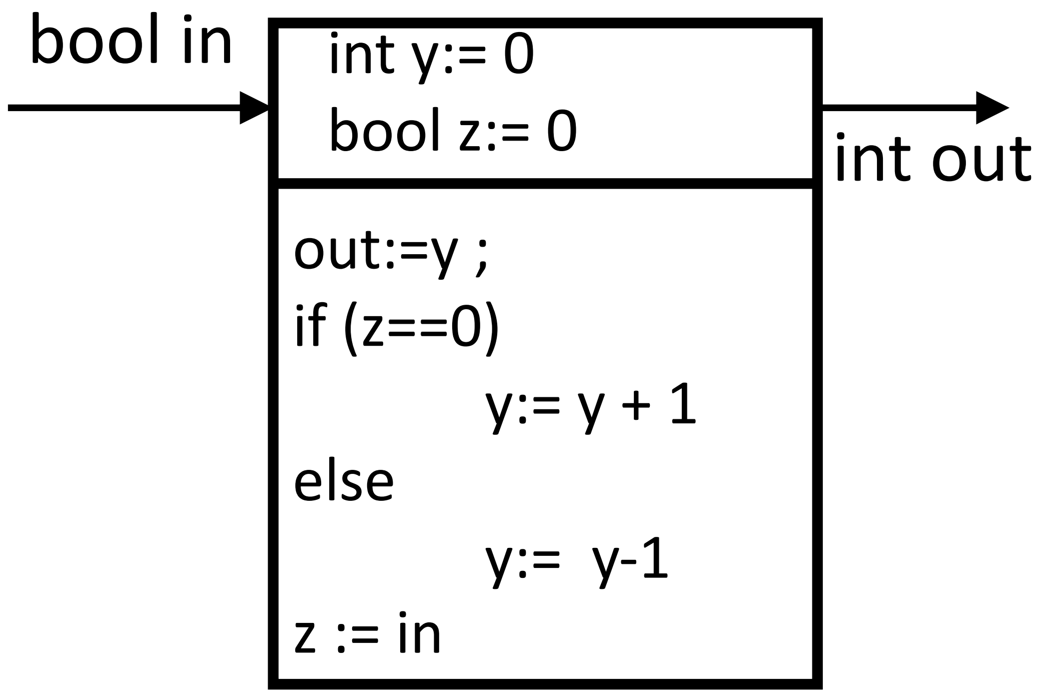
Semantics of updates & initialization

- Let the set of input, output, and state values be Q_I, Q_O, Q_X
- Semantics of the initialization function:
  - At time/round 0, maps the state variables to some specified value (or values) in Q_X
- Semantics of the update function (some sequence of conditionals and assignments):
  - A set R of transitions where each transition is of the form: $q \xrightarrow{i/o} \mathrm{q}'$, where q is the old value of the state variables, q' is the new value of the state variables, i is the value of the input in that round, and o is the value of the output
  - R is a subset of $Q_X \times Q_I \times Q_O \times Q_X$

What are the Q_I, Q_O, Q_X for these SRCs?



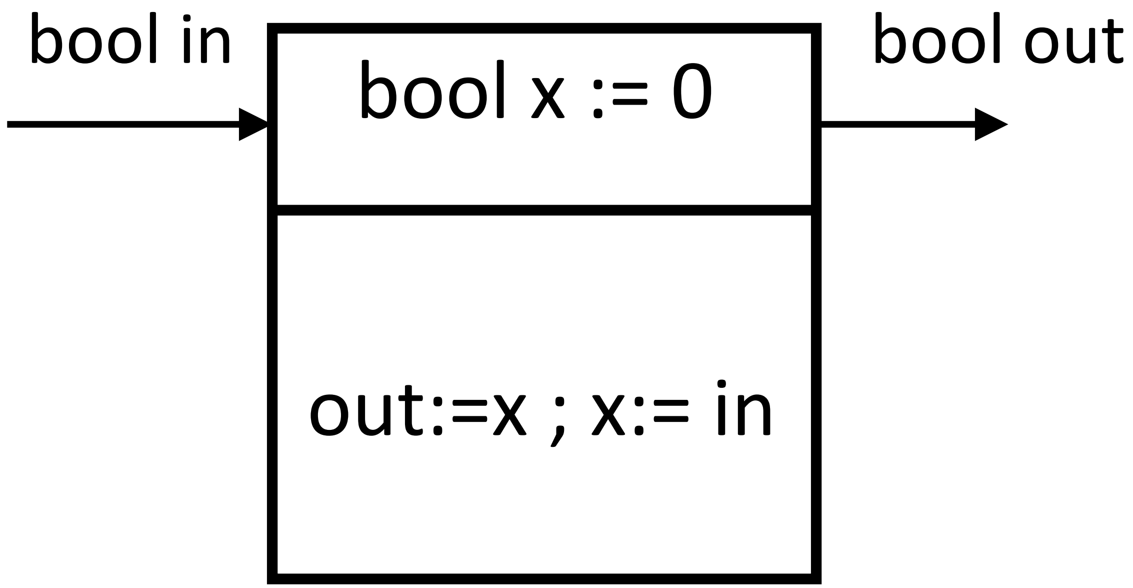
$Q_I = \{0,1\}, Q_X = \{0,1\}, Q_O = \{0,1\}$



```
out:=y ;
if (z==0)
        y:= y + 1
else
        y:=  y-1
z := in
```

$Q_I = \{0,1\}, Q_X = \text{int}\times\{0,1\}, Q_O = \text{int}$

Transitions for Delay



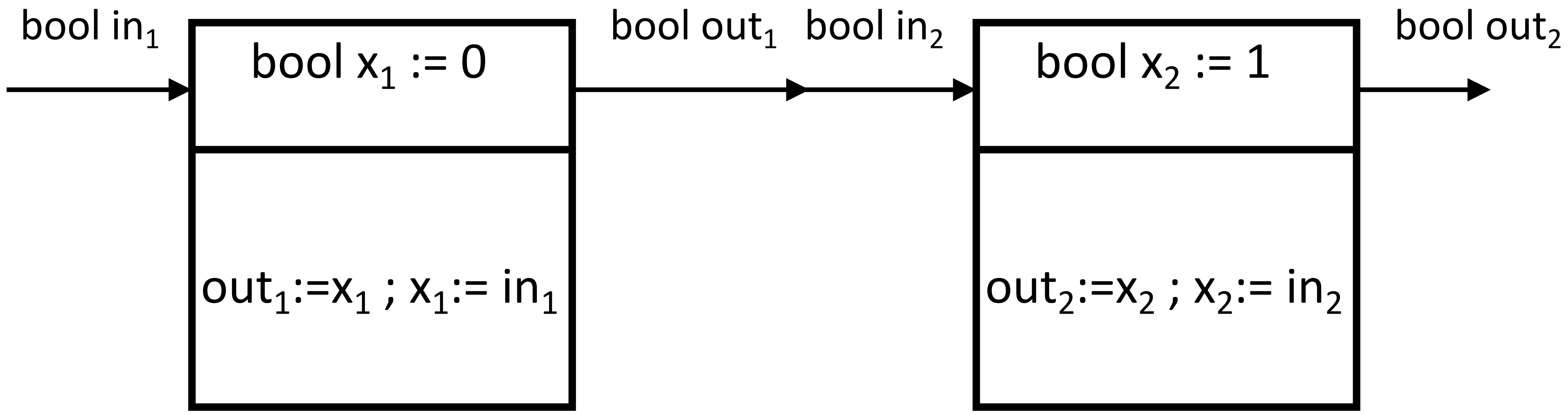
$$0 \xrightarrow{0/0} 0$$

$$0 \xrightarrow{1/0} 1$$

$$1 \xrightarrow{0/1} 0$$

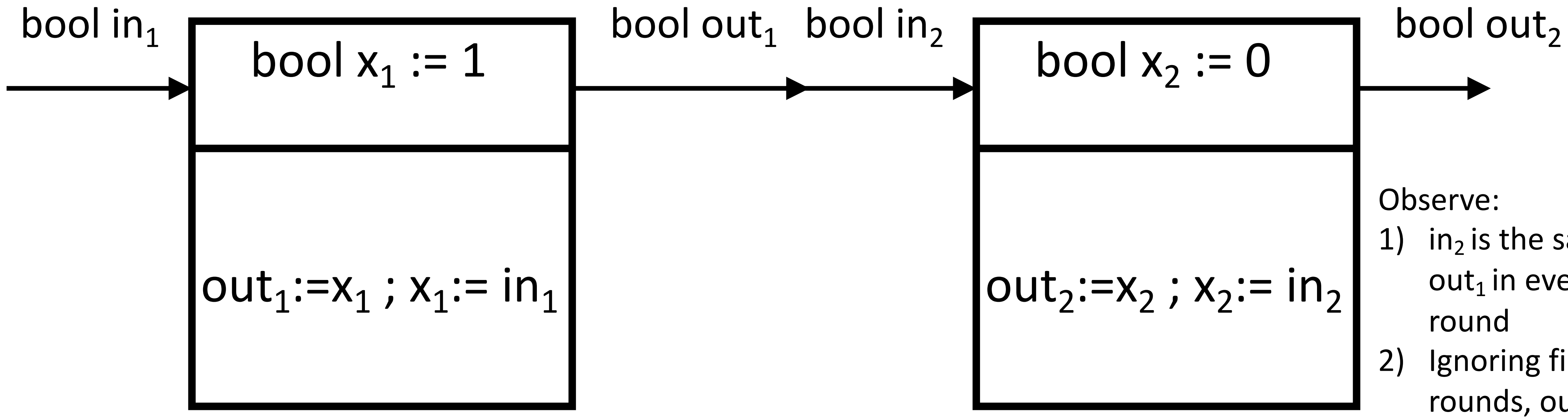
$$1 \xrightarrow{1/1} 1$$

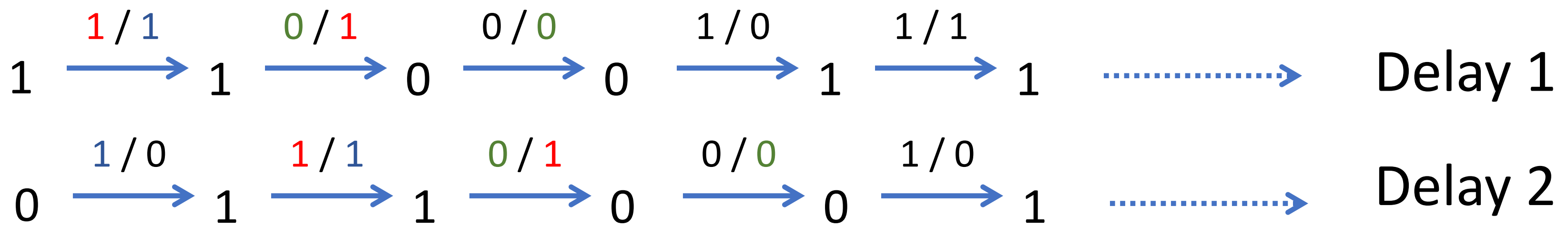
Composition of Synchronous Components



Delay sequentially composed with Delay

Composition of Synchronous Components



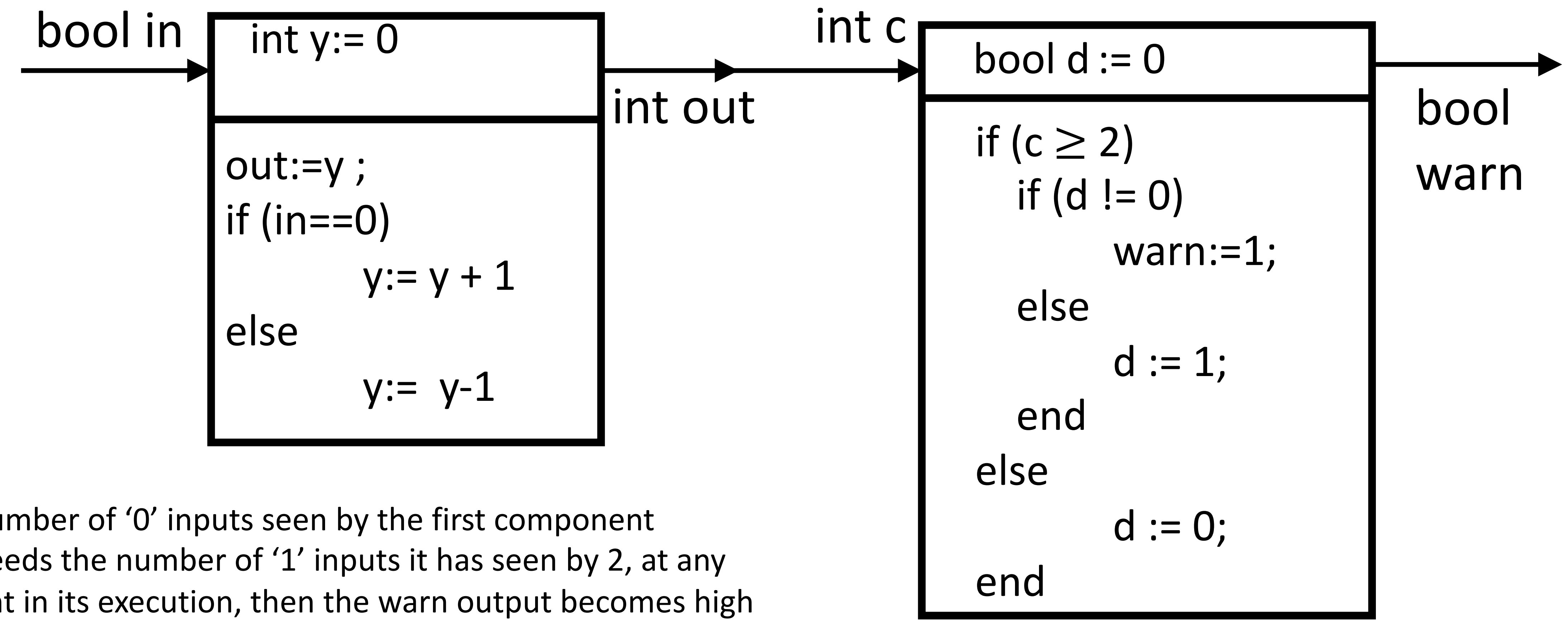


$$1 \xrightarrow{1/1} 1 \xrightarrow{0/1} 0 \xrightarrow{0/0} 0 \xrightarrow{1/0} 1 \xrightarrow{1/1} 1 \cdots\rightarrow$$ Delay 1
$$0 \xrightarrow{1/0} 1 \xrightarrow{1/1} 1 \xrightarrow{0/1} 0 \xrightarrow{0/0} 0 \xrightarrow{1/0} 1 \cdots\rightarrow$$ Delay 2

Observe:

1) in_2 is the same as out_1 in every round
2) Ignoring first 2 rounds, outputs of d2 are the inputs to d1 delayed by 2 rounds

What does this model achieve?



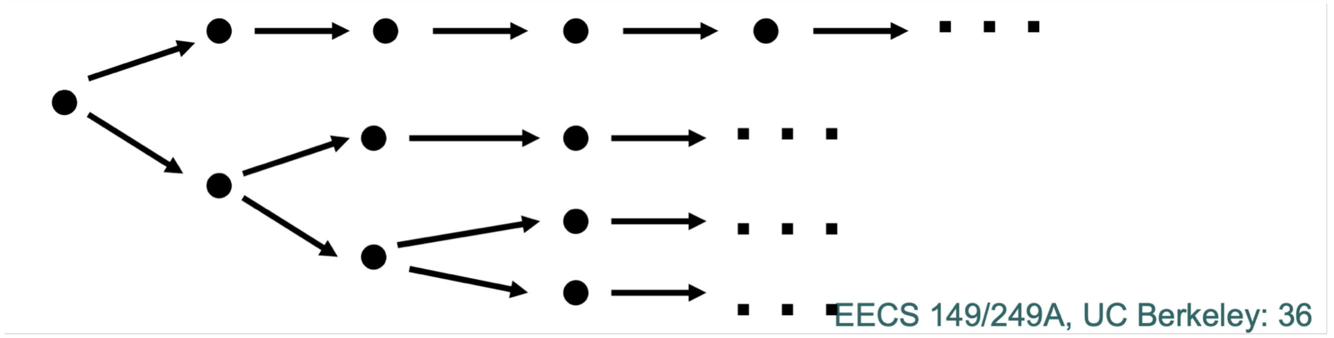
```
out:=y ;
if (in==0)
        y:= y + 1
else
        y:=  y-1
```

```
if (c ≥ 2)
    if (d != 0)
            warn:=1;
    else
            d := 1;
    end
else
            d := 0;
end
```

If number of ‘0’ inputs seen by the first component exceeds the number of ‘1’ inputs it has seen by 2, at any point in its execution, then the warn output becomes high

Deterministic Component

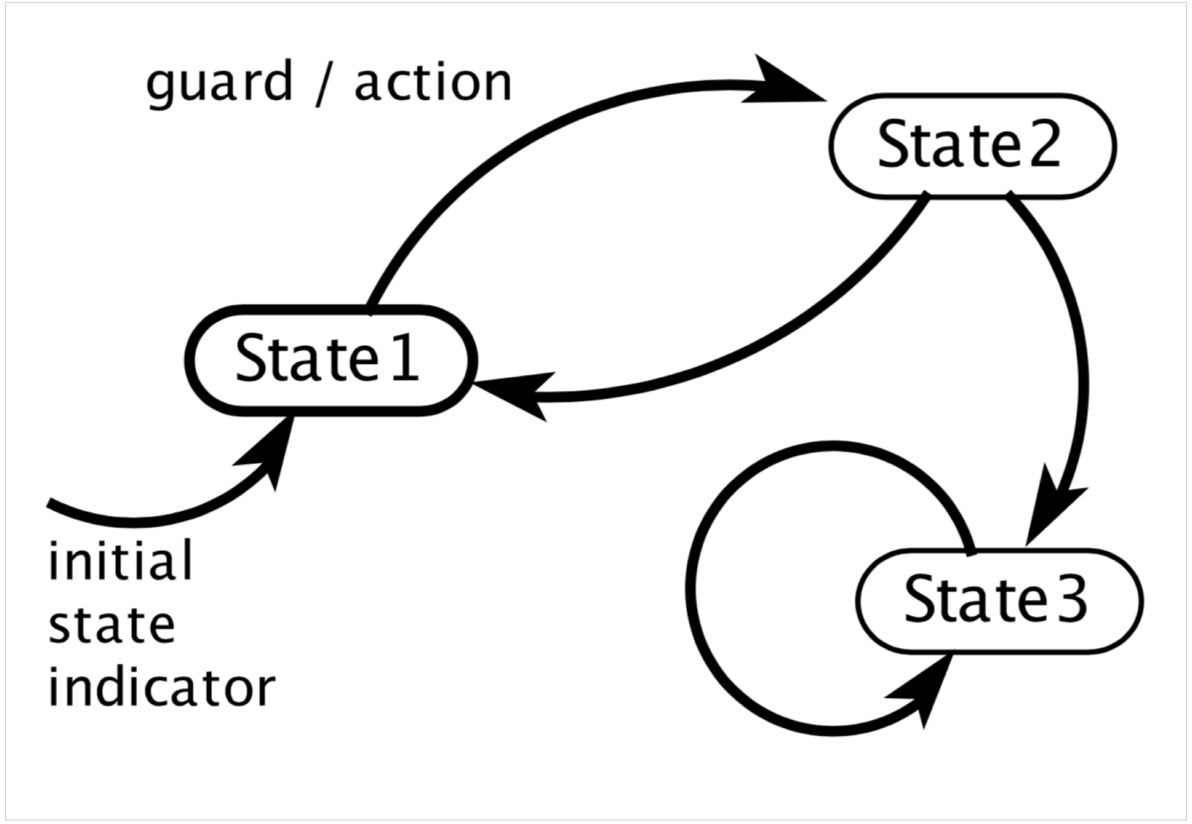
- An SRC is deterministic if:
  - It has a single initial state
  - Updates ensure that for every state q and input i, there is a unique state q' and output o such that (q, i, o, q') is a transition
- Determinism means for same input sequence, you get same state/output sequence every single time
- Note:
  - Nondeterminism is useful for modeling uncertainty/unknown and compactness



  - It is not the same as probabilistic/random choice!

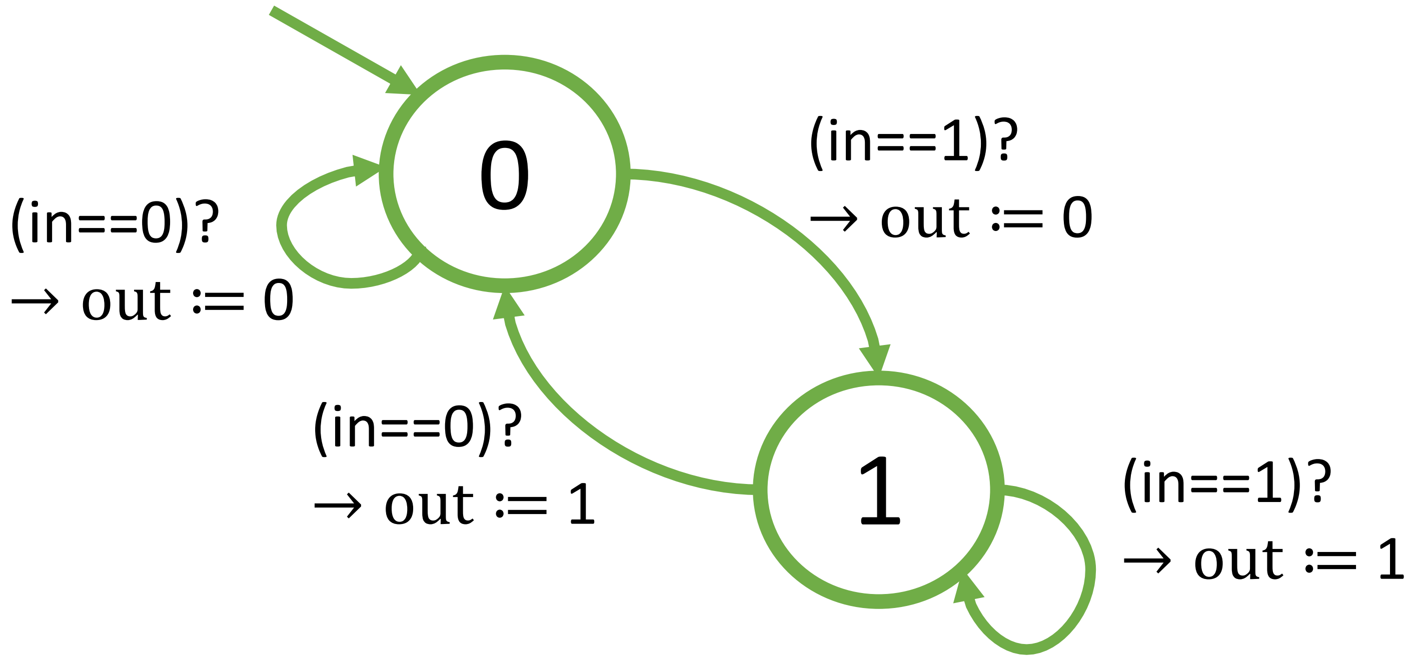
Extended State Machines

- Commonly used to describe behavior of MBD models

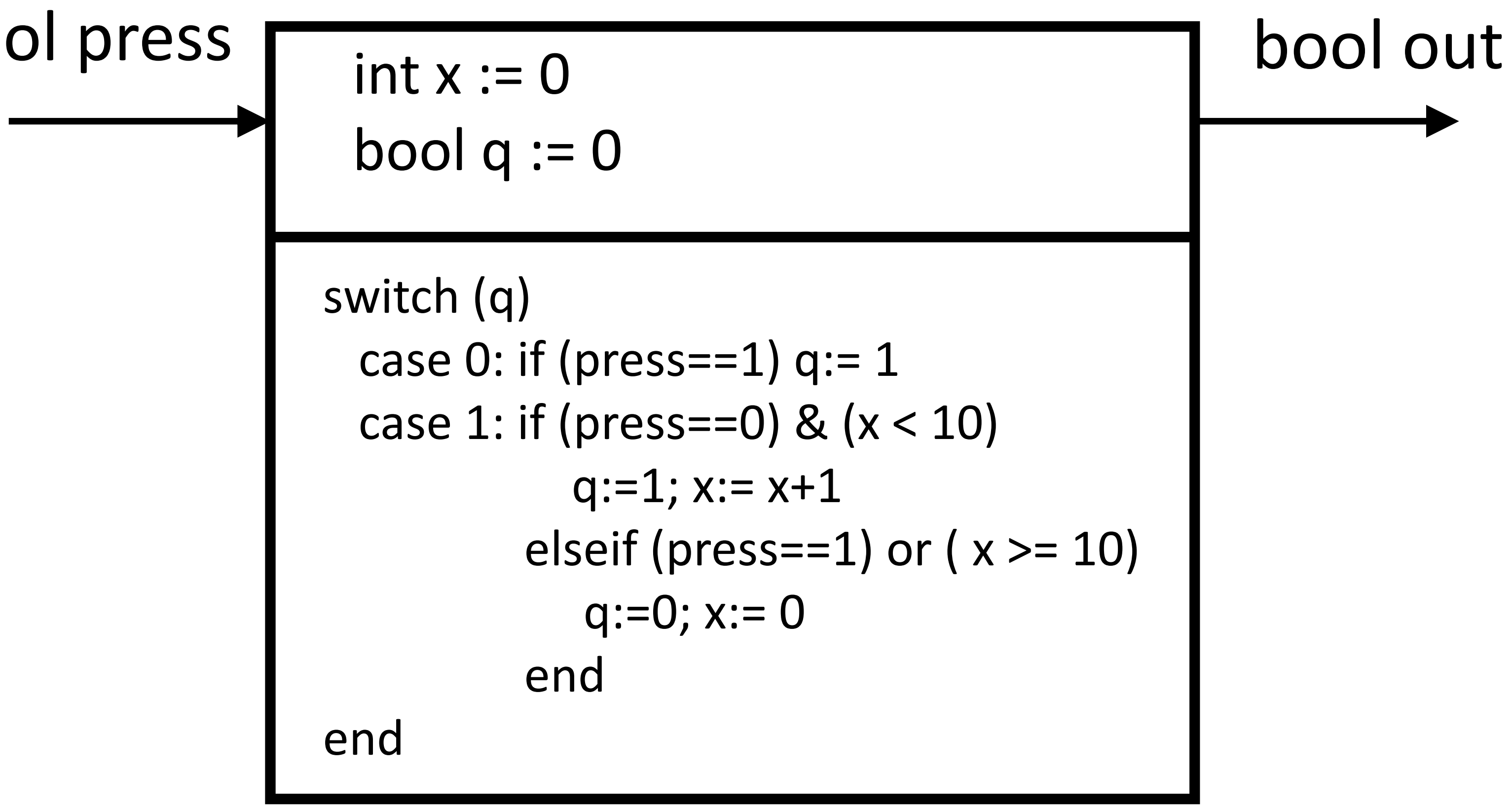


Extended State Machines

- Does this ESM remind you of something?


$$\begin{tikzcd}
\arrow[r] & 0 \arrow[loop left, "{(in==0)? \to out := 0}"] \arrow[rd, bend left, "{(in==1)? \to out := 0}"] & \\
& & 1 \arrow[lu, bend left, "{(in==0)? \to out := 1}"] \arrow[loop right, "{(in==1)? \to out := 1}"]
\end{tikzcd}$$

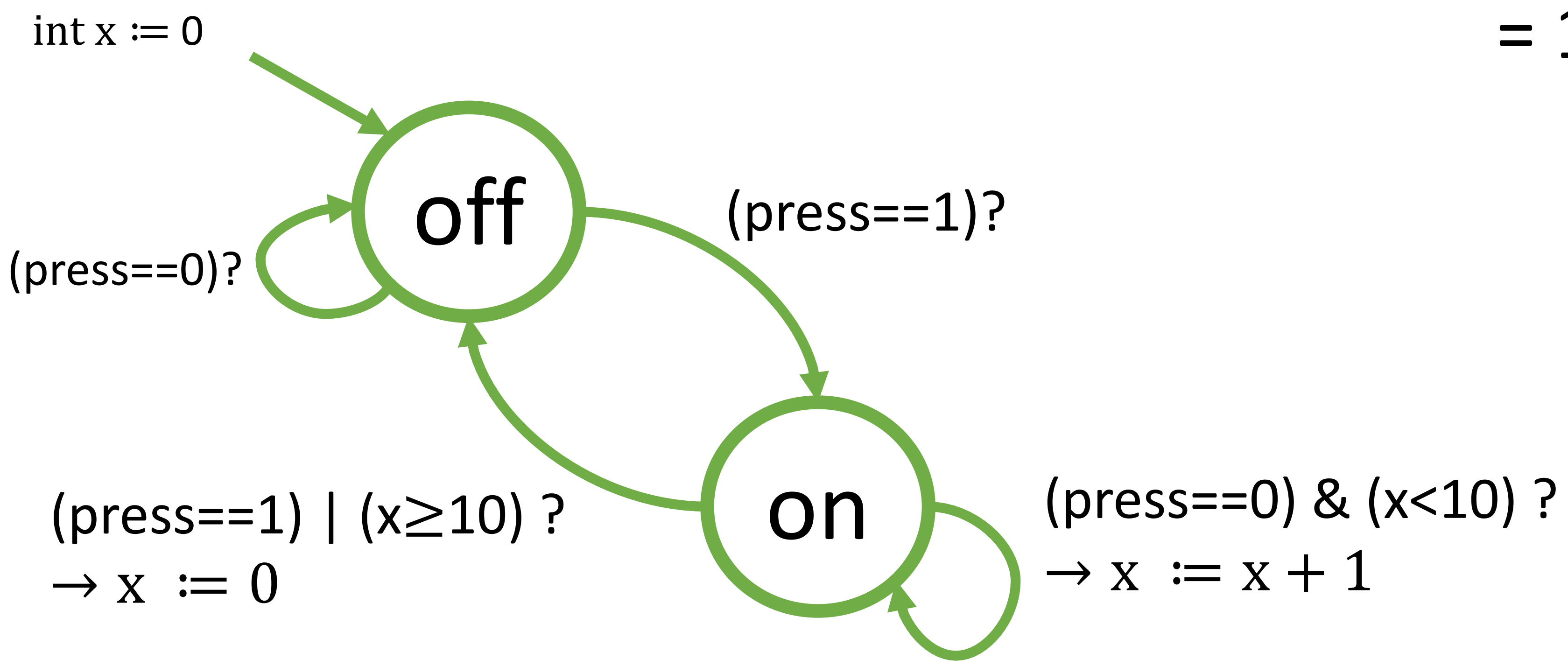
Component Switch: What does this do?



```
int x := 0
bool q := 0
```

```
switch (q)
  case 0: if (press==1) q:= 1
  case 1: if (press==0) & (x < 10)
              q:=1; x:= x+1
            elseif (press==1) or ( x >= 10)
              q:=0; x:= 0
            end
end
```

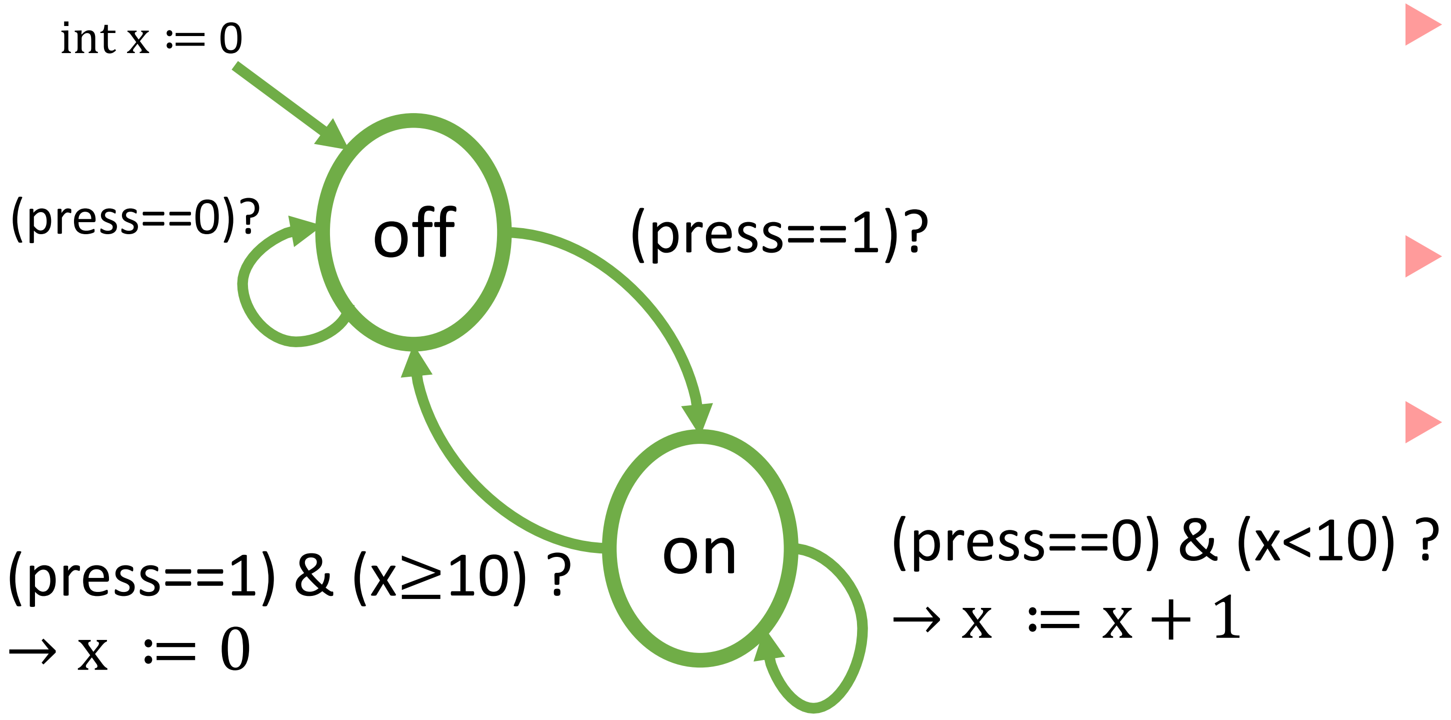
ESM corresponding to Switch SRC



int x ≔ 0
off
on
(press==0)?
(press==1)?
(press==1) | (x≥10) ? → x ≔ 0
(press==0) & (x<10) ? → x ≔ x + 1

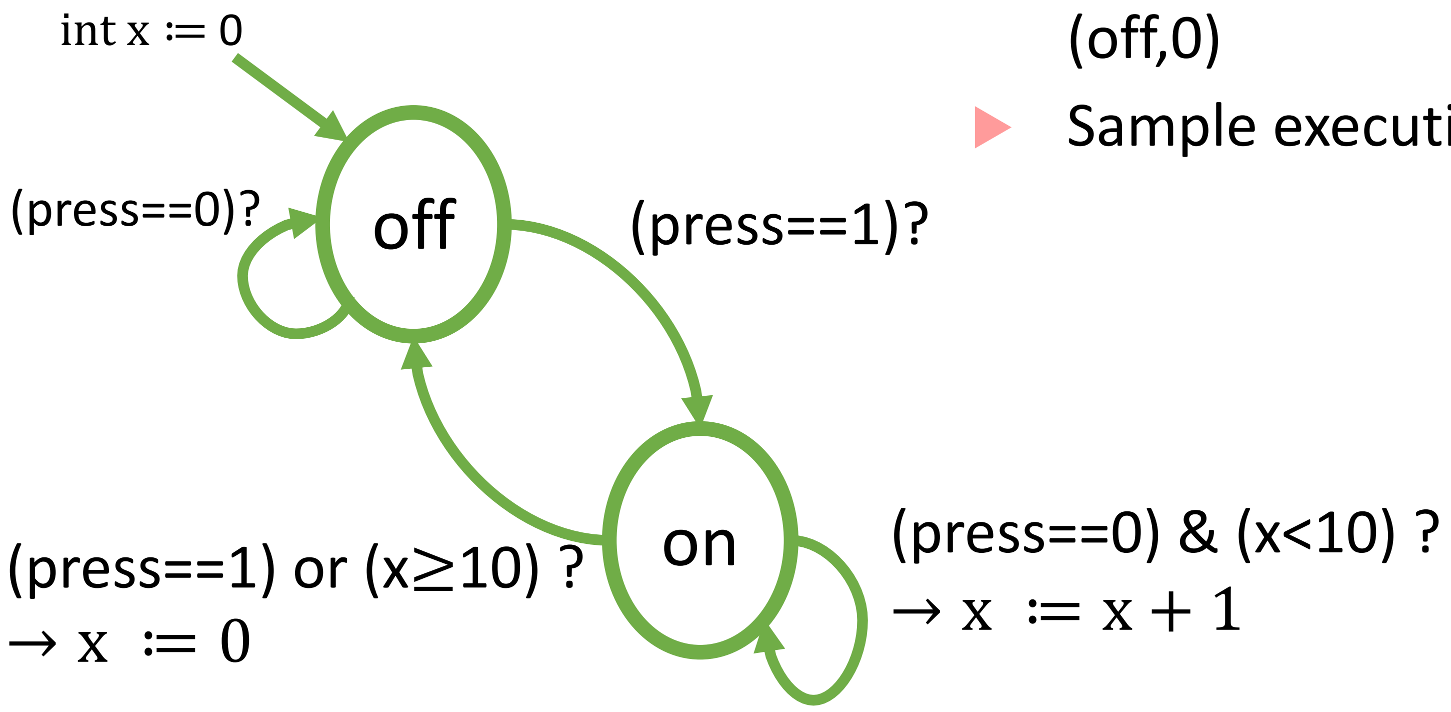
q = 0 : off
  = 1 : on

ESM notation



- Implicit variable called "mode" that is a *discrete* state variable over some finite enumeration. Here: {on, off}
- SRC transition may correspond to mode-switch
- Each mode-switch has guard/update. Example:
  - Guard: (press==0) & (x<10) and
  - Update: x:= x+1

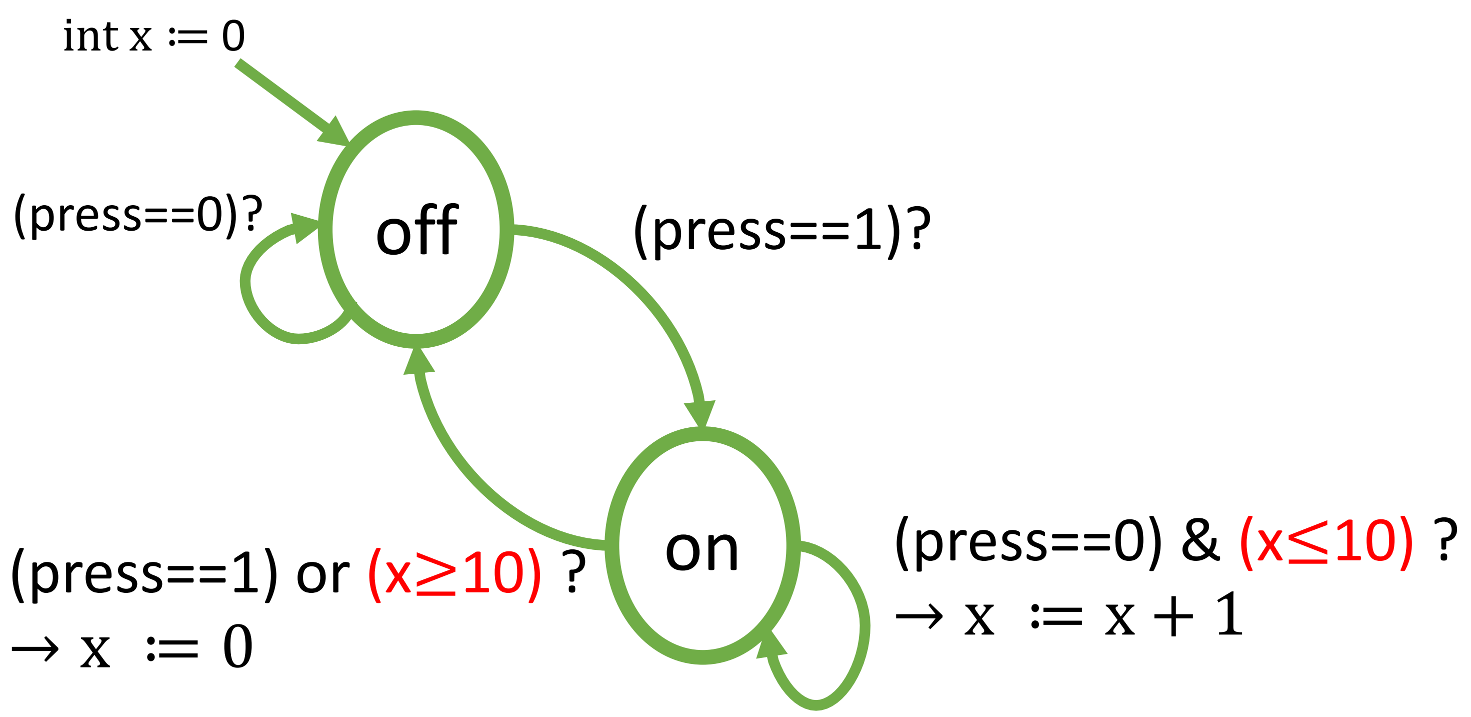
ESM execution



- Start in mode off; initial state = (off,0)
- Sample executions:

$$\begin{array}{cc}
(off,0) & (off,0) \\
\downarrow 0 & \downarrow 0 \\
(off,0) & (off,0) \\
\downarrow 1 & \downarrow 1 \\
(on,0) & (on,0) \\
\downarrow 0 & \downarrow 0 \\
(on,1) & (on,1) \\
\vdots & \vdots \\
\downarrow 0 & \downarrow 0 \\
(on,10) & (on,5) \\
\downarrow 0 & \downarrow 1 \\
(off,0) & (off,0)
\end{array}$$

ESM transitions could be nondeterministic!

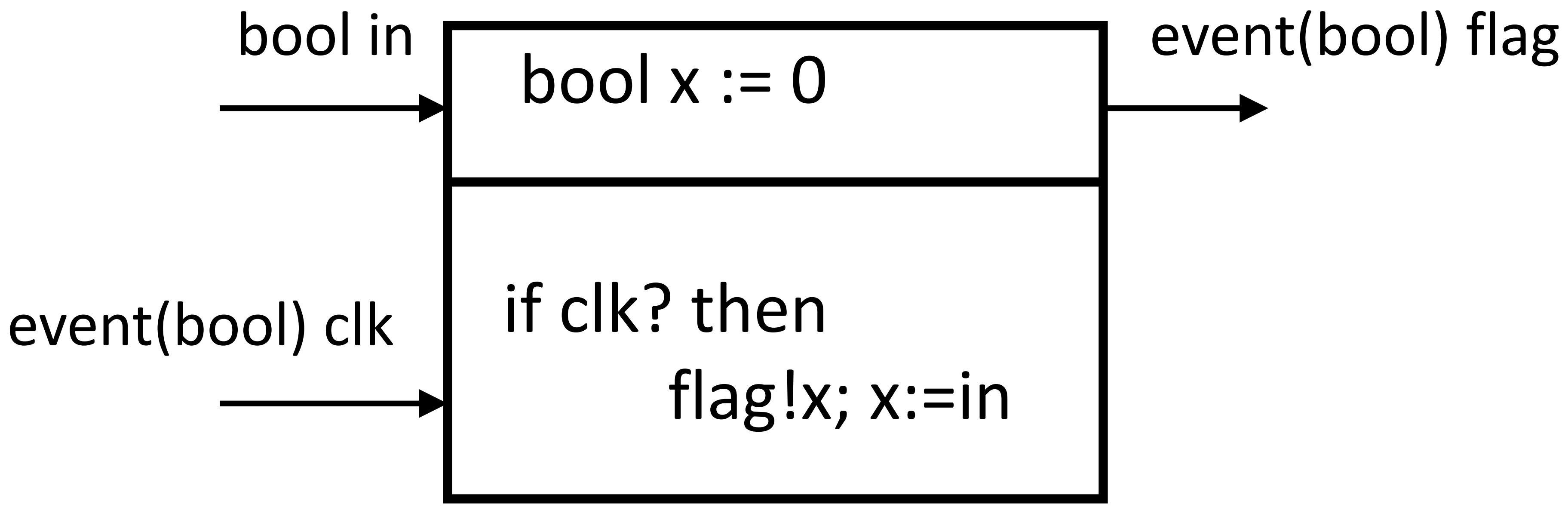


Event-triggered Components

- What to do if we want some components to *not* participate in some rounds?
- Event is a special input/output variable, which can be absent or present
- Event variable has value only if it is *present*
- Syntax:

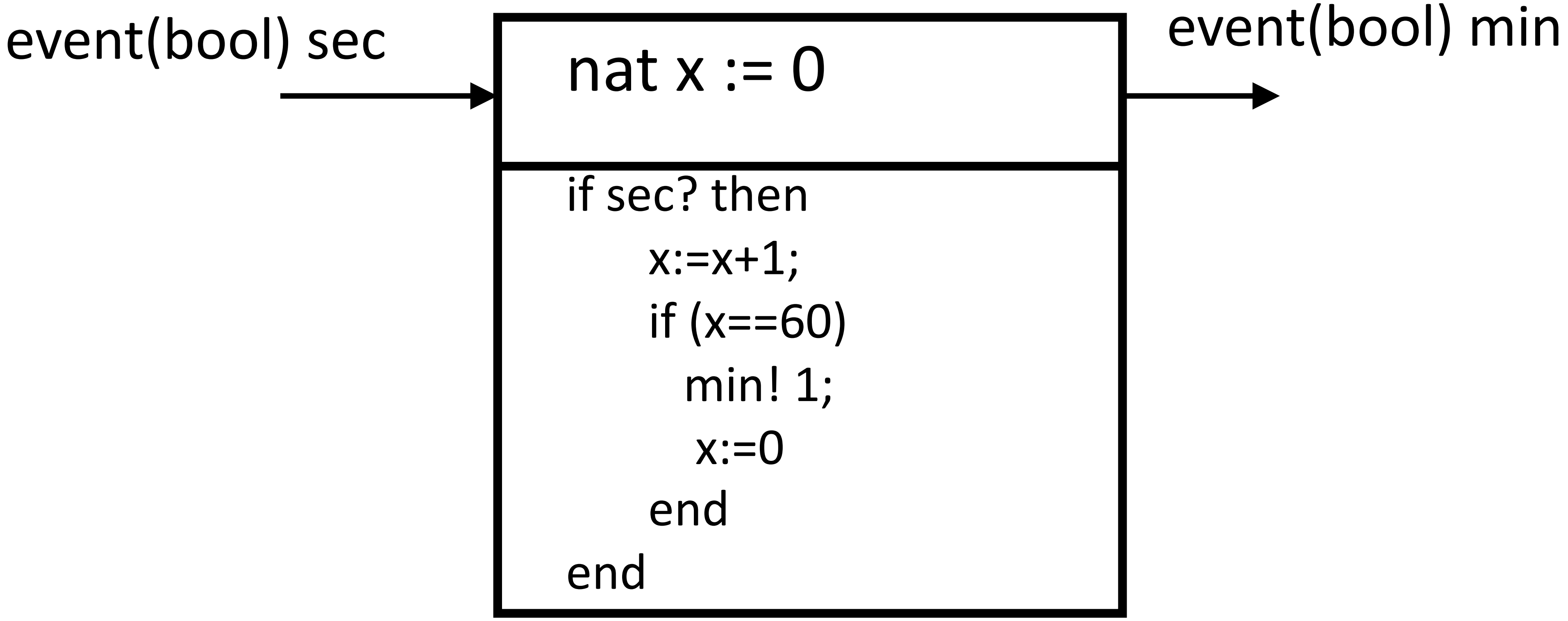
e?	True if e is present
e!a	e gets the value of the assignment a

Event-triggered Copy



Event-triggered Components

- No need to execute in a round where triggering events are absent

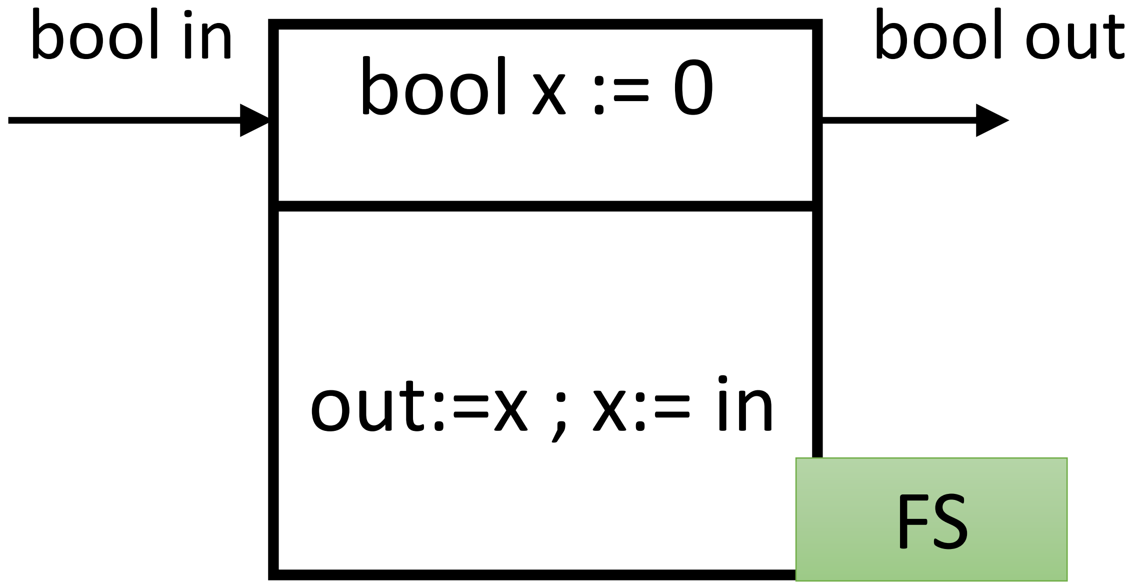


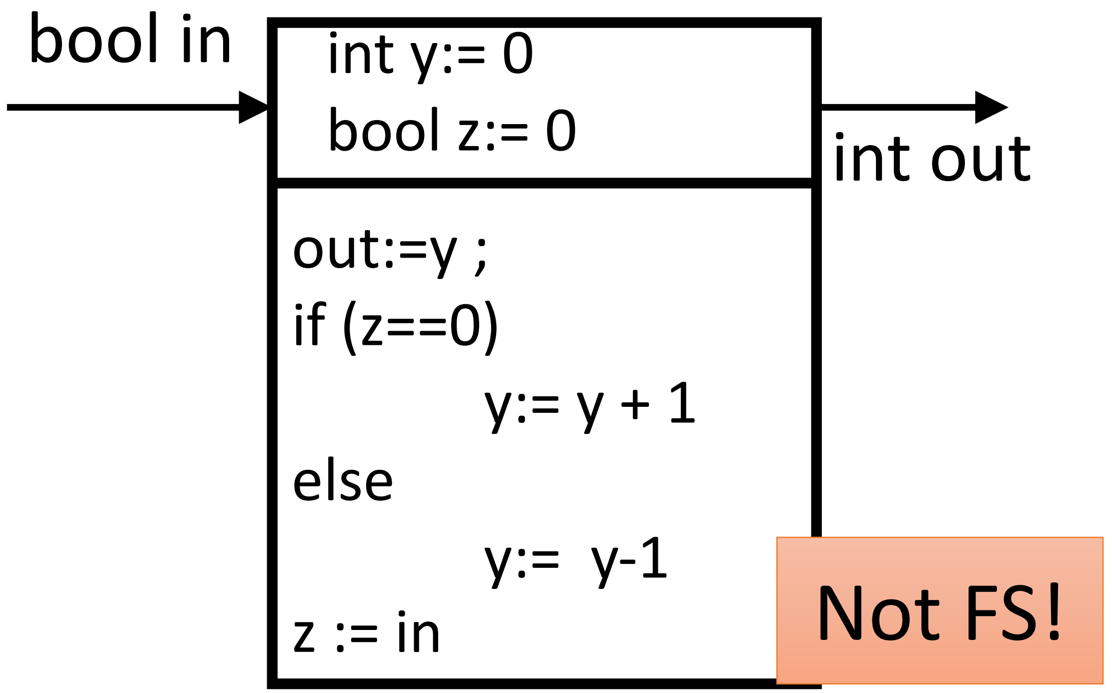
```
nat x := 0

if sec? then
      x:=x+1;
      if (x==60)
         min! 1;
          x:=0
      end
end
```

Finite-state Components

- Component is finite state if all variables are over finite types



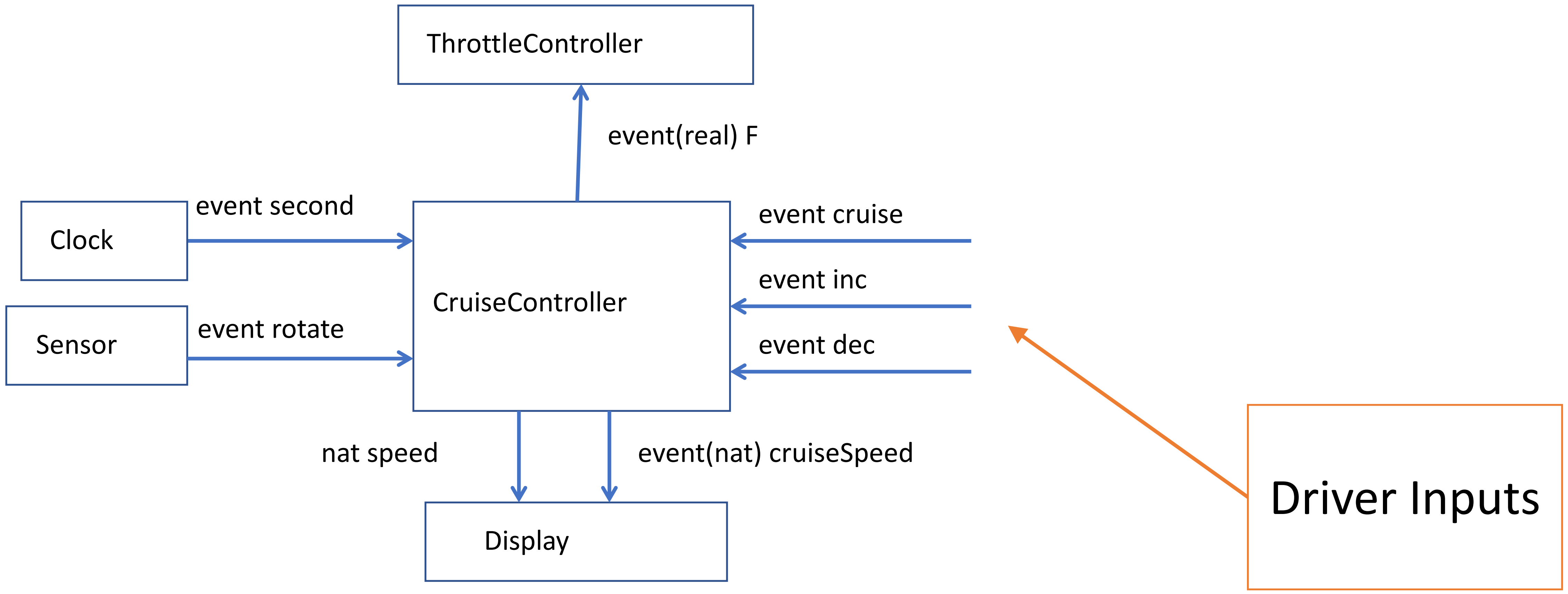


```
out:=y ;
if (z==0)
        y:= y + 1
else
        y:=  y-1
z := in
```

FSM Software Tools

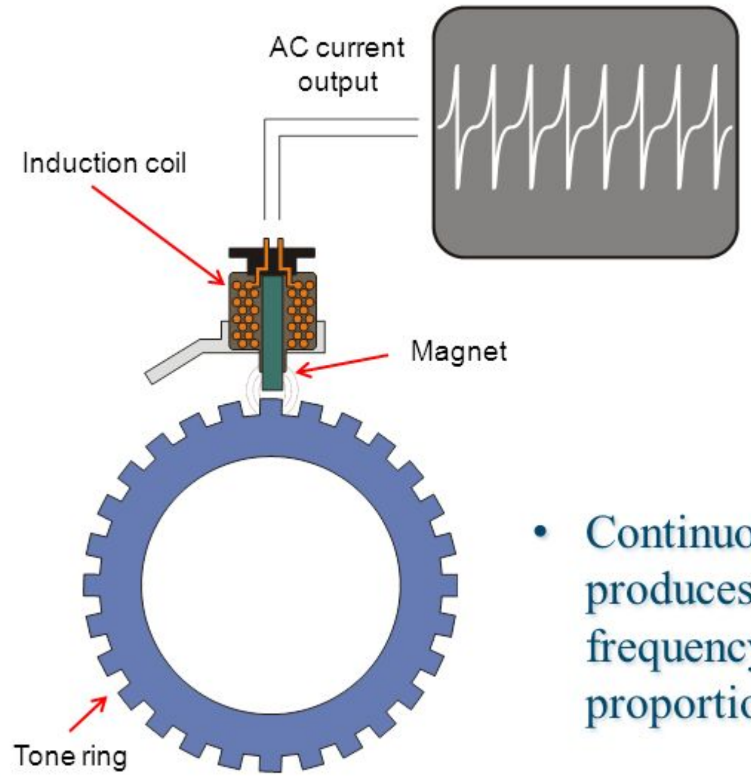
- Statecharts (Harel, 1987), a notation for concurrent com- position of hierarchical FSMs, has influenced many of these tools.
- One of the first tools supporting the Statecharts notation is STATEMATE (Harel et al., 1990), which subse- quently evolved into Rational Rhapsody, sold by IBM.
- Almost every software engineering tool that provides UML (unified modeling language) capabilities (Booch et al., 1998).
- SyncCharts (André́ , 1996) is a particularly nice variant in that it borrows the rigorous semantics of Esterel (Berry and Gonthier, 1992) for composition of concurrent FSMs.
- LabVIEW supports a variant of Statecharts that can operate within dataflow diagrams
- Simulink with its Stateflow extension supports a variant that can operate within continuous-time models.

Cruise Controller Example



Sensors

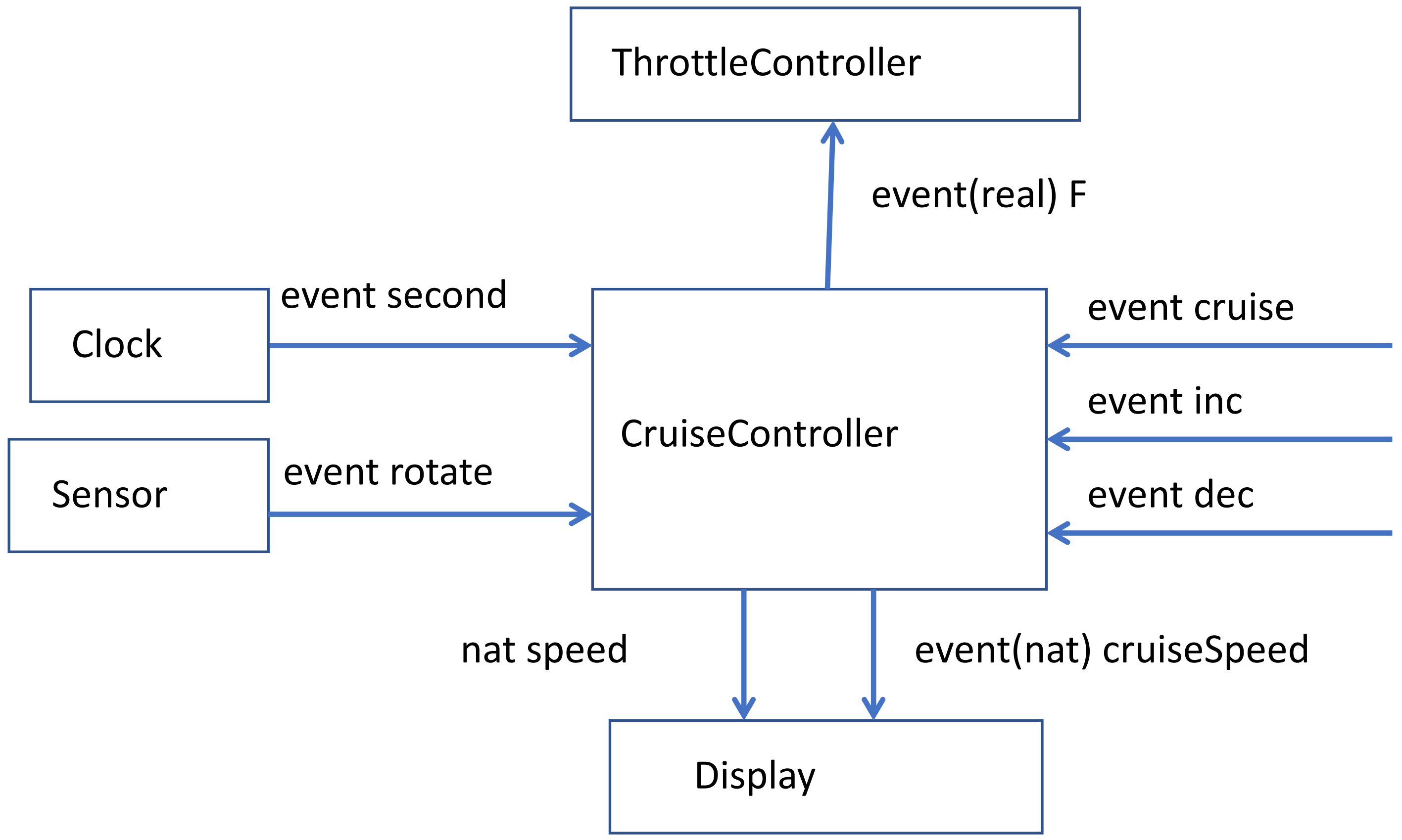
- Rotation Sensor: Wheel speed sensor or vehicle speed sensor
- Type of a tachometer
- Counts number of rotations per second and as the wheel radius is known, can compute the linear speed of the car



- The ABS wheel speed sensor generates a small electrical pulse whenever a tooth on the tone ring moves through the magnetic field of the pick up coil
- Continuous rotation of the tone ring produces an AC current whose frequency – measured in Hertz – is proportional to wheel speed

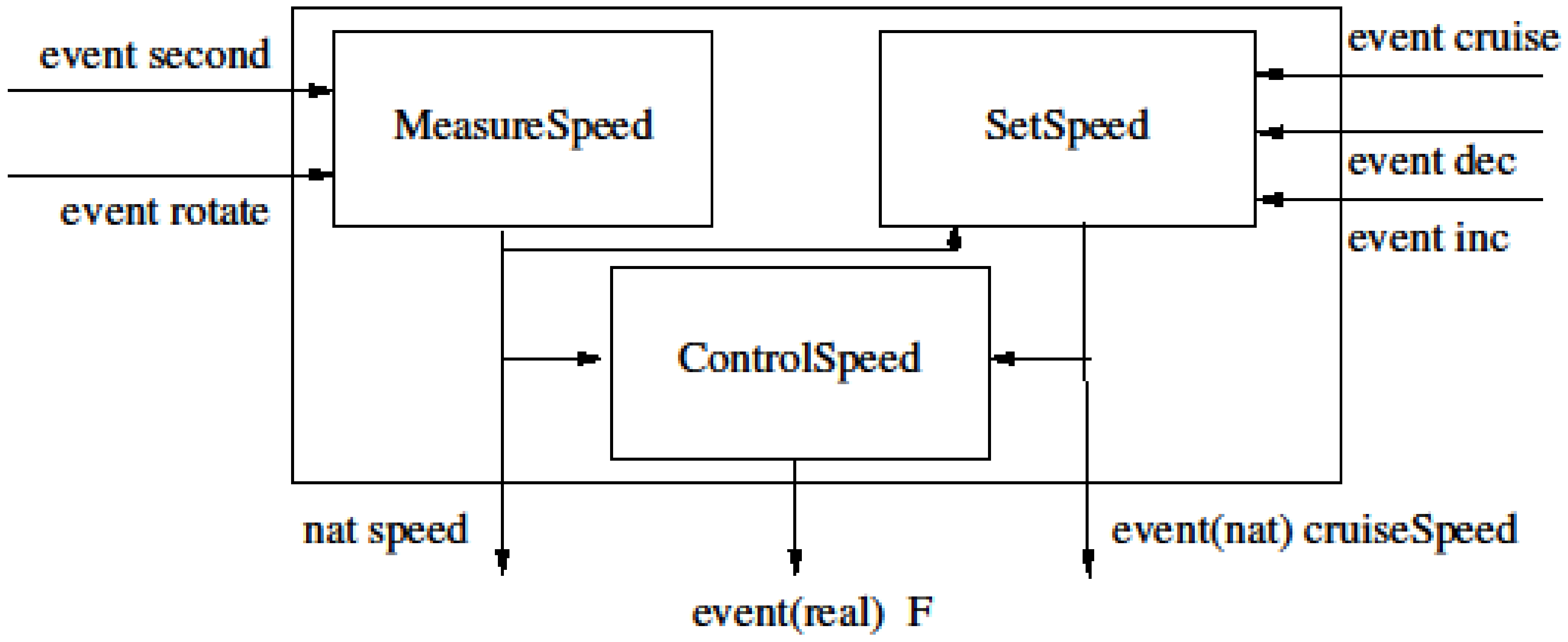
(From Porter and Chester Institute slides on Google Image Search)

Actuator

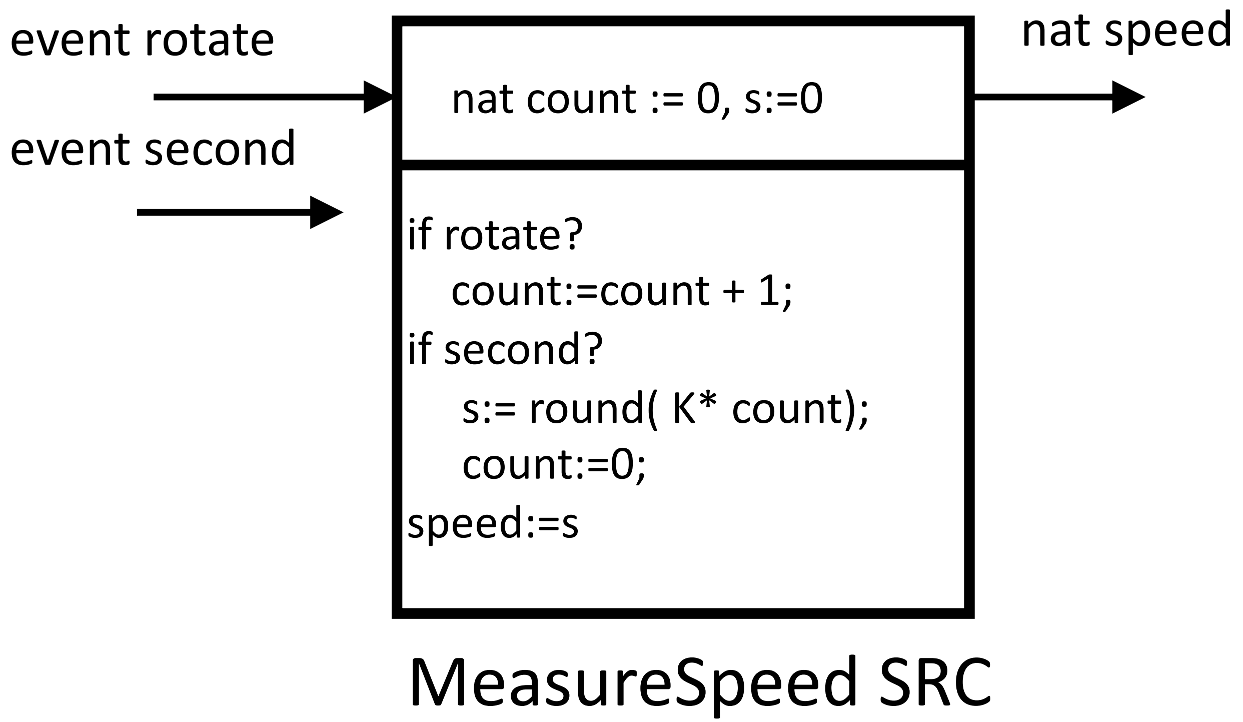


- ThrottleController is an actuator that gets a force/torque required to adjust the throttle plate which leads to tracking the desired speed

Decomposing CruiseController further



MeasureSpeed SRC



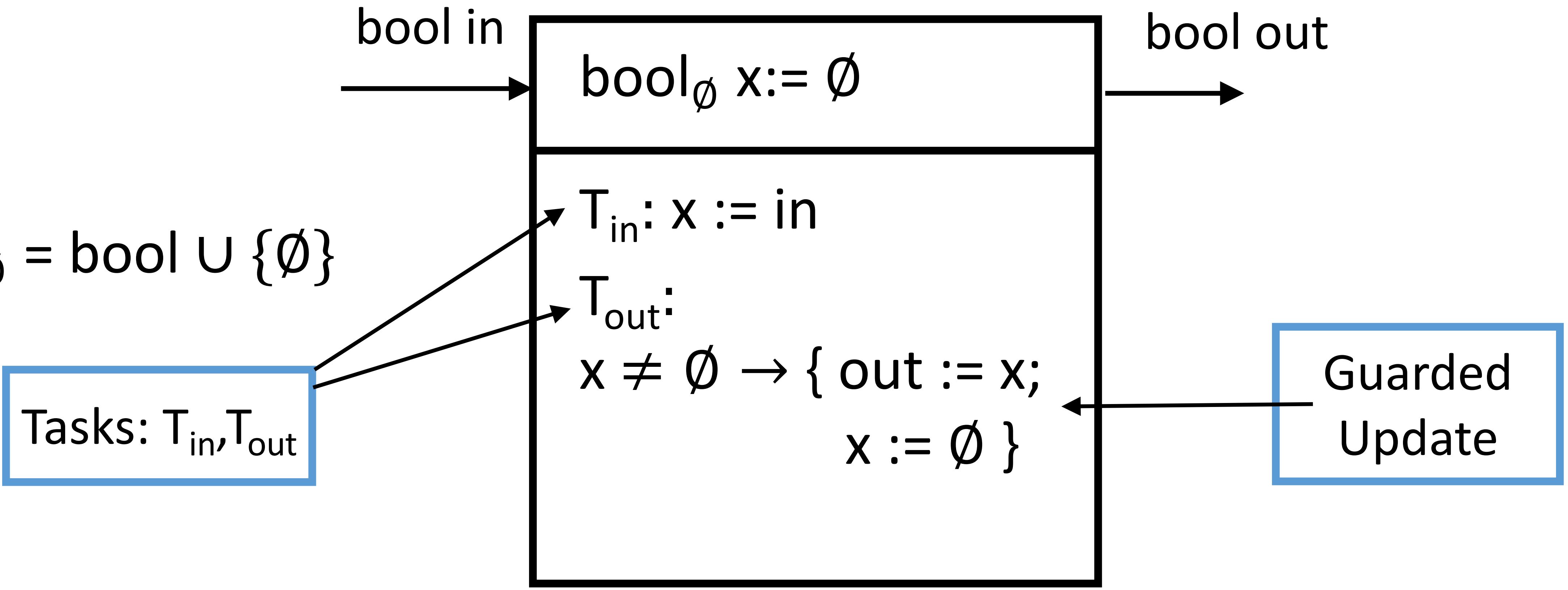
```
if rotate?
   count:=count + 1;
if second?
    s:= round( K* count);
    count:=0;
speed:=s
```

Asynchronous Components

Asynchrony

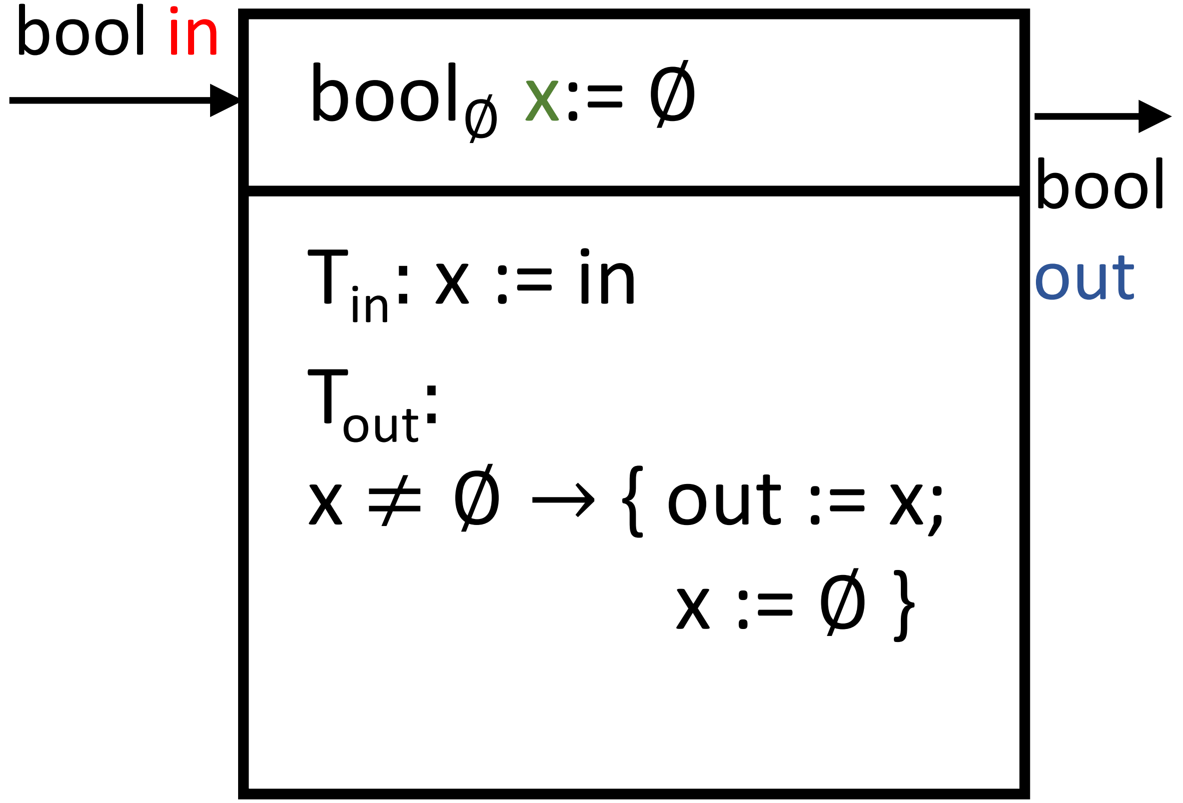
- Synchrony: All components execute in a sequence of rounds in lock-step
- Asynchrony: No lock-step computation!
- Natural model for networked, distributed communicating components executing independently and at possibly different speeds
- As there is no central, global clock, explicit coordination is required between components
- Examples:
  - Processes in distributed computation, multiple threads in any modern OS
  - Interrupt-driven processing

Asynchronous Reactive Component Example



$bool_\emptyset$ = bool ∪ {∅}

Asynchronous Reactive Component



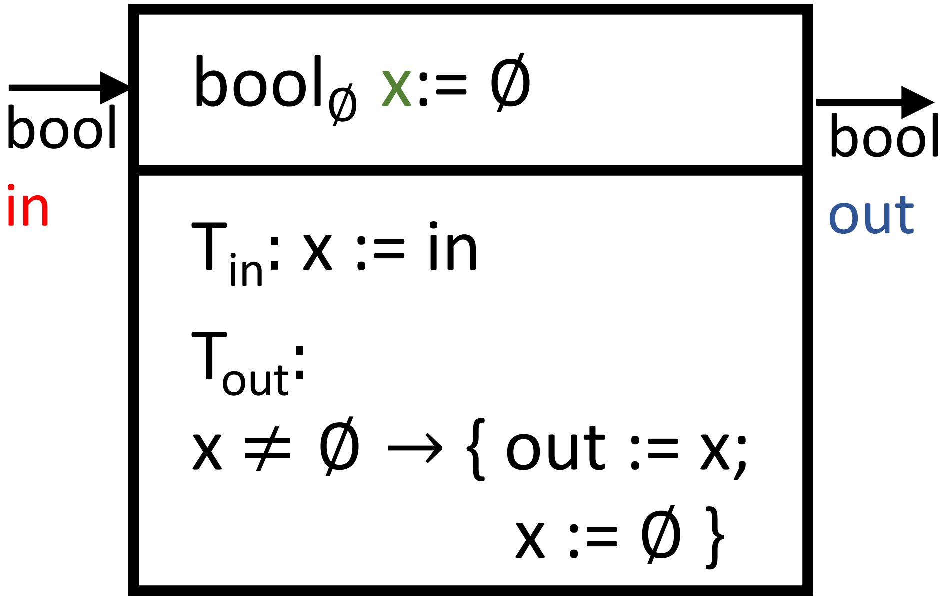
bool in
$\text{bool}_{\emptyset}$ x:= ∅
T_{in}: x := in
T_{out}:
$x \neq \emptyset \rightarrow$ { out := x;
x := ∅ }
bool
out

- Input channel in of type bool
- Output channel out of type bool
- State variable x of type bool+∅. The value ∅ indicates empty or null.
- x initialized to ∅
- Input task T_{in} reads input value into x
- Output task T_{out} produces output if x is not empty

Asynchronous Reactive Component Execution

- Execution Model: In each step only one task is executed
- Task can be executed only if it is enabled (i.e. if its guard condition is true)
- If multiple guard conditions are true, one task is nondeterministically executed
- Sample execution:

$$\emptyset \xrightarrow[T_{in}]{in?0} 0 \xrightarrow[T_{out}]{out!0} \emptyset \xrightarrow[T_{in}]{in?1} 1 \xrightarrow[T_{in}]{in?0} 0 \xrightarrow[T_{out}]{out!0} \emptyset$$



Buffer

Example: Asynchrony + Nondeterminism

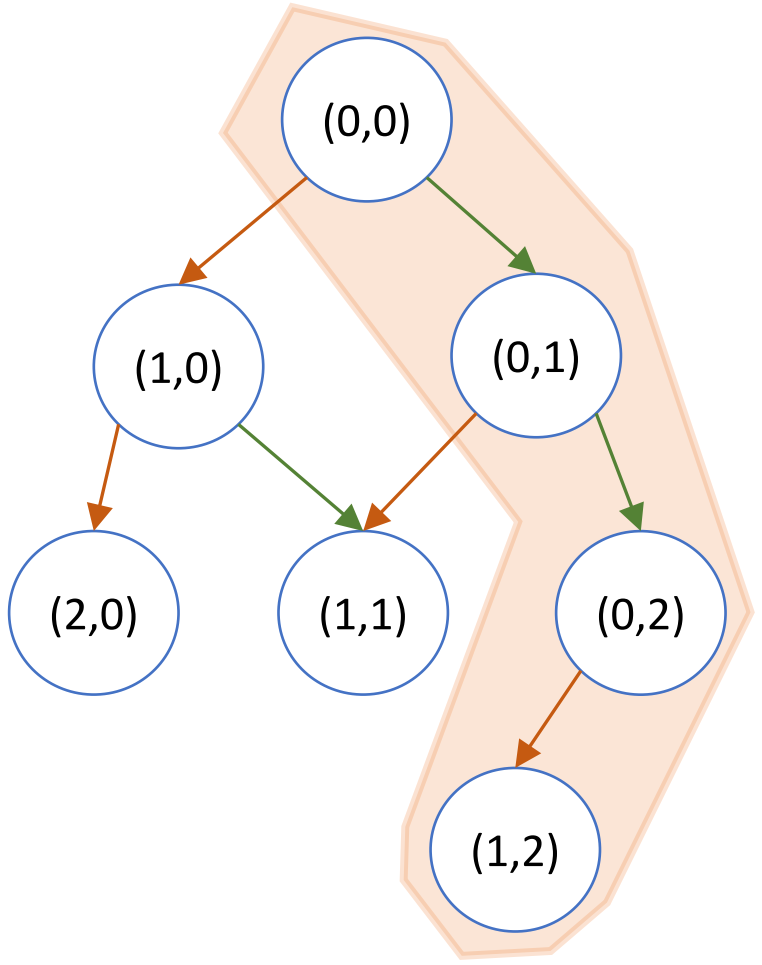
```
int x:= 0, y:= 0
```

```
Tx: x := x+1
Ty: y:= y+1
```

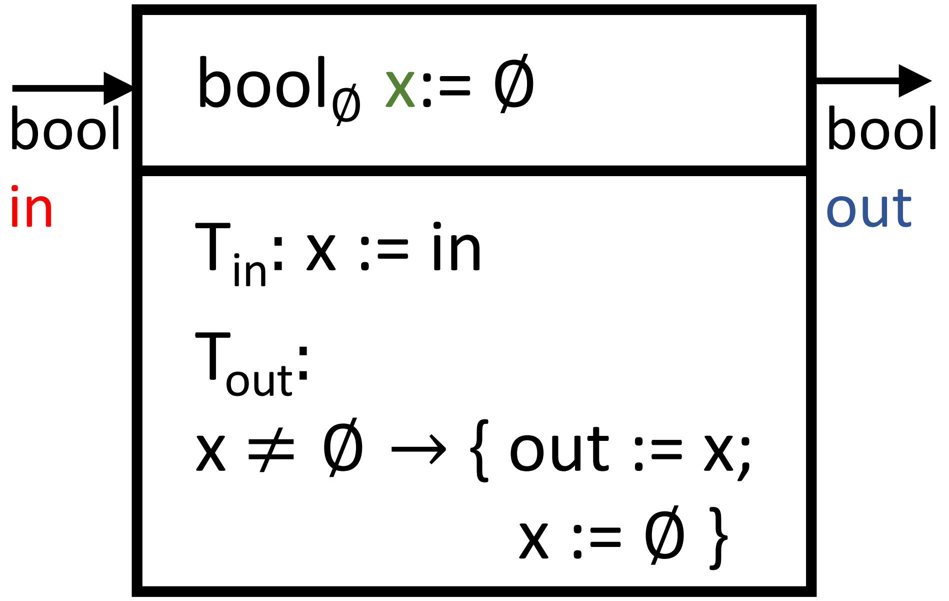
- ARC may have no inputs or outputs, just internal tasks
  - Update may have no guards
- In each step, execute T_x or T_y
- Sample execution:

$$(0,0) \underset{T_y}{\rightarrow} (0,1) \underset{T_y}{\rightarrow} (0,2) \underset{T_x}{\rightarrow} (1,2) \underset{T_y}{\rightarrow} (1,3)$$

- Interleaved model of concurrency



Asynchronous Process/Reactive Component



bool
in
$\text{bool}_{\emptyset}\ x := \emptyset$
$T_{in}: x := in$
$T_{out}:$
$x \neq \emptyset \rightarrow \{ out := x;$
$x := \emptyset \}$
bool
out

- Set of input channels: I
  - ESM representation: in?v, where v is the value to be received
- Set of output channels: O
  - ESM representation: out!v, where v is the value to be written
- Set of state variables X
- Initialization Init which maps state variables to initial values

Updates are different from SRCs!

Input Task defines updates of the form: **G → x:= E(X,in)**

- Guard condition G: some expression over ***only*** state variables X; input task can be executed only if G is true
- For each in in I, we associate a read-set (X ∪ {in}): variables that can appear in E for input task associated with in (rationale: can read value on in only if that task is enabled)
- Any state variable can appear in the LHS of the assignment
- Defines a set of input actions of the form: q $\xrightarrow{\text{in?v}}$ q'
  - where q is value of state variables before update, and q satisfies G
  - value of state variables after update is q' = E(X↦q, in↦v)

Updates are different from SRCs!

Output Task: defines updates of the form: **G → out := E(X)**

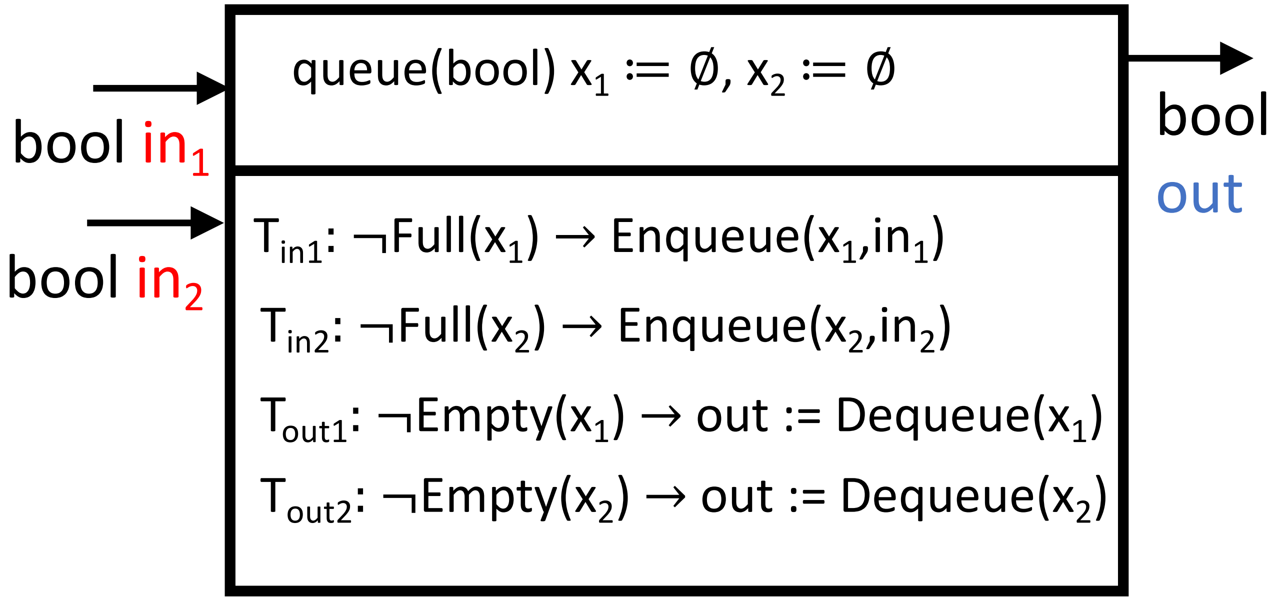
- Guard condition G: some expression over ***only*** variables in X; output task can be executed only if G is true
- For each out in O, we associate a write-set {out}: variables that appear on LHS of the assignment
- Any expression containing only state variables can appear in E
- Defines an output action of the form $q \xrightarrow{\mathrm{out}!v} q'$
  - where q is value of state variables before update, and q satisfies G
  - value of state variables after update is q'
  - value v is output on channel out

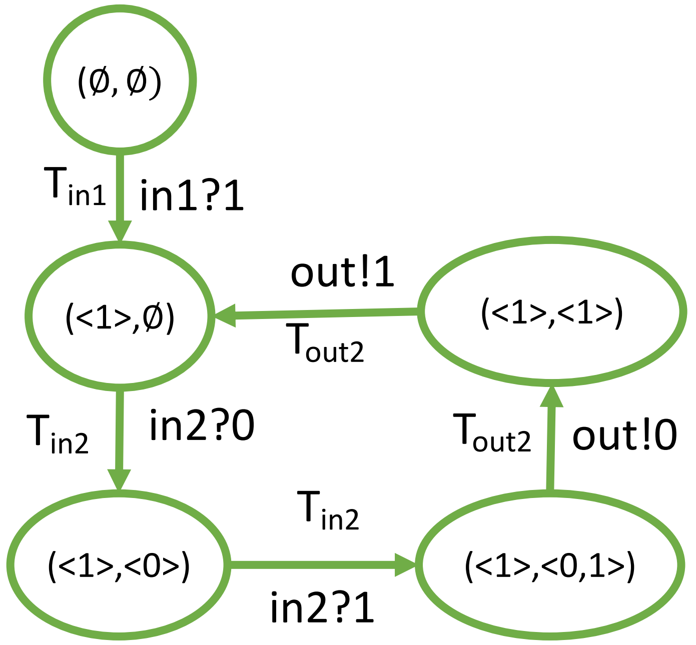
Updates are different from SRCs!

Internal Task: defines updates of the form: **G → x := E(X)**

- Guard condition G: some expression over ***only*** variables in X; internal task can be executed only if G is true
- Any expression containing only state variables can appear in E, only state variables appear on LHS
- Defines an internal action of the form $q \xrightarrow{\varepsilon} q'$
  - where q is value of state variables before update, and q satisfies G
  - value of state variables after update is q'
  - No input is read or output is produced!

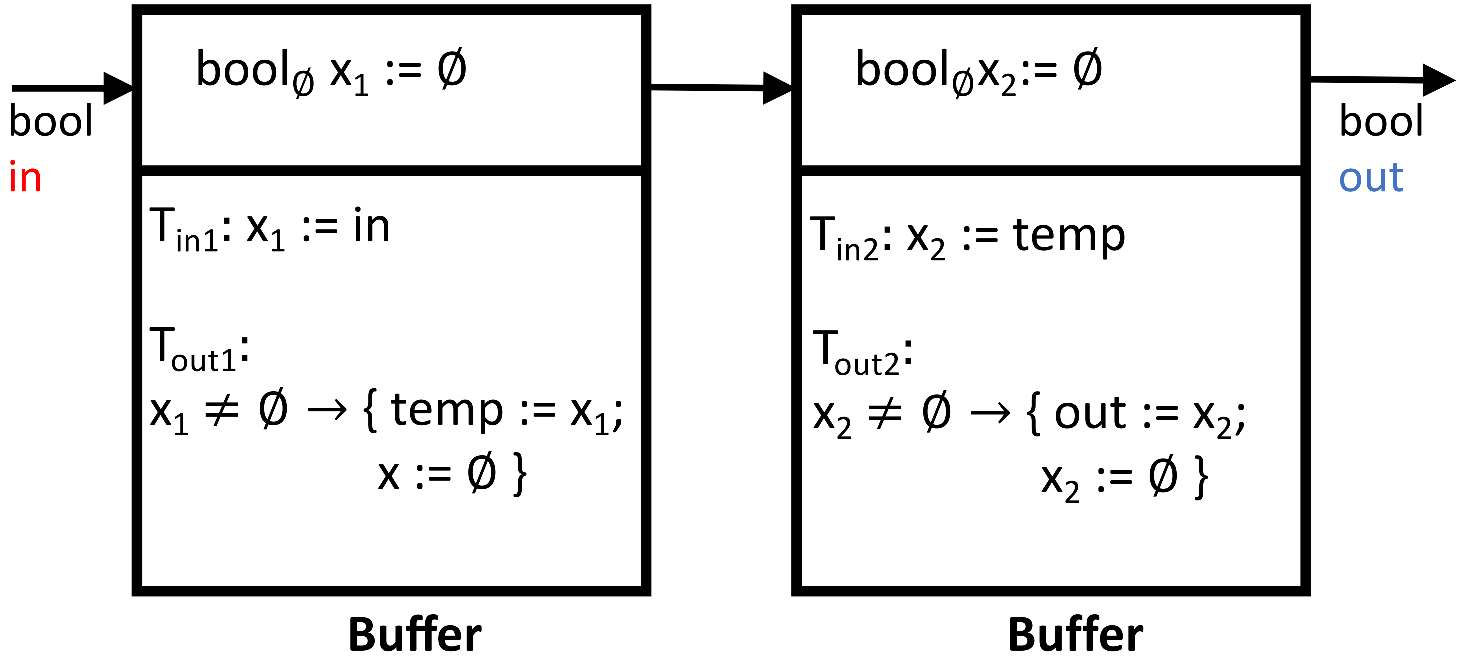
Asynchronous Merge: Sequence of Actions


bool in$_1$
bool in$_2$
bool out
queue(bool) $x_1 := \emptyset$, $x_2 := \emptyset$
T_{in1}: $\neg$Full(x_1) $\rightarrow$ Enqueue(x_1,in$_1$)
T_{in2}: $\neg$Full(x_2) $\rightarrow$ Enqueue(x_2,in$_2$)
T_{out1}: $\neg$Empty(x_1) $\rightarrow$ out := Dequeue(x_1)
T_{out2}: $\neg$Empty(x_2) $\rightarrow$ out := Dequeue(x_2)



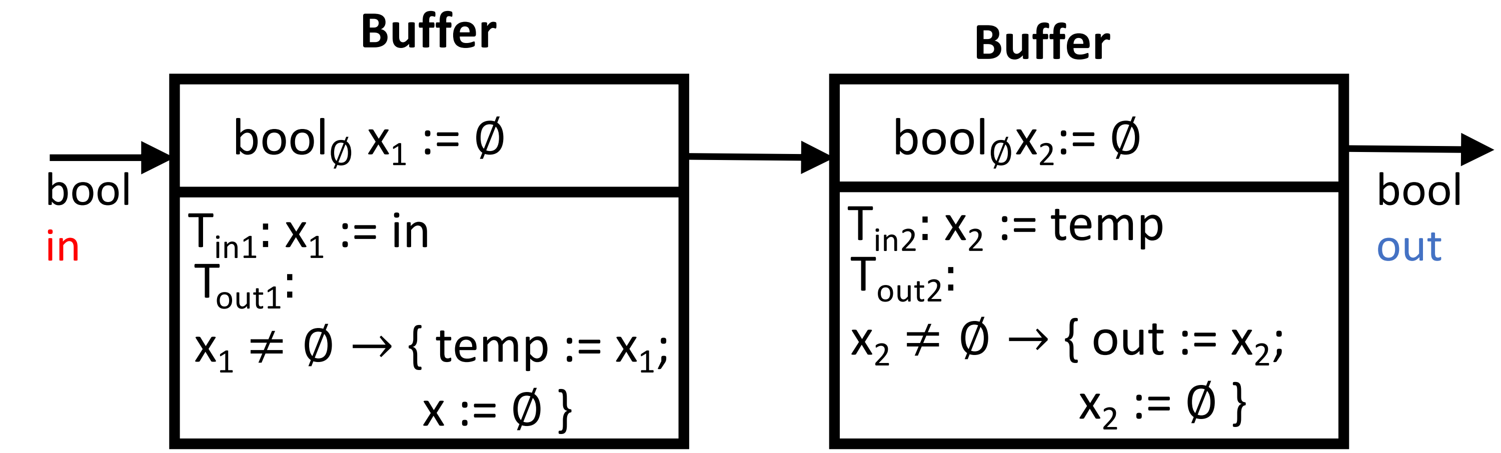
Asynchronous Processes can also be represented with extended state machines

Composing Asynchronous Processes


bool
in
$\text{bool}_{\emptyset}\ x_1 := \emptyset$
$T_{in1}: x_1 := in$
$T_{out1}:$
$x_1 \neq \emptyset \rightarrow \{\ temp := x_1;$
$x := \emptyset\ \}$
Buffer
$\text{bool}_{\emptyset} x_2 := \emptyset$
$T_{in2}: x_2 := temp$
$T_{out2}:$
$x_2 \neq \emptyset \rightarrow \{\ out := x_2;$
$x_2 := \emptyset\ \}$
Buffer
bool
out

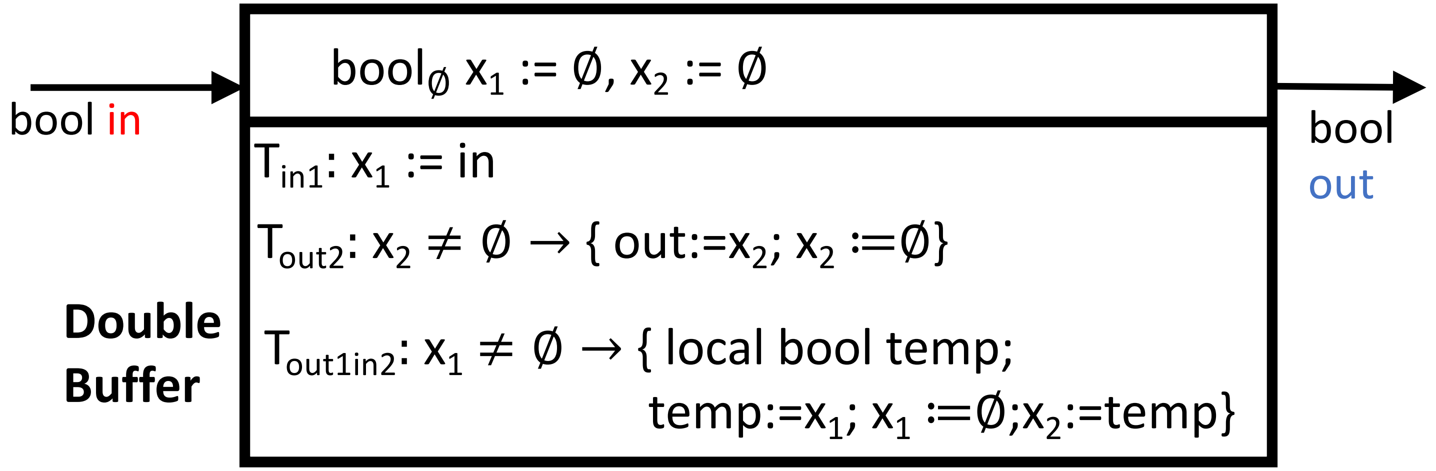
- Parallel composition: Inputs, Outputs, States and Initialization similar to the synchronous case
- Input consumption needs to be synchronized with output production for the 'temp' variable

Composed DoubleBuffer



```
$bool_\emptyset\ x_1 := \emptyset$
$T_{in1}$: $x_1 := in$
$T_{out1}$:
$x_1 \neq \emptyset \rightarrow \{$ temp := $x_1$;
        $x := \emptyset$ }
```

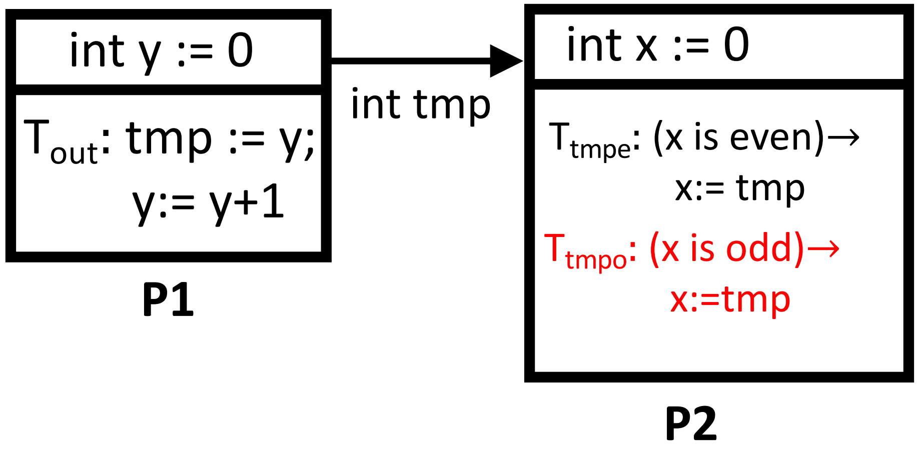
```
$bool_\emptyset x_2 := \emptyset$
$T_{in2}$: $x_2$ := temp
$T_{out2}$:
$x_2 \neq \emptyset \rightarrow \{$ out := $x_2$;
        $x_2 := \emptyset$ }
```



```
$bool_\emptyset\ x_1 := \emptyset,\ x_2 := \emptyset$
$T_{in1}$: $x_1 := in$
$T_{out2}$: $x_2 \neq \emptyset \rightarrow \{$ out:=$x_2$; $x_2 := \emptyset\}$
$T_{out1in2}$: $x_1 \neq \emptyset \rightarrow \{$ local bool temp;
        temp:=$x_1$; $x_1 := \emptyset$;$x_2$:=temp}
```

- Defining $P_1 \mid\mid P_2$
- In each step only 1 task executes
- If y is an output channel of P_1 and input channel of P_2:
- A_1: output task for P_1 with code: $G_1 \rightarrow U_1$
- A_2: input task for P_2 with code: $G_2 \rightarrow U_2$
- Composition has output task for y with code: $G_1 \wedge G_2 \rightarrow U_1;U_2$

Blocking vs. Non-blocking Synchronization

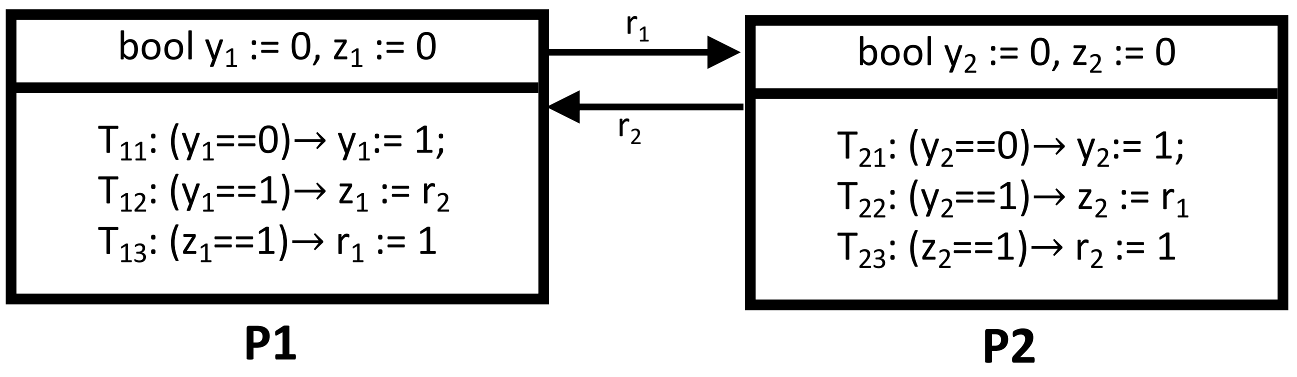


How do you make P2 non-blocking?

- Task T_{out} of P1 can *produce* a value on the output only if P2 has an input task that is enabled to *consume* the value with some input task
- In this example, once x becomes odd, P2 cannot consume (no enabled input task) and it **blocks** communication
- Process is **non-blocking** on channel in if at least one guarded update corresponding to input task for in is enabled
- Process is **non-blocking** if for every input channel, the disjunction of all guards corresponding to input tasks for that channel is *valid* or the Boolean formula 1 (true).

Deadlocks

- Common error in asynchronous designs
- Caused by each process waiting for another process to execute a task, but no task is enabled



$\text{bool } y_1 := 0,\ z_1 := 0$

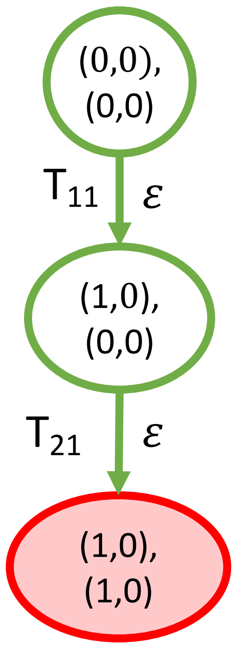
$T_{11}: (y_1{==}0) \rightarrow y_1 := 1;$
$T_{12}: (y_1{==}1) \rightarrow z_1 := r_2$
$T_{13}: (z_1{==}1) \rightarrow r_1 := 1$

P1

$\text{bool } y_2 := 0,\ z_2 := 0$

$T_{21}: (y_2{==}0) \rightarrow y_2 := 1;$
$T_{22}: (y_2{==}1) \rightarrow z_2 := r_1$
$T_{23}: (z_2{==}1) \rightarrow r_2 := 1$

P2



$$\begin{tikzcd}
(0,0),(0,0) \arrow[d, "T_{11}"', "\varepsilon"] \\
(1,0),(0,0) \arrow[d, "T_{21}"', "\varepsilon"] \\
(1,0),(1,0)
\end{tikzcd}$$