Computational Models for Embedded Systems

Vescan Andreea, PHD, Assoc. Prof.

Faculty of Mathematics and Computer Science
Babeș-Bolyai University
Cluj-Napoca

2025-2026

Lecture 4b: Asynchronous Reactive Models

Software Systems Verification and Validation

“Tell me and I forget, teach me and I may remember, involve me and I learn.”

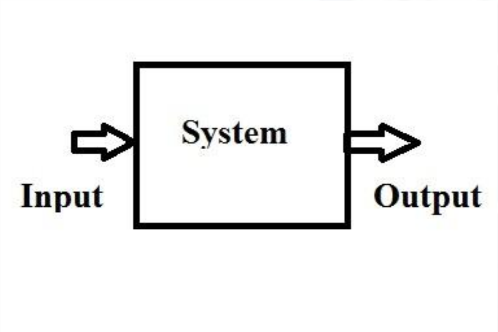
(Benjamin Franklin)

Outline

- Asynchronous Models - Asynchronous Processes
  - States, Inputs, and Outputs
  - Input, Output, and Internal Actions
  - Executions
  - Operations on Processes
- Asynchronous Design Primitives
  - Blocking vs. Non-blocking Synchronization
  - Deadlocks
  - Shared Memory
  - Fairness
- Asynchronous Coordination Protocols
  - Leader Election

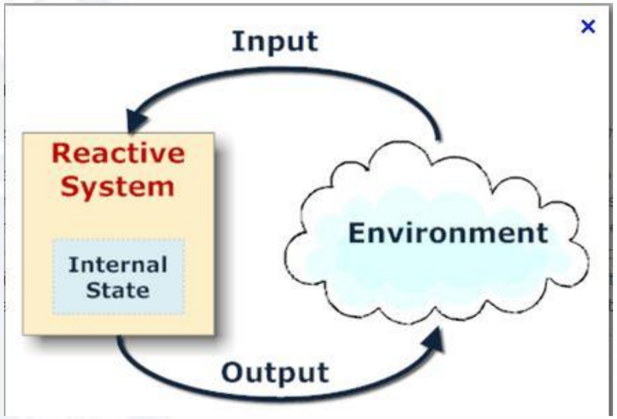
Functional vs. Reactive

Functional component



- produces outputs when supplied with inputs
- its behavior can be mathematically described using a mapping between input values and output values.

Reactive component



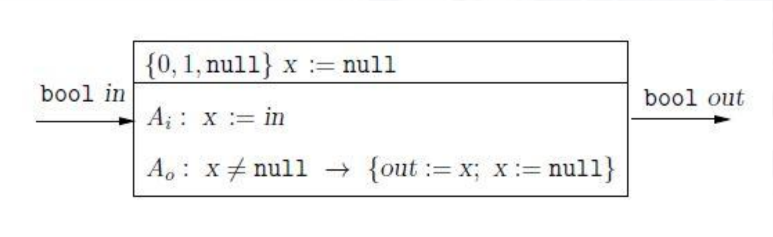
ate

- interacts with other components via inputs and outputs in an ongoing manner

Asynchronous Models- Asynchronous Processes

States, Inputs, and Outputs

- The input and output variables of a process are called channels.
- when there are multiple input channels, the arrival of input values on different channels is not synchronized.
- x ? v = can be interpreted as receiving the value v on the input channel x.
- When there are multiple output channels, in one step, a process can produce a value for only one of the output channels.
- y ! v= can be interpreted as sending the value v on the output channel y.
-



$\{0, 1, \texttt{null}\}\ x := \texttt{null}$

$A_i :\ x := in$

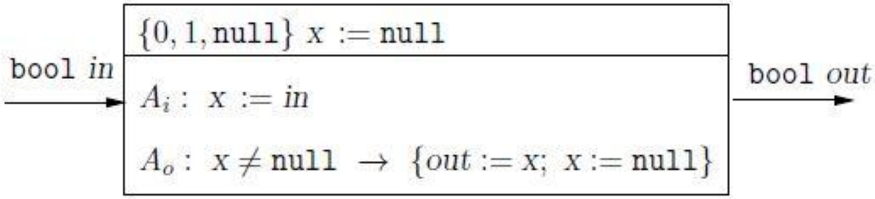
$A_o :\ x \neq \texttt{null} \rightarrow \{out := x;\ x := \texttt{null}\}$

S={x}, I={in}, O={out}
set of states={0, 1, null}
set of initial states = {null}
set of inputs = {in?0, in?1}
set of outputs = {out!0, out!1}

Buffer

Asynchronous Processes

Asynchronous Models



bool in

$\{0, 1, \texttt{null}\}$ $\texttt{x} := \texttt{null}$

A_i : $\texttt{x} := in$

A_o : $\texttt{x} \neq \texttt{null} \rightarrow \{out := \texttt{x};\ \texttt{x} := \texttt{null}\}$

bool out

- Asynchronous process Buffer

- The input and output variables of a process are called channels.
- The internal state of the process Buffer is a buffer of size 1, which can either be empty, or contain a Boolean value.
- The process Buffer has two possible types of actions.
  - It can process an input value available in the input channel in by copying it into its buffer.
  - Alternatively, if the buffer is non-empty, the process can output the buffer state by writing it to the output channel out, and then reset the buffer to empty. Each type of action is specified using a task, and in one step only one of the tasks is executed.

Asynchronous Models- Asynchronous Processes

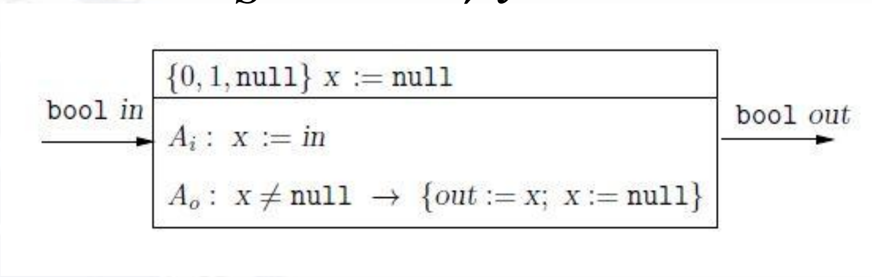
Input, Output, and Internal Actions

- For asynchronous processes also we specify its computation using a set of tasks.
- In contrast to the synchronous case, during one step, instead of executing all the tasks, only one task is executed.
- To indicate whether a task is ready to be executed, we explicitly associate a **guard condition** with each task.
  - This condition is given as a Boolean formula over state variables, and the task is enabled in a given state if the state satisfies this formula.
  - If multiple tasks are enabled, then one of them is chosen non deterministically for execution.
- To assure that a process either receives a single input value or sends a single output value in a step**, we require that each task can either read at most one input channel or write at most one output channel.**

Asynchronous Models- Asynchronous Processes

Input tasks

$$s \xrightarrow{x?v} t$$



bool *in*

$\{0,1,\texttt{null}\}\ \texttt{x} := \texttt{null}$

$A_i:\ \texttt{x} := in$

$A_o:\ \texttt{x} \neq \texttt{null} \rightarrow \{out := \texttt{x};\ \texttt{x} := \texttt{null}\}$

bool *out*

- **Input action** = processing of an input
  - The process
    - can only update its state
    - Does not produce outputs
  - is specified using input task associated with one input channel

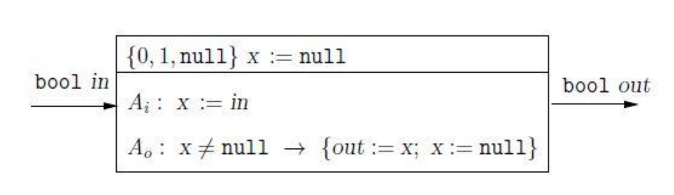
- The description of an input task A is given as Guard→Update
  - Guard = gives the condition under which this task is willing to process inputs on the channel x
  - Update=described how the task updates state variables based on the old values of the state variables together with input value received on the channel x.

Asynchronous Models-Asynchronous Processes

Output tasks

$$s \xrightarrow{y\,!v} t$$

- **Output action** = producing an output
  - The process can update its state.
  - Is specified using output task associated with one output channel.


bool in
$\{0, 1, \text{null}\}\ x := \text{null}$
$A_i:\ x := in$
$A_o:\ x \neq \text{null} \rightarrow \{out := x;\ x := \text{null}\}$
bool out

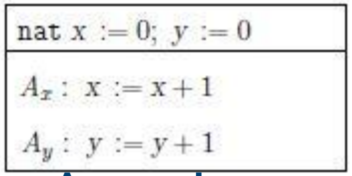
  - Is described using
    - a guard condition Guard= specifies the set of states in which the output task is ready to be executed
    - An update description Update = specifies how the task updates state variables and the output value based on the values of the state variables it reads.

Asynchronous Models- Asynchronous Processes

$$s \xrightarrow{\varepsilon} t$$

label indicates that there is no observable communication during an internal action.

Internal tasks



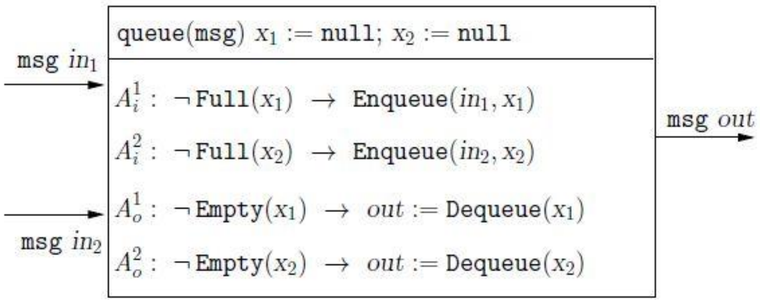
```
nat x := 0; y := 0
A_x : x := x + 1
A_y : y := y + 1
```

AsyncInc

- **internal action** = describe internal computation of a process
  - Such actions neither process inputs nor produce outputs, but update internal state.
  - is described using internal task that has
    - Guard =describes the states in which the task is enabled
    - Update=specifies how the task updates state variables based on their old values.
  - Set of all internal task A={A_x, A_y}.
  - A step of the process corresponds to executing one of these two tasks.
  - The set of internal actions consists of $(i, j) \xrightarrow{\varepsilon} (i+1, j)$ and $(i, j) \xrightarrow{\varepsilon} (i, j+1)$ for every pair of natural numbers i and j.

Asynchronous Models- Asynchronous Processes

Internal tasks - Asynchronous merge



queue(msg) x_1 := null; x_2 := null

A_i^1 : $\neg$Full(x_1) → Enqueue(in_1, x_1)

A_i^2 : $\neg$Full(x_2) → Enqueue(in_2, x_2)

A_o^1 : $\neg$Empty(x_1) → out := Dequeue(x_1)

A_o^2 : $\neg$Empty(x_2) → out := Dequeue(x_2)

Asynchronous process Merge

- the process uses a buffer dedicated to each of the input channels to store values received on that channel.
- we model buffer using the type queue.
- Compared to the process Buffer, this captures a different style of synchronization: the environment, or the process sending values on the channel in_1, is blocked, if the process Merge has its internal queue x_1 full.
- A_{out} = {$A^1{}_o$, $A^2{}_o$ }. When both the queues x_1 and x_2 are non-empty, both the output tasks are enabled and either of them can be executed.

Asynchronous Models Asynchronous Processes

Asynchronous Process

- An asynchronous process P has

  (1) a finite set I of typed input channels defining the set of inputs of the form x ? v with $x \in I$ and a value v for x;

  (2) a finite set O of typed output channels defining the set of outputs of the form y ! v with $y \in O$ and a value v for y;

  (3) a finite set S of typed state variables defining the set Q_S of states;

  (4) an initialization Init defining the set [[Init]] $\subseteq$ Q_S of initial states;

  (5) for each input channel x, a set A_x of input tasks, each described by a guard condition over S and $S \cup \{x\}$ an update from the read-set $S \cup \{y\}$ to the write-set S defining a set of input actions of the form $s \xrightarrow{x?v} t$

  (6) for each output channel y, a set A_y of output tasks, each described by a guard condition over S and an update from the read-set S to the write-set

  defining a set of output actions of the form $s \xrightarrow{y!v} t$

- (7) a set A of internal tasks, each described by a guard condition over S and an update from the read-set S to the write-set S defining a set of internal actions of the form $s \xrightarrow{\varepsilon} t$

Asynchronous Models

Asynchronous Processes

Executions

- **Executions** = captures the operational semantics of a process.
- execution in an initial state, at every step one of the task is enabled and executed; the order in which different tasks are executed is totally unconstrained. **Such a semantics for asynchronous interaction is called the interleaving semantics.**
- A finite **execution** of an asynchronous process P consists of a finite sequence of the form

$$s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} s_3 \ldots s_{k-1} \xrightarrow{l_k} s_k$$

where for 0≤j≤k, each s_j is a state of P, s_0 is an initial state of P, and for 1≤j≤k,

$$s_{j-1} \xrightarrow{l_j} s_j$$

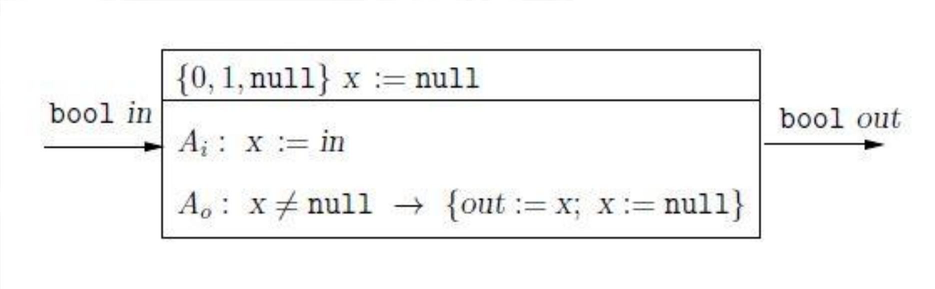
is either an input, or an output, or an internal action P.

Asynchronous Models

Asynchronous Processes

Executions (cont)

- Asynchronous process Buffer



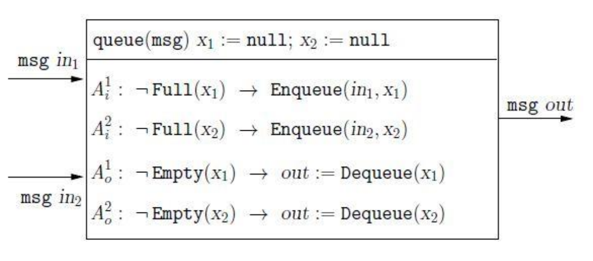
$\{0, 1, \mathtt{null}\}\ \mathtt{x} := \mathtt{null}$

$A_i :\ \mathtt{x} := in$

$A_o :\ \mathtt{x} \neq \mathtt{null} \rightarrow \{out := \mathtt{x};\ \mathtt{x} := \mathtt{null}\}$

$$null \xrightarrow{in?1} 1 \xrightarrow{out!1} null \xrightarrow{in?0} 0 \xrightarrow{in?1} 1 \xrightarrow{in?1} 1 \xrightarrow{out!1} null$$

- Asynchronous process Merge



$\mathtt{queue(msg)}\ x_1 := \mathtt{null};\ x_2 := \mathtt{null}$

$A_i^1 :\ \neg\mathtt{Full}(x_1) \rightarrow \mathtt{Enqueue}(in_1, x_1)$

$A_i^2 :\ \neg\mathtt{Full}(x_2) \rightarrow \mathtt{Enqueue}(in_2, x_2)$

$A_o^1 :\ \neg\mathtt{Empty}(x_1) \rightarrow out := \mathtt{Dequeue}(x_1)$

$A_o^2 :\ \neg\mathtt{Empty}(x_2) \rightarrow out := \mathtt{Dequeue}(x_2)$

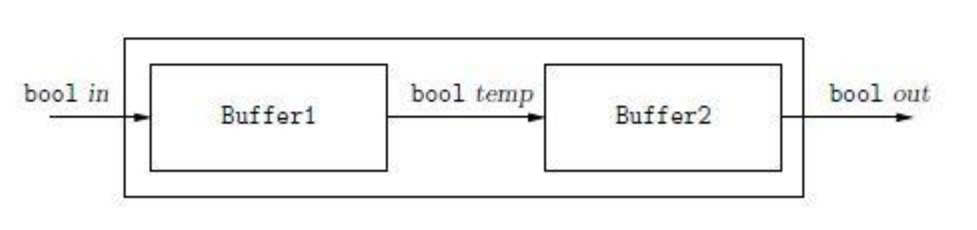
$$(null, null) \xrightarrow{in_1?0} (0, null) \xrightarrow{in_1?2} (02, null) \xrightarrow{in_2?5} (02,5) \xrightarrow{out!5}$$

$$(02, null) \xrightarrow{in_2?3} (02,3) \xrightarrow{out!0} (2,3) \xrightarrow{out!3} (2, null) \xrightarrow{in_1?0} (20, null)$$

Asynchronous Models- Asynchronous Processes

Operations on Processes

- Input/output channel renaming



$$[Buffer[out \mapsto temp]] | Buffer[in \mapsto temp] \setminus temp$$

DoubleBuffer Block diagram

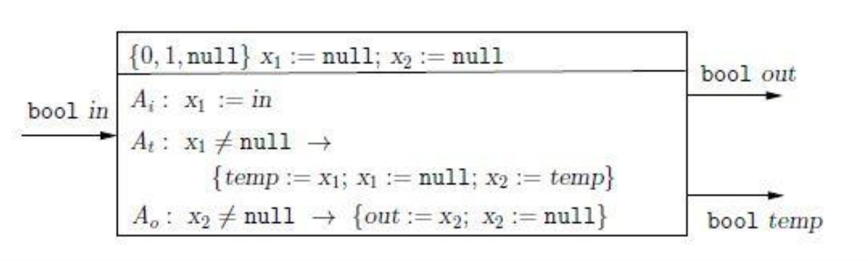
Asynchronous Models- Asynchronous Processes

Operations on Processes

- Parallel composition
  - two processes can be composed only if their variable declarations are mutually consistent: there are no name conflicts concerning state variables, and the two sets of output channels are disjoint.
    - ➔These requirements capture the assumption that only one process is responsible for controlling the value of any given variable.

Composition cases:

- **If an input channel x is common to both processes**
  - the task A12 for processing input values on the channel x is enabled exactly when both the corresponding input tasks of the two component processes are enabled, and it updates the state variables of P1 using the update code Update1, and then updates the state variables of P2 by executing the update code Update2. The order in which the two update descriptions are executed does not really matter as they update disjoint sets of variables.
- **If a channel x is an output channel of one process, say process P1, and an input channel of the other process P2**
  - the two processes synchronize using this channel: when P1 executes an output action sending a value on channel x, the receiver P2 executes a matching input action. The resulting joint action is an output action for the composite.
- **P1 has an input channel x that is not a channel of the other process P2.**
  - to process an input value on the channel x, the composite process simply executes the input task of P1 corresponding to the channel x, and the state of P2 stays unchanged during such an input action
  - Same holds for **output action**s for a channel that in
- **If an internal action** of the composite process is an intern the process maintaining its state unchanged. Thus, every inte composite with the same guard condition and update de



$\{0,1,\mathtt{null}\}\ x_1 := \mathtt{null};\ x_2 := \mathtt{null}$

$A_i:\ x_1 := in$

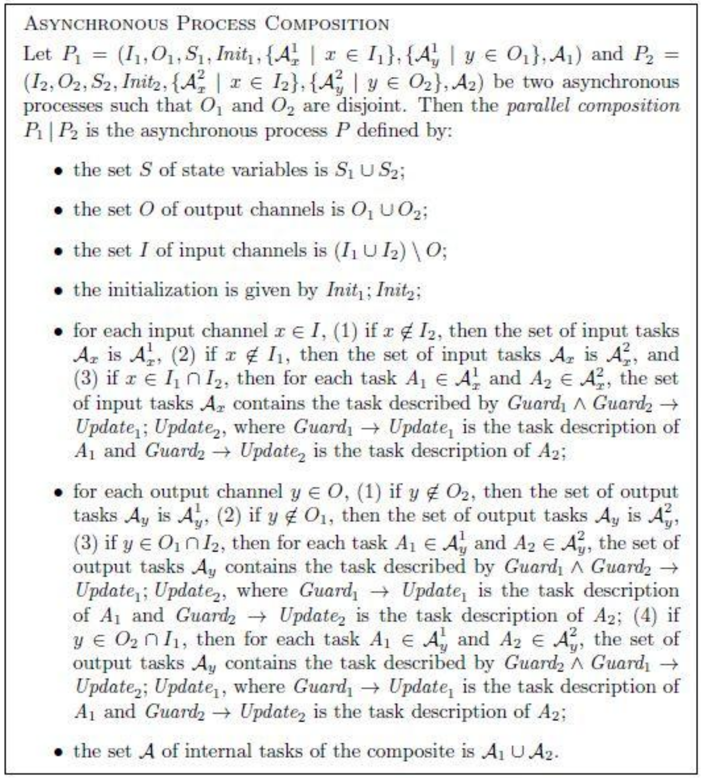
$A_t:\ x_1 \neq \mathtt{null} \rightarrow \{temp := x_1;\ x_1 := \mathtt{null};\ x_2 := temp\}$

$A_o:\ x_2 \neq \mathtt{null} \rightarrow \{out := x_2;\ x_2 := \mathtt{null}\}$

Asynchronous parallel composition of two Buffer processes

Asynchronous Models- Asynchronous Processes

Asynchronous process composition



Asynchronous Process Composition

Let $P_1 = (I_1, O_1, S_1, Init_1, \{\mathcal{A}^1_x \mid x \in I_1\}, \{\mathcal{A}^1_y \mid y \in O_1\}, \mathcal{A}_1)$ and $P_2 = (I_2, O_2, S_2, Init_2, \{\mathcal{A}^2_x \mid x \in I_2\}, \{\mathcal{A}^2_y \mid y \in O_2\}, \mathcal{A}_2)$ be two asynchronous processes such that O_1 and O_2 are disjoint. Then the *parallel composition* $P_1 \mid P_2$ is the asynchronous process P defined by:

- the set S of state variables is $S_1 \cup S_2$;
- the set O of output channels is $O_1 \cup O_2$;
- the set I of input channels is $(I_1 \cup I_2) \setminus O$;
- the initialization is given by $Init_1; Init_2$;
- for each input channel $x \in I$, (1) if $x \notin I_2$, then the set of input tasks $\mathcal{A}_x$ is $\mathcal{A}^1_x$, (2) if $x \notin I_1$, then the set of input tasks $\mathcal{A}_x$ is $\mathcal{A}^2_x$, and (3) if $x \in I_1 \cap I_2$, then for each task $A_1 \in \mathcal{A}^1_x$ and $A_2 \in \mathcal{A}^2_x$, the set of input tasks $\mathcal{A}_x$ contains the task described by $Guard_1 \wedge Guard_2 \rightarrow Update_1; Update_2$, where $Guard_1 \rightarrow Update_1$ is the task description of A_1 and $Guard_2 \rightarrow Update_2$ is the task description of A_2;
- for each output channel $y \in O$, (1) if $y \notin O_2$, then the set of output tasks $\mathcal{A}_y$ is $\mathcal{A}^1_y$, (2) if $y \notin O_1$, then the set of output tasks $\mathcal{A}_y$ is $\mathcal{A}^2_y$, (3) if $y \in O_1 \cap I_2$, then for each task $A_1 \in \mathcal{A}^1_y$ and $A_2 \in \mathcal{A}^2_y$, the set of output tasks $\mathcal{A}_y$ contains the task described by $Guard_1 \wedge Guard_2 \rightarrow Update_1; Update_2$, where $Guard_1 \rightarrow Update_1$ is the task description of A_1 and $Guard_2 \rightarrow Update_2$ is the task description of A_2; (4) if $y \in O_2 \cap I_1$, then for each task $A_1 \in \mathcal{A}^1_y$ and $A_2 \in \mathcal{A}^2_y$, the set of output tasks $\mathcal{A}_y$ contains the task described by $Guard_2 \wedge Guard_1 \rightarrow Update_2; Update_1$, where $Guard_1 \rightarrow Update_1$ is the task description of A_1 and $Guard_2 \rightarrow Update_2$ is the task description of A_2;
- the set $\mathcal{A}$ of internal tasks of the composite is $\mathcal{A}_1 \cup \mathcal{A}_2$.

Asynchronous Models- Asynchronous Processes

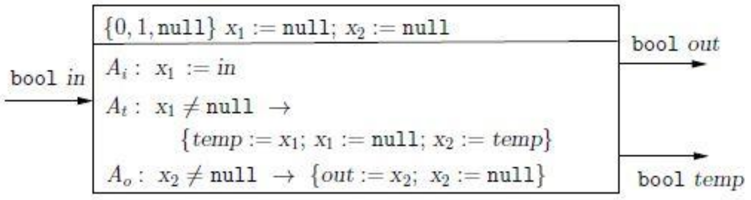
Operations on Processes

- Output hiding
  - If y is an output channel of a process P, the result of hiding y in P gives a process that behaves exactly like P, but y is no longer an output that is observable outside.
  - by removing y from the set of output channels, and turning each output task associated with the channel y into an internal task by declaring y to be a local variable.

  Recall: a local variable is an auxiliary variable used in the description the update code of a task, and is not stored in the state.

- Asynchronous parallel composition o

  two Buffer processes



$\{0, 1, \texttt{null}\}\ x_1 := \texttt{null};\ x_2 := \texttt{null}$

$A_i :\ x_1 := in$

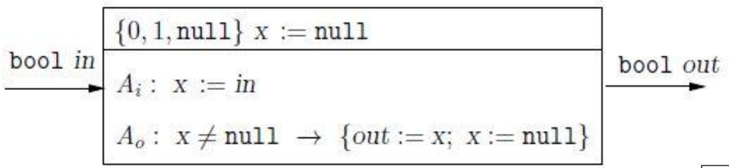
$A_t :\ x_1 \neq \texttt{null} \rightarrow$
$\{temp := x_1;\ x_1 := \texttt{null};\ x_2 := temp\}$

$A_o :\ x_2 \neq \texttt{null} \rightarrow \{out := x_2;\ x_2 := \texttt{null}\}$

Asynchronous Design Primitives

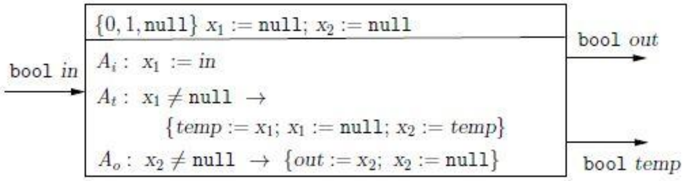
Blocking vs. Non-blocking Synchronization

- In the asynchronous model, exchange of information between two processes (synchronization between them)
  - occurs when the production of an output by one process is matched with the consumption of the corresponding input by another.
- This is a form of **blocking** communication where the producer P1 needs the cooperation of the receiver P2 to produce an output on the channel x. A process that is willing to accept every input in every state cannot prevent the producer from producing outputs, and is said to be **non-blocking**.
- Non-blocking Process
  - An asynchronous process P is said to be non-blocking if for every input channel x, the disjunction of the guards of the tasks in the set Ax of tasks associated with the channel x is valid.


$\{0,1,\mathtt{null}\}\ x := \mathtt{null}$
bool in
$A_i:\ x := in$
$A_o:\ x \neq \mathtt{null} \rightarrow \{out := x;\ x := \mathtt{null}\}$
bool out

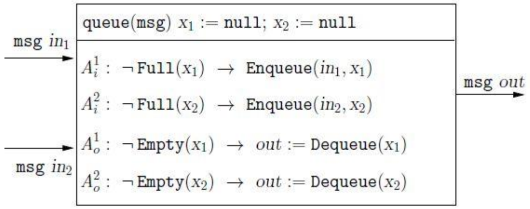
Process Buffer

is non-blocking.


$\{0,1,\mathtt{null}\}\ x_1 := \mathtt{null};\ x_2 := \mathtt{null}$
bool in
$A_i:\ x_1 := in$
$A_t:\ x_1 \neq \mathtt{null} \rightarrow$
$\{temp := x_1;\ x_1 := \mathtt{null};\ x_2 := temp\}$
$A_o:\ x_2 \neq \mathtt{null} \rightarrow \{out := x_2;\ x_2 := \mathtt{null}\}$
bool out
bool $temp$

Process Merge

is blocking.


$\mathtt{queue}(\mathtt{msg})\ x_1 := \mathtt{null};\ x_2 := \mathtt{null}$
msg in_1
msg in_2
$A_i^1:\ \neg\mathtt{Full}(x_1) \rightarrow \mathtt{Enqueue}(in_1, x_1)$
$A_i^2:\ \neg\mathtt{Full}(x_2) \rightarrow \mathtt{Enqueue}(in_2, x_2)$
$A_o^1:\ \neg\mathtt{Empty}(x_1) \rightarrow out := \mathtt{Dequeue}(x_1)$
$A_o^2:\ \neg\mathtt{Empty}(x_2) \rightarrow out := \mathtt{Dequeue}(x_2)$
msg out

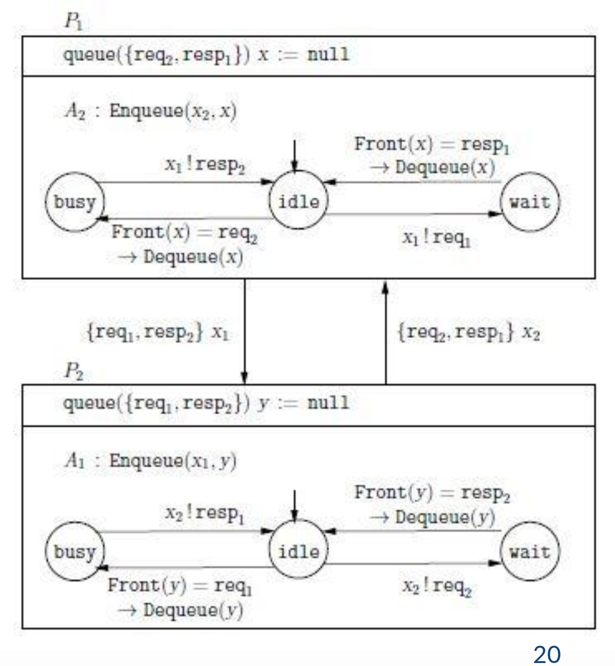
Process DoubleBuffer (composing two Buffer)

is non-blocking.

Asynchronous Design Primitives

Deadlocks

- In a system composed of multiple processes, a **deadlock** refers to a situation in which each process is waiting for some other process to execute a task, but no task is enabled, and thus, there is no continuation of the execution.


P_1
queue({req_2, $resp_1$}) x := null
A_2 : Enqueue(x_2, x)
x_1 ! $resp_2$
Front(x) = $resp_1$ → Dequeue(x)
busy
idle
wait
Front(x) = req_2 → Dequeue(x)
x_1 ! req_1
{req_1, $resp_2$} x_1
{req_2, $resp_1$} x_2
P_2
queue({req_1, $resp_2$}) y := null
A_1 : Enqueue(x_1, y)
x_2 ! $resp_1$
Front(y) = $resp_2$ → Dequeue(y)
busy
idle
wait
Front(y) = req_1 → Dequeue(y)
x_2 ! req_2

- In general, a state **s** of an asynchronous process **P** is a deadlock state if
  (1) no task is enabled in state **s**
  (2) state **s** does not correspond to "successful" termination of the system.

Asynchronous Design Primitives

Shared Memory

- In a shared memory architecture, processes communicate by reading and writing shared variables (shared objects).
- In the **asynchronous mode**l, executions of different **tasks are interleaved**.
  - A crucial design decision concerns how much computation can happen in one computation step of a task, that is, which operations are supported by shared objects as atomic operations that can be executed in a single step.
- How to model shared variables as asynchronous processes?
  - **Model of atomic registers** -the only allowed operations are the most basic read and write operations.
- **Data races** = interference between concurrent accesses to shared objects by asynchronous processes (i.e a bug is caused by the other process accessing the shared register in between the execution of read and write access statements of one process)
- **Mutual exclusion** - no two processes should be in the critical section simultaneously.

Asynchronous Design Primitives

Fairness

- The execution of a process in the asynchronous model is obtained by interleaving executions of different tasks. At every step of the execution, if multiple tasks can be executed, there is a choice.

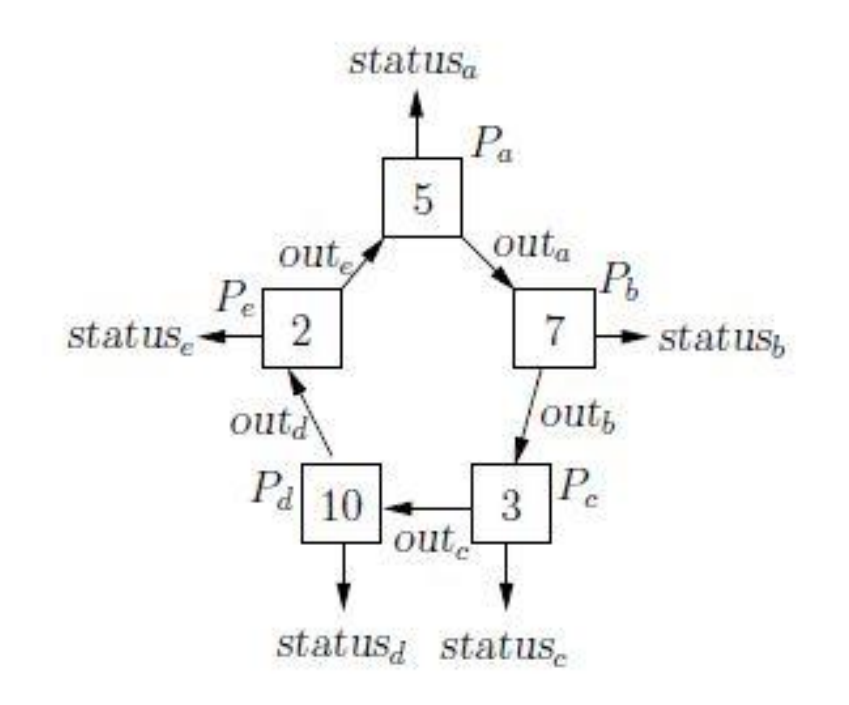
- **Fairness** = “execution of a task can be delayed arbitrarily long, but not forever“.

- The standard mathematical framework for capturing the informal assumption requires us to consider finite executions - ω-execution.

Asynchronous Coordination Protocols

Leader Election

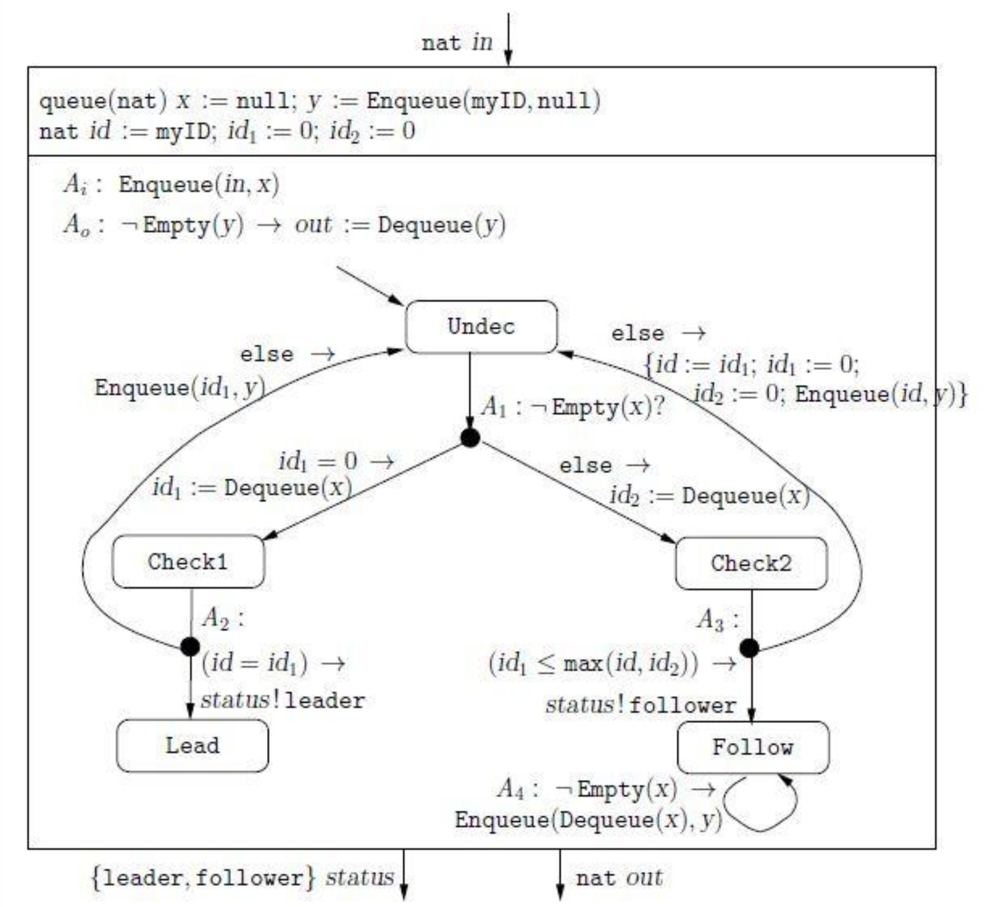
- each node has a unique identifier
- the protocol consists of a strategy for nodes to exchange messages so that eventually a single node declares itself to be the leader with the remaining nodes declaring themselves to be followers.



- We model each network node as an asynchronous process P.
- The input channel *in* receives identifiers sent by the unique predecessor of P in the ring, and the output channel *out* sends identifiers to the unique successor of P in the ring.
- An internal queue x is used to store messages received on the channel in, and the queue y is used to store messages to be sent, which get delivered by the output task on the channel out one by one.
- When the process concludes that it is either the leader or one of the followers, the decision is issued on the output channel status.
- the underlying network connects the nodes in a unidirectional ring
  - To form a ring, we create multiple instances of the process P and compose them together using the asynchronous composition operation.

Asynchronous Coordination Protocols

Leader Election


nat in
queue(nat) x := null; y := Enqueue(myID, null)
nat id := myID; id_1 := 0; id_2 := 0
A_i : Enqueue(in, x)
A_o : $\neg$Empty(y) → out := Dequeue(y)
Undec
else → Enqueue(id_1, y)
else → {id := id_1; id_1 := 0; id_2 := 0; Enqueue(id, y)}
A_1 : $\neg$Empty(x)?
id_1 = 0 → id_1 := Dequeue(x)
else → id_2 := Dequeue(x)
Check1
Check2
A_2 :
($id = id_1$) → $status$!leader
A_3 :
($id_1 \leq$ max(id, id_2)) → $status$!follower
Lead
Follow
A_4 : $\neg$Empty(x) → Enqueue(Dequeue(x), y)
{leader, follower} $status$
nat out

- Our goal is to complete the description of the process P so that when multiple instances of this process are composed to form a ring, the following requirements are met:
  - every process eventually terminates, that is, there is no infinite execution of the protocol;
  - in every terminating execution, exactly one process has output the value leader on its output channel status, and the remaining processes have output the value follower on their output channels status.
- Initially, each process sends its identifier to two successive processes along the ring. To achieve this, each process first sends its identifier, as well as the first input message it receives, on the output channel. When a process receives two messages on the input channel, it knows its own identifier, captured by the variable id, the identifier of its predecessor, captured by the variable id1, and the identifier of the predecessor's predecessor, captured by the variable id2.

References

- [1] Principles of Embedded Computation, Rajeev Alur

  http://www.seas.upenn.edu/~cis540/

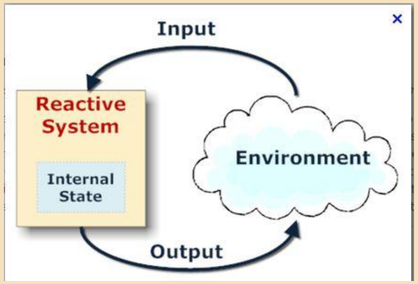
- [2] Design of Embedded Systems: Models, Validation and Synthesis, Alberto Sangiovanni-Vincentelli
- https://inst.eecs.berkeley.edu/~ee249/fa07/

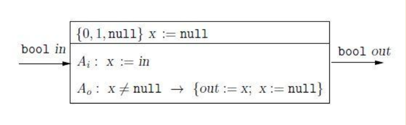
- Rajeev Alur, Principles of Cyber-Physical Systems, 2015
  COTA:9491
- https://www.bcucluj.ro/ro/despre-noi/filiala/biblioteca-de-matematic%C4%83-%C5%9Fi-informatic%C4%83)

10/25/2025 25

CMES – Today Bring it All Together

Asynchronous Model of Computation

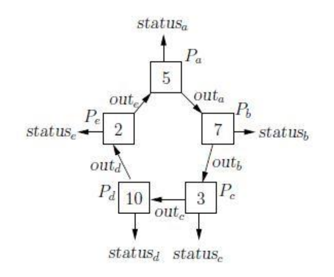


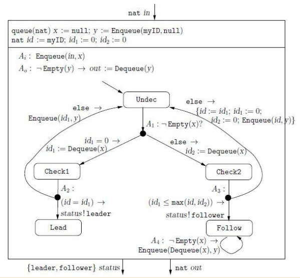

bool in
$\{0,1,\texttt{null}\}\ x := \texttt{null}$
$A_i:\ x := in$
$A_o:\ x \neq \texttt{null} \rightarrow \{out := x;\ x := \texttt{null}\}$
bool out

Buffer

Input, Output, and Internal Actions

Example. Leader Election.




nat in
queue(nat) x := null; y := Enqueue(myID, null)
nat id := myID; $id_1 := 0$; $id_2 := 0$
A_i : Enqueue(in, x)
A_o : ¬Empty(y) → out := Dequeue(y)
Undec
else →
Enqueue(id_1, y)
else →
{$id := id_1$; $id_1 := 0$;
$id_2 := 0$; Enqueue(id, y)}
A_1 : ¬Empty(x)?
$id_1 = 0$ →
id_1 := Dequeue(x)
else →
id_2 := Dequeue(x)
Check1
Check2
A_2 :
($id = id_1$) →
$status$!leader
A_3 :
($id_1 \leq$ max(id, id_2)) →
$status$!follower
Lead
Follow
A_4 : ¬Empty(x) →
Enqueue(Dequeue(x), y)
{leader, follower} $status$
nat out

Next Lecture

- Lecture on SLR

Thank You For Your Attention!

- ExitTicket
- Mentimeter
  - menti.com
  - Code: ?

Software Systems Verification and Validation

“Tell me and I forget, teach me and I may remember, involve me and I learn.”

(Benjamin Franklin)