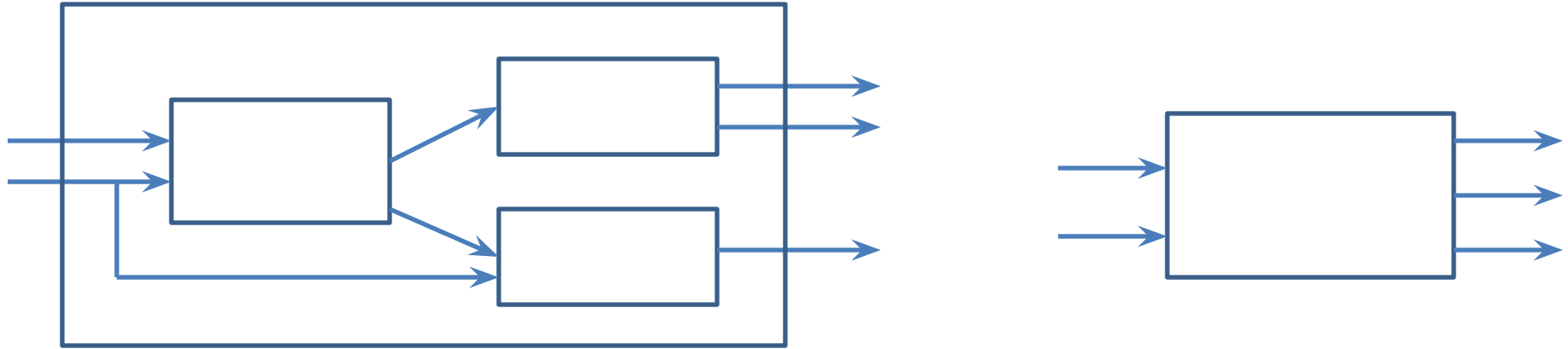


Principles of Cyber-Physical Systems

Chapter 2: Synchronous Model

Instructor: Rajeev Alur
alur@cis.upenn.edu

Model-Based Design

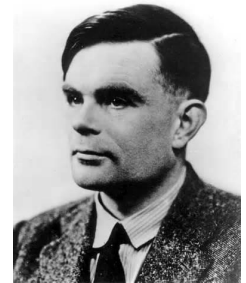


- ❑ Block Diagrams
 - Widely used in industrial design
 - Tools: Simulink, Modelica, LabView, RationalRose...
- ❑ Key question: what is the execution semantics?
 - What is a base component?
 - How do we compose components to form complex components?

Functional vs Reactive Computation

□ Classical model of computation: Functional

- Given inputs, a program produces outputs
- Desired functionality described by a mathematical function
- Example: Sorting of names, Shortest paths in a weighted graph
- Theory of computation provides foundation for this
- Canonical model: Turing machines



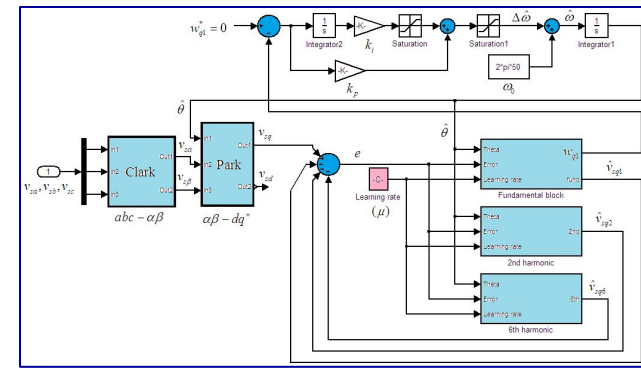
□ Reactive

- System interacts with its environment via inputs and outputs in an ongoing manner
- Desired behaviors: which sequences of observed input/output interactions are acceptable?
- Example: Cruise controller in a car

Sequential vs Concurrent Computation

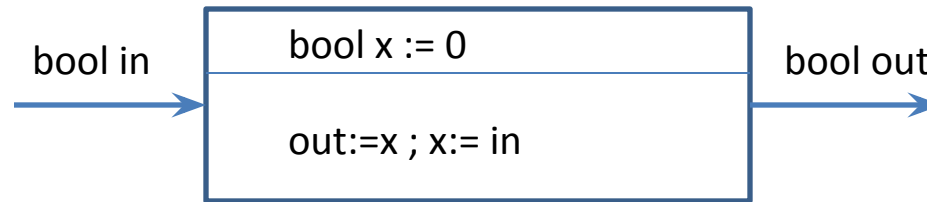
- ❑ Classical model of computation: Sequential
 - A computation is a sequence of instructions executed one at a time
 - Well understood and canonical model: Turing machines
- ❑ Concurrent Computation
 - Multiple components/processes exchanging information and evolving concurrently
 - Logical vs physical concurrency
 - Broad range of formal models for concurrent computation
 - Key distinction: Synchronous vs Asynchronous

Synchronous Models



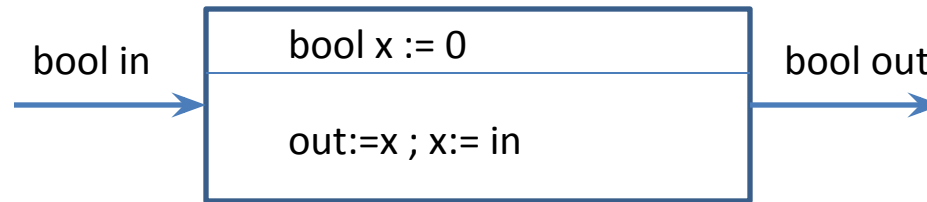
- ❑ All components execute in a sequence of (logical) rounds in lock-step
- ❑ Example: Component blocks in digital hardware circuit
 - Clock drives all components in a synchronized manner
- ❑ Key idea in synchronous languages:
 - Design system using such a synchronous round-based computation
 - Benefit: Design is simpler (why?)
 - Challenge: Ensure synchronous execution even if implementation platform is not single-chip hardware

First Example: Delay

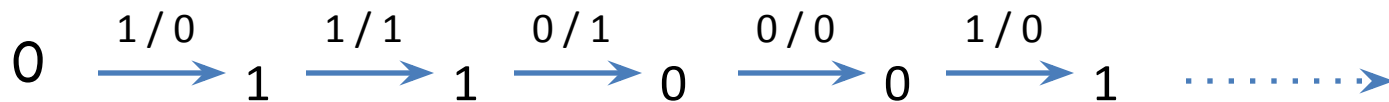


- ☐ Input variable: in of type Boolean
- ☐ Output variable: out of type Boolean
- ☐ State variable: x of type Boolean
- ☐ Initialization of state variables: assignment $x:=0$
- ☐ In each round, in response to an input, produce output and update state by executing the update code: $out:=x; x:=in$

Delay: Round-based Execution



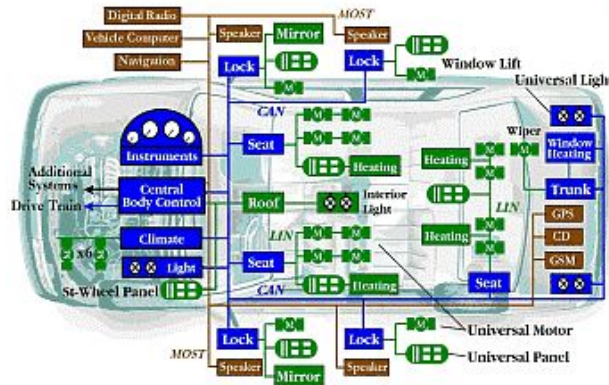
- ❑ Initialize state to 0
- ❑ Repeatedly execute rounds. In each round:
Choose a value for the input variable in
Execute the update code to produce output out and change state
- ❑ Sample execution:



[illegible]

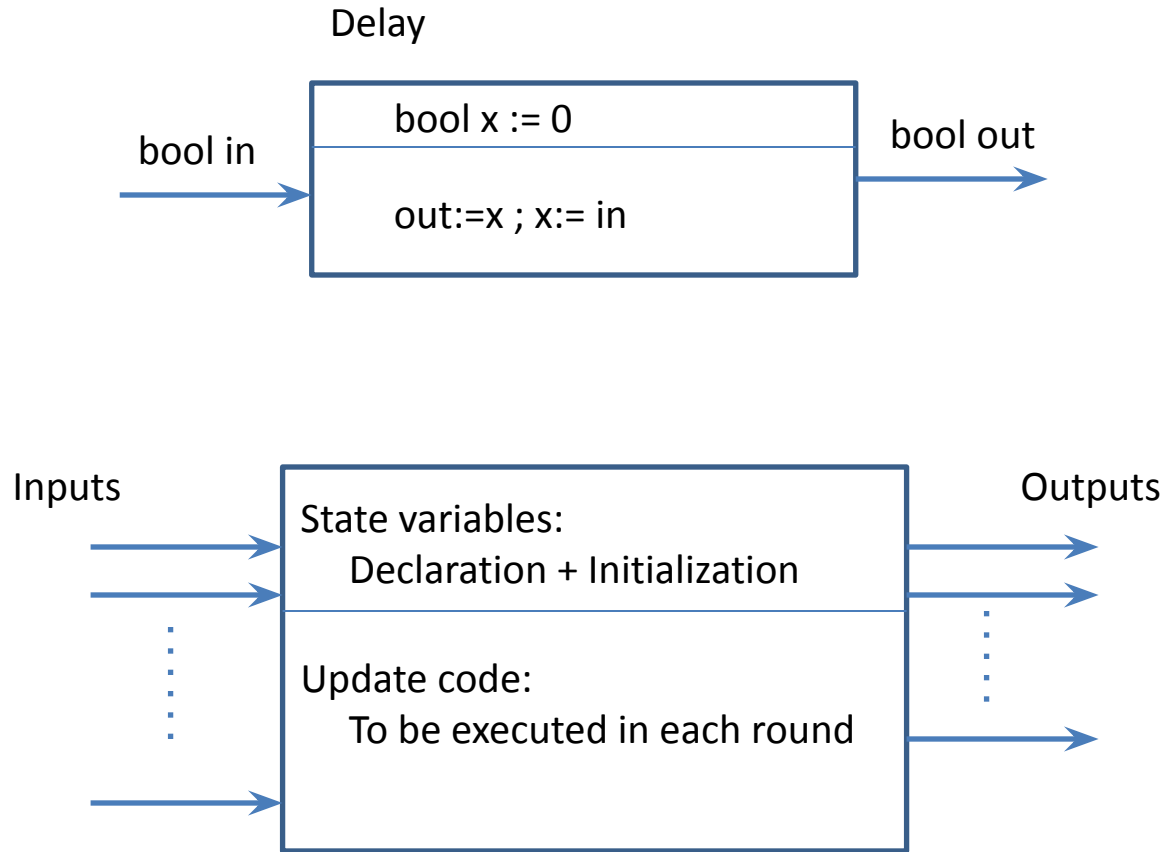
-

Components in an Automobile



- ❑ Components need to communicate and coordinate over a shared bus
- ❑ Design abstraction: Synchronous **time-triggered communication**
 - Time is divided into slots
 - In each slot, exactly one component sends a message over the bus
- ❑ CAN protocol implements time-triggered communication

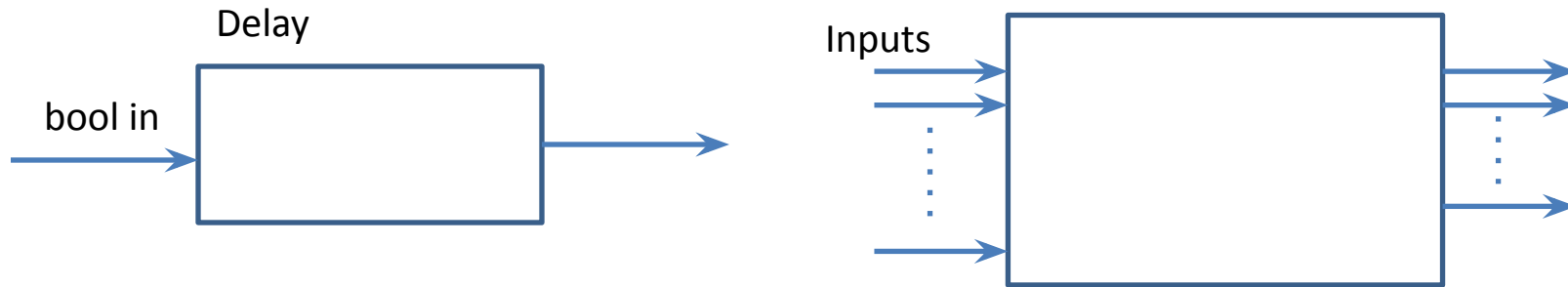
Synchronous Reactive Component



Model Definition

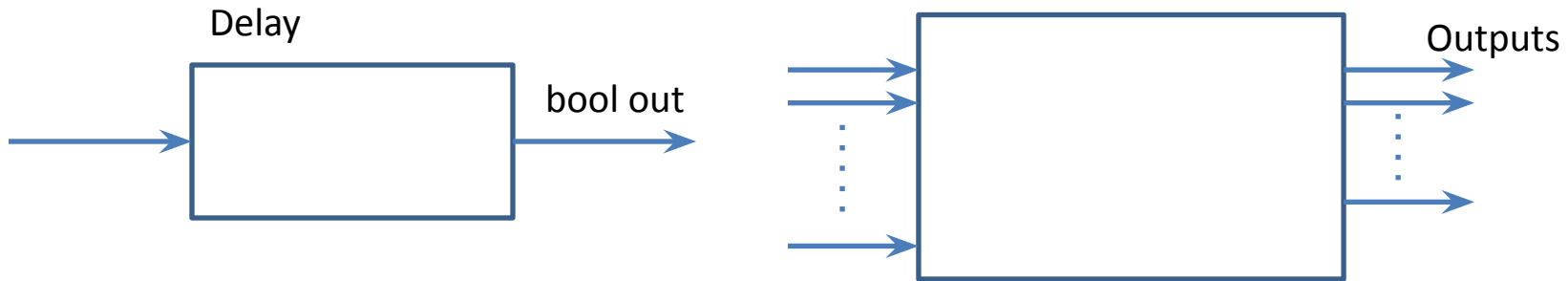
- ❑ Syntax: How to describe a component?
 - Variable declarations, types, code describing update ...
- ❑ Semantics: What does the description mean?
 - Defined using mathematical concepts such as sets, functions ...
- ❑ Formal: Semantics is defined precisely
 - Necessary for tools for analysis, compilation, verification ...
 - Defining formal semantics for a “real” language is challenging
 - But concepts can be illustrated on a “toy” modeling language
- ❑ Our modeling language: Synchronous Reactive Components
 - Representative of many “academic” proposals
 - Industrial-strength synchronous languages
Esterel, Lustre, VHDL, Verilog, Stateflow...

SRC Definition (1): Inputs



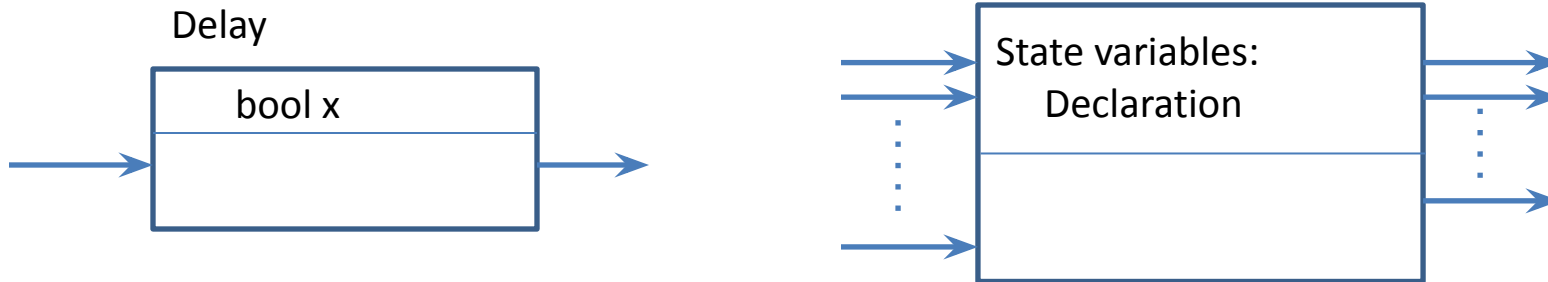
- ❑ Each component has a set I of input variables
 - Variables have types. E.g. bool, int, nat, real, {on, off} ...
- ❑ Input: Valuation of all the input variables
 - The set of inputs is denoted Q_I
- ❑ For Delay
 - I contains a single variable in of type bool
 - The set of inputs is $\{0, 1\}$
- ❑ Example: I contains two variables: int x , bool y
 - Each input is a pair: (integer value for x and 0/1 value for y)

SRC Definition (2): Outputs



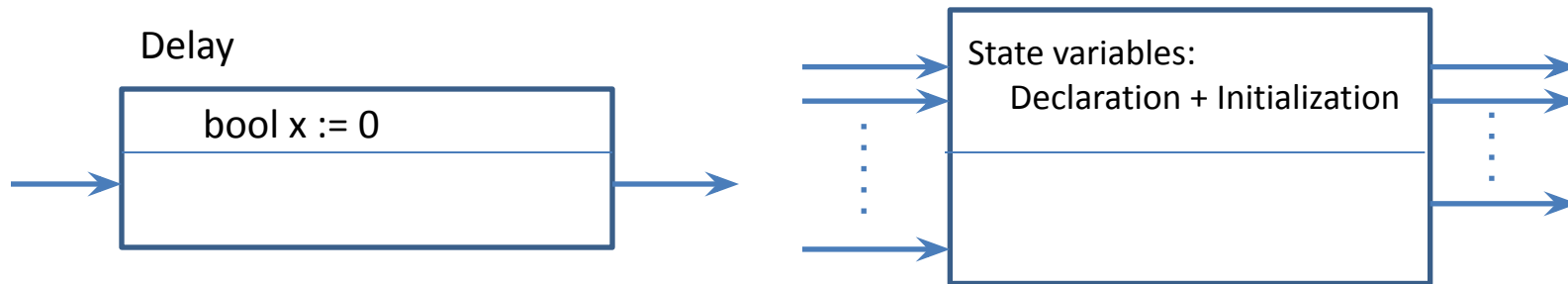
- ❑ Each component has a set O of typed output variables
- ❑ Output: Valuation of all the output variables
 - The set of outputs is denoted Q_O
- ❑ For Delay
 - O contains a single variable out of type bool
 - The set of outputs is $\{0, 1\}$

SRC Definition (3): States



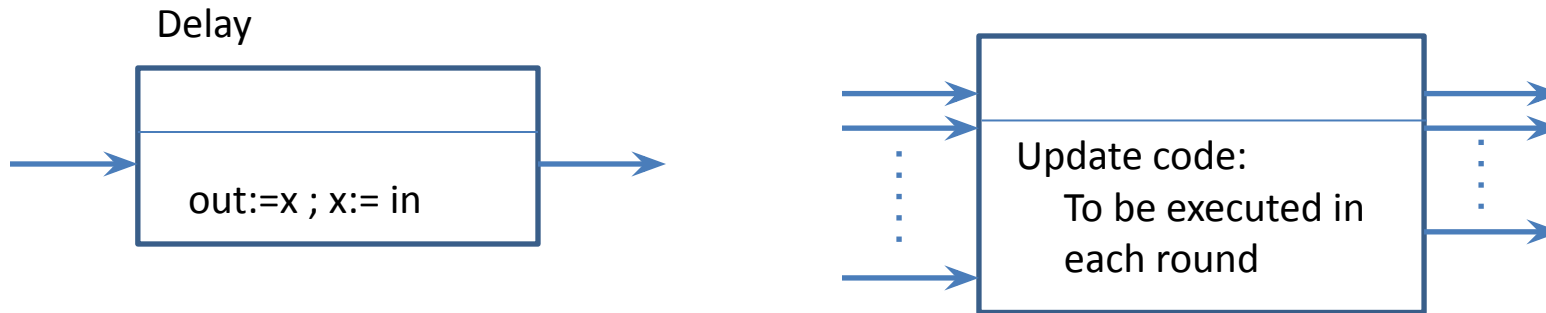
- ❑ Each component has a set S of typed state variables
- ❑ State: Valuation of all the state variables
 - The set of states is denoted Q_S
- ❑ For Delay
 - S contains a single variable x of type `bool`
 - The set of states is $\{0, 1\}$
- ❑ State is internal and maintained across rounds

SRC Definition (4): Initialization



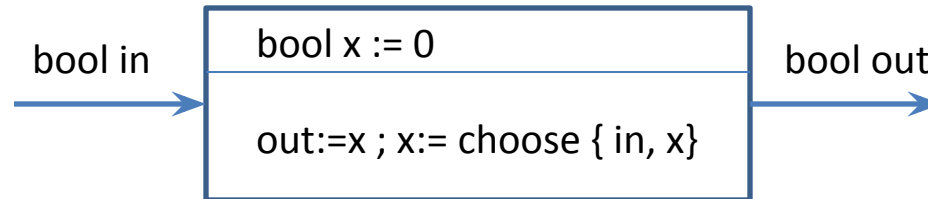
- ❑ Initialization of state variables specified by Init
 - Sequence of assignments to state variables
- ❑ Semantics of initialization:
 - The set [Init] of initial states, which is a subset of Q_s
- ❑ For Delay
 - Init is given by the code fragment $x:=0$
 - The set [Init] of initial states is $\{0\}$
- ❑ Component can have multiple initial states
 - Example: $\text{bool } x := \text{choose } \{0, 1\}$

SRC Definition (5): Reactions



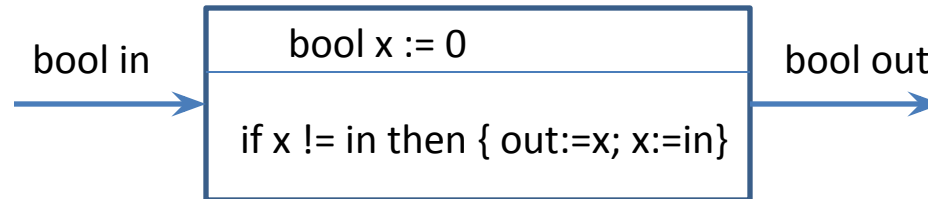
- ❑ Execution in each round given by code fragment React
 - Sequence of assignments and conditionals that assign output variables and update state variables
- ❑ Semantics of update:
 - The set [React] of reactions, where each reaction is of the form (old) state -- input / output \rightarrow (new) state
 - [React] is a subset of $Q_S \times Q_I \times Q_O \times Q_S$
- ❑ For Delay:
 - React is given by the code fragment out:=x; x:=in
 - There are 4 reactions: 0 -0/0- \rightarrow 0; 0 -1/0- \rightarrow 1; 1 -0/1- \rightarrow 0; 1 -1/1- \rightarrow 1

Multiple Reactions



- ❑ During update, either x is updated to input in , or left unchanged
 - Motivation: models that an input may be “lost”
- ❑ Nondeterministic reactions
 - Given (old) state and input, output/new state need not be unique
 - The set [React] of reactions now contains
 - $0 - 0/0 \rightarrow 0$
 - $0 - 1/0 \rightarrow 1$; $0 - 1/0 \rightarrow 0$
 - $1 - 0/1 \rightarrow 0$; $1 - 0/1 \rightarrow 1$
 - $1 - 1/1 \rightarrow 1$

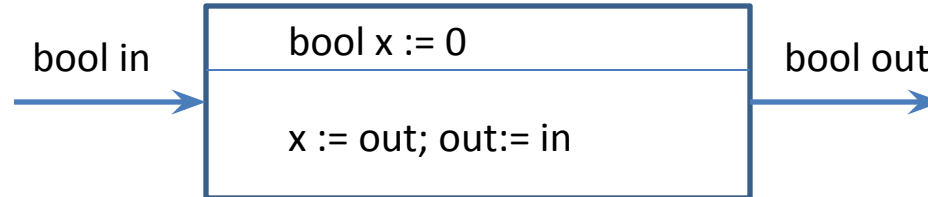
Multiple Reactions



- ❑ A component may not accept all inputs in all states
 - Motivation: “blocking” communication

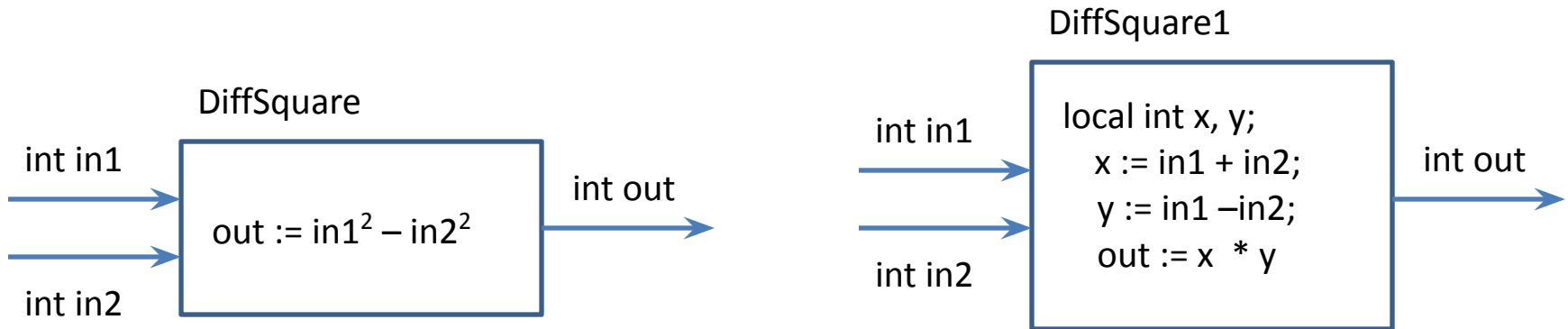
- ❑ Possible set of reactions in certain state/input combinations may be empty
 - The set [React] of reactions now contains
 - 0 -1/0-> 1
 - 1 -0/1-> 1

Syntax Errors



- ❑ If update code cannot be executed, then no reaction possible
 - In above: set [React] of reactions is the empty set
- ❑ Update code expected to satisfy a number of requirements
 - Types of variables and expressions should match
 - Output variables must first be written before being read
 - Output variable must be explicitly assigned a value

Semantic Equivalence



- ❑ Both have identical sets of reactions
- ❑ Syntactically different but semantically equivalent
- ❑ Compiler can optimize code as long as semantics is preserved!

Synchronous Reactive Component Definition

- ❑ Set I of typed input variables: gives set Q_I of inputs
- ❑ Set O of typed output variables: gives set Q_O of outputs
- ❑ Set S of typed state variables: gives set Q_S of states
- ❑ Initialization code $Init$: defines set $[Init]$ of initial states
- ❑ Reaction description $React$: defines set $[React]$ of reactions of the form $s -i/o \rightarrow t$, where s, t are states, i is an input, and o is an output

Synchronous languages in practice:

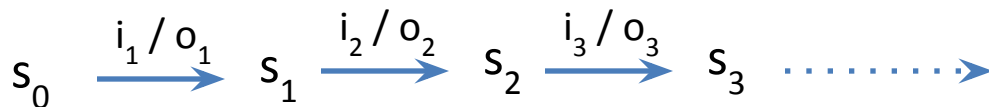
Richer syntactic features to describe $React$

Key to understanding: what happens in a single reaction?

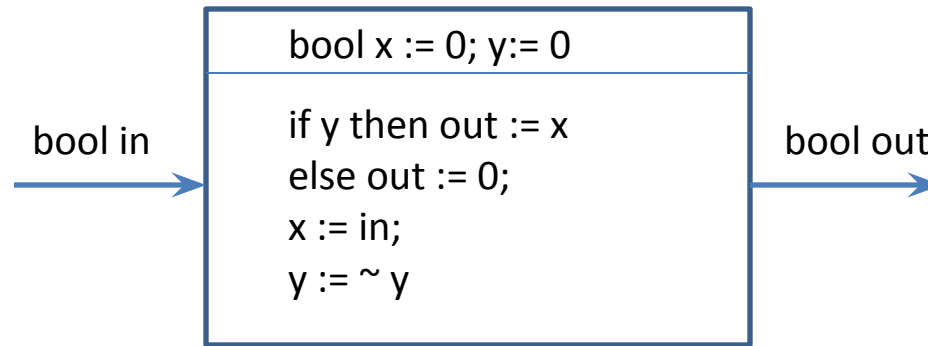
Formal semantics: Necessary for development of tools!

Definition of Executions

- ❑ Given component $C = (I, O, S, \text{Init}, \text{React})$, what are its executions?
- ❑ Initialize state to some state s_0 in $[\text{Init}]$
- ❑ Repeatedly execute rounds. In each round $n=1,2,3,\dots$
Choose an input value i_n in Q_I
Execute React to produce output o_n and change state to s_n
that is, $s_{n-1} - i_n / o_n \sqsubseteq s_n$ must be in $[\text{React}]$
- ❑ Sample execution:

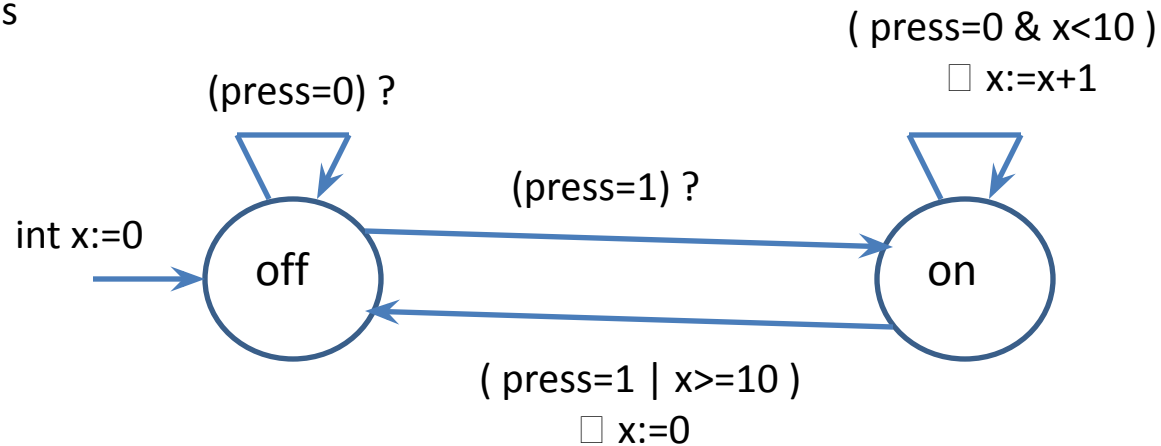


What does this component do ?



Extended State Machines

Input: bool press



mode is a state variable ranging over {on, off}

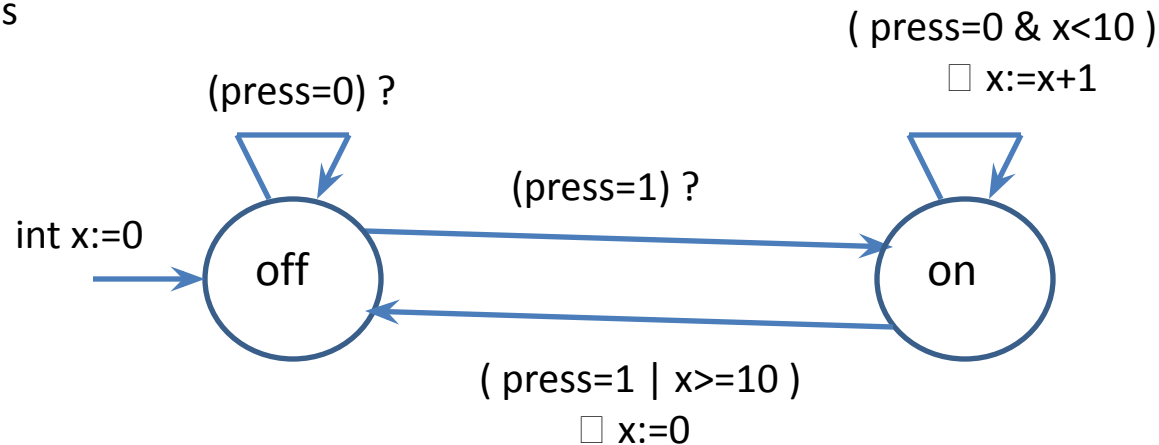
Reaction corresponds to executing a **mode-switch**

Example mode-switch: from on to off with

Guard (press=1 | x>=10) and **Update** code x:=0

Executing ESMs: Switch

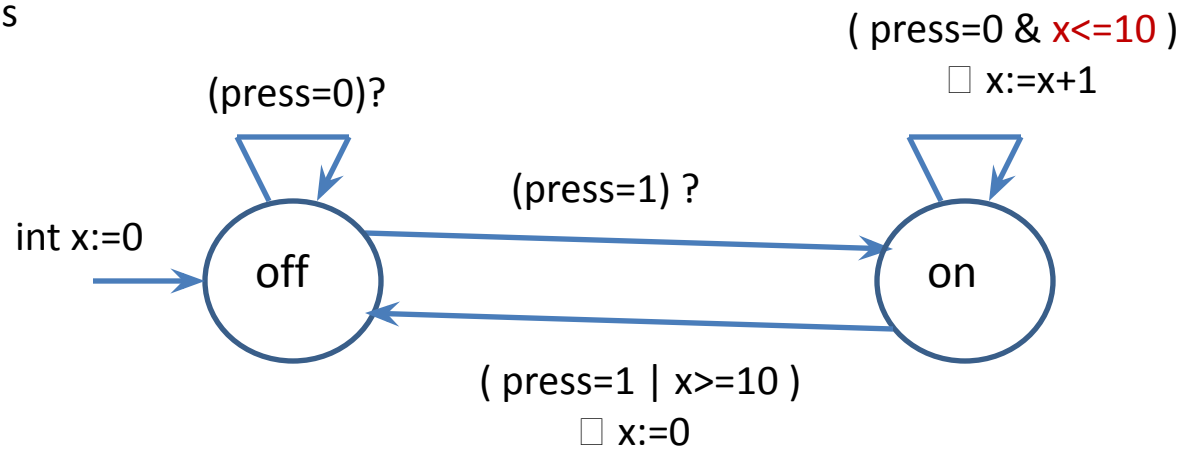
Input: bool press



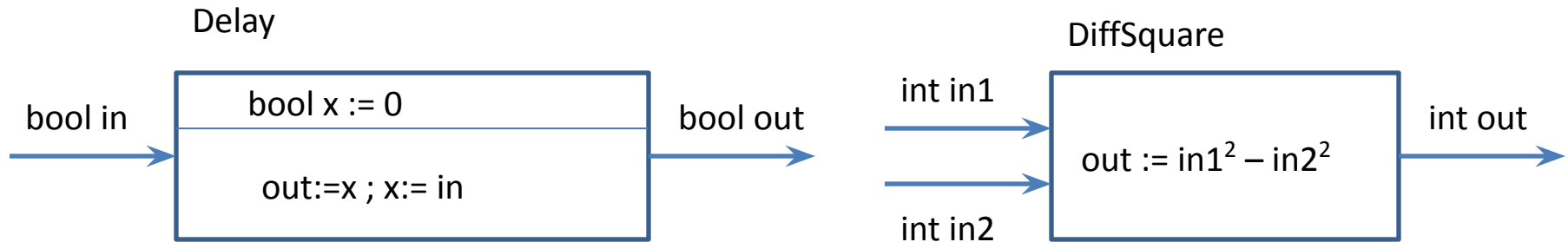
- ❑ State of the component Switch assigns values to mode and x
- ❑ Initial state: (off, 0)
- ❑ Sample Execution:
(off,0) -0→ (off,0) -1→ (on,0) -0→ (on,1) -0→ (on,2) ... -0→(on,10) -0→ (off,0)

Modified Switch: What executions are possible?

Input: bool press

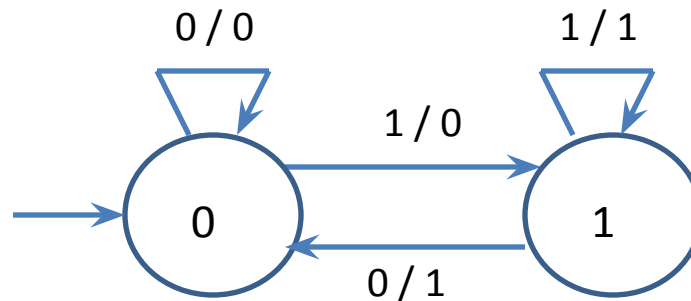
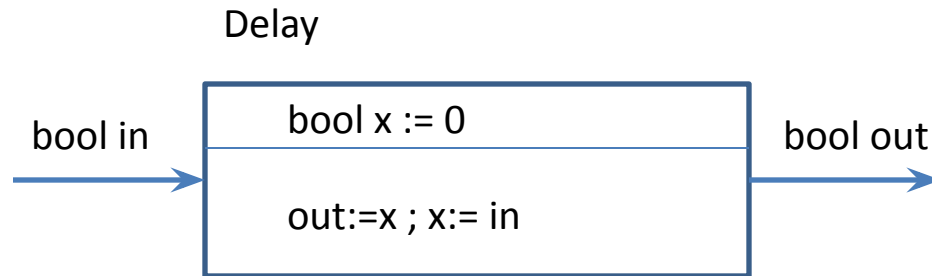


Finite-State Components



- ❑ A component is finite-state if all its variables range over finite types
 - Finite types: `bool`, enumerated types (e.g. `{on, off}`), `int[-5..5]`
 - Delay is finite-state, but DiffSquare is not

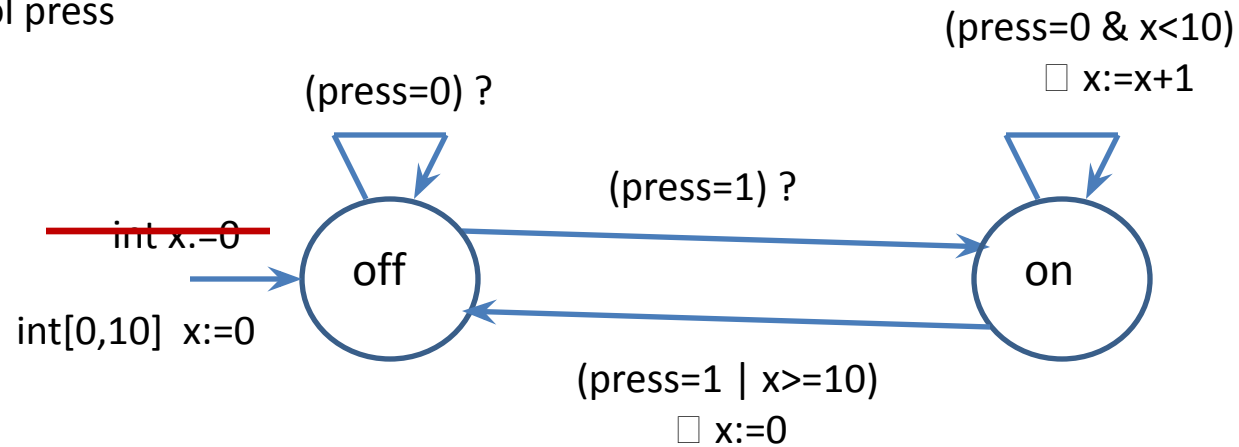
Mealy Machines (for Finite-state components)



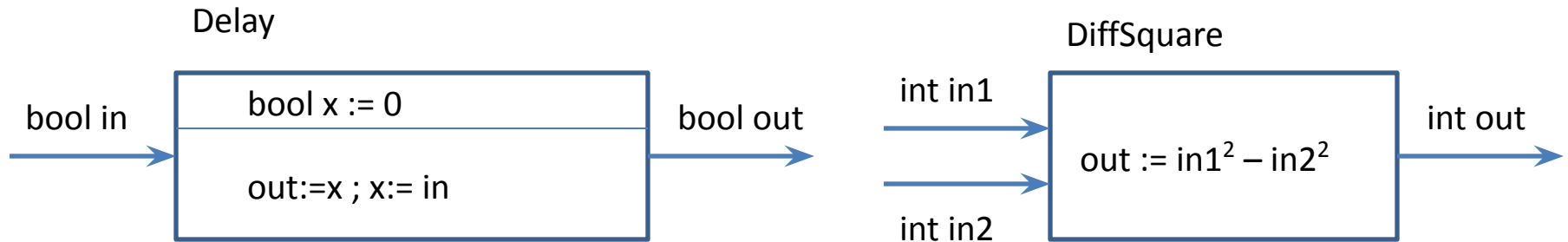
- Finite-state components are amenable to exact, algorithmic analysis

Switch: Is it finite-state?

Input: bool press



Combinational Components



- ❑ A component is combinational if it has no state variables
 - DiffSquare is combinational, but Delay is not
 - Hardware gates are combinational components

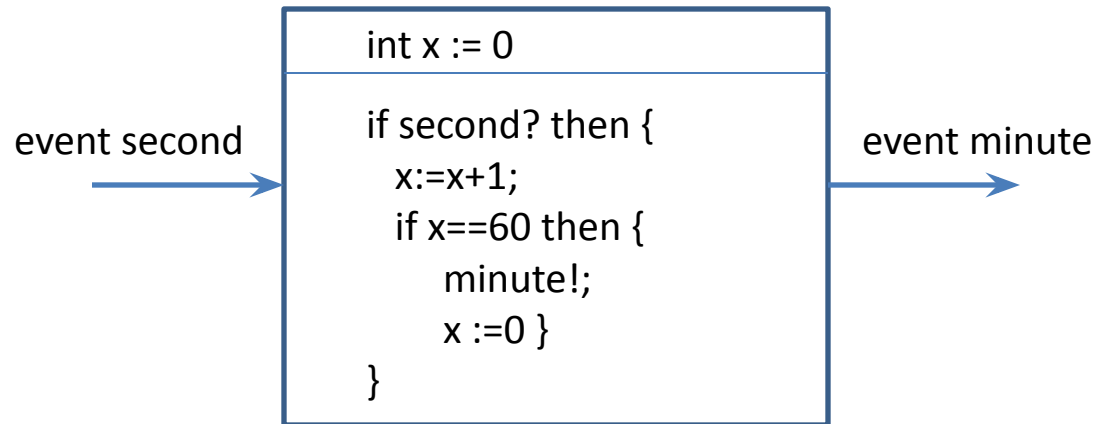
Events

- ❑ Input/output variable can be of type **event**
- ❑ An event can be absent, or present, in which case has a value
 - $\text{event } x$ means x ranges over $\{\text{present}, \text{absent}\}$
 - $\text{event}(\text{bool}) x$ means x ranges over $\{0, 1, \text{absent}\}$
 - $\text{event}(\text{nat}) x$ means x ranges over $\{\text{absent}, 0, 1, 2, \dots\}$
- ❑ Syntax: $x?$ means the test $(x \neq \text{absent})$
- ❑ Syntax: $x!v$ means the assignment $x := v$
- ❑ Event-based communication:
 - If no value is assigned to an output event, then it is absent (by default)
 - Event-triggered component executes only in those rounds where input events are present (actual definition slightly more general, see textbook)
 - Motivation: notion of "clock" can be different for different components

Second-To-Minute

Desired behavior (spec):

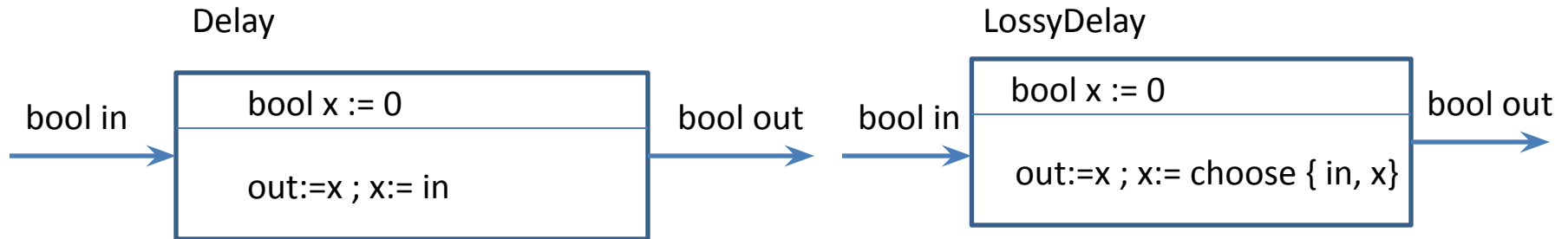
Issue the output event every 60th time the input event is present



□ Event-Triggered Components

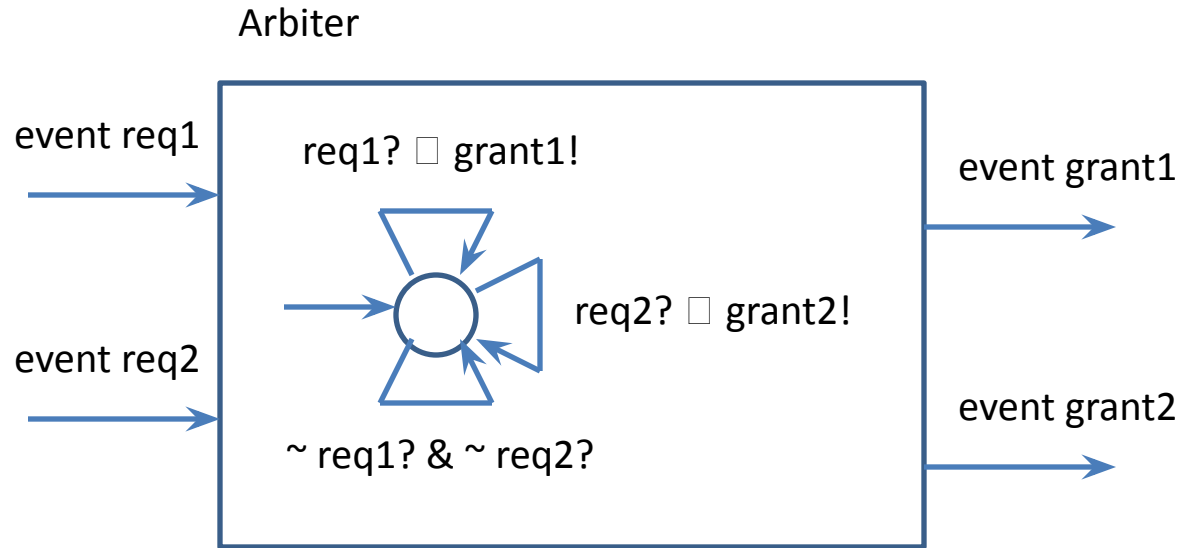
- No need to execute in a round where triggering input events absent
- Read textbook for formal definition

Deterministic Components

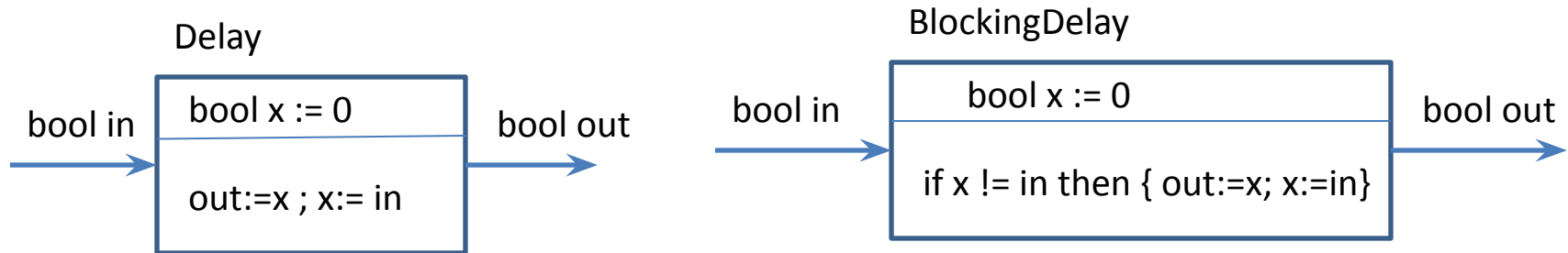


- ❑ A component is deterministic if (1) it has a single initial state, and (2) for every state s and input i , there is a unique state t and output o such that $s \xrightarrow{i/o} t$ is a reaction
 - Delay is deterministic, but LossyDelay is not
- ❑ Deterministic: If same sequence of inputs supplied, same outputs observed (predictable, repeatable behavior)
- ❑ Nondeterminism is useful in modeling uncertainty /unknown
- ❑ Nondeterminism is not same as probabilistic (or random) choice

What does this component do?

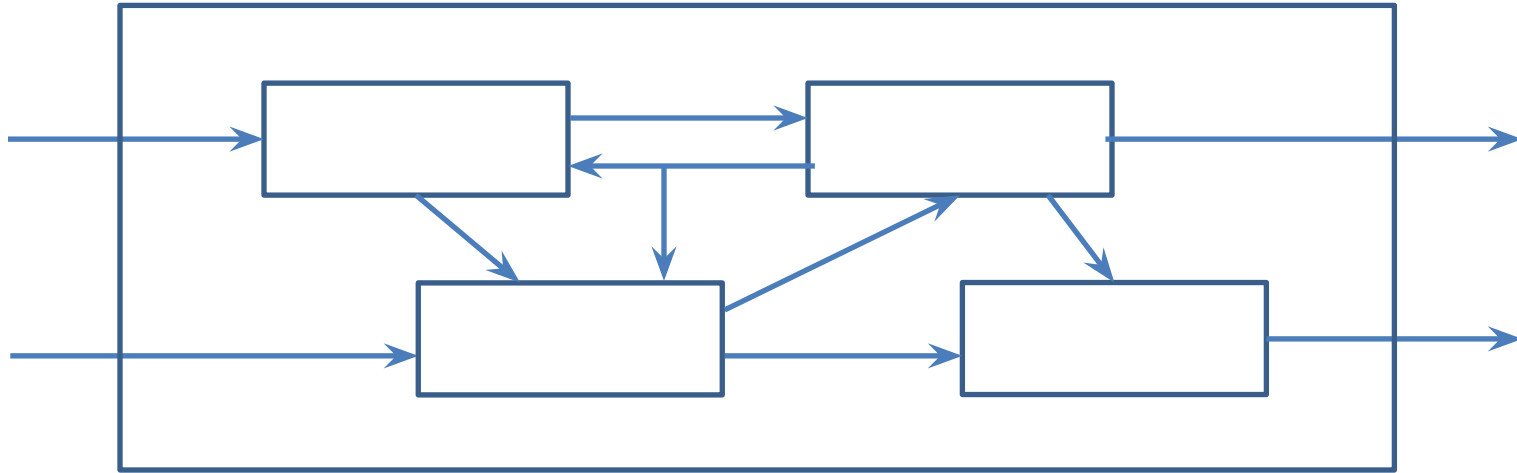


Input Enabled Components



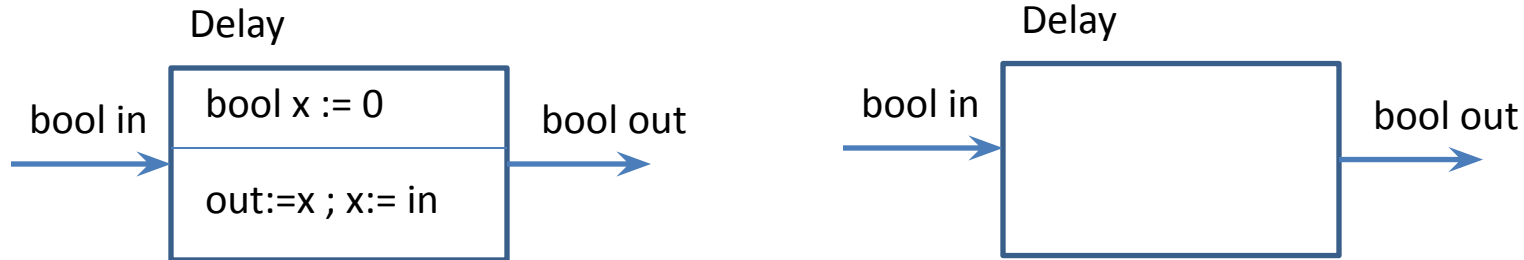
- ❑ A component is input-enabled if for every state s and input i , there exists a state t and an output o such that $s \xrightarrow{i/o} t$ is a reaction
 - Delay is input-enabled, but BlockingDelay is not
- ❑ Not input-enabled means component is making assumptions about the context in which it is going to be used
 - When rest of system is designed, must check that it indeed satisfies these assumptions

Block Diagrams



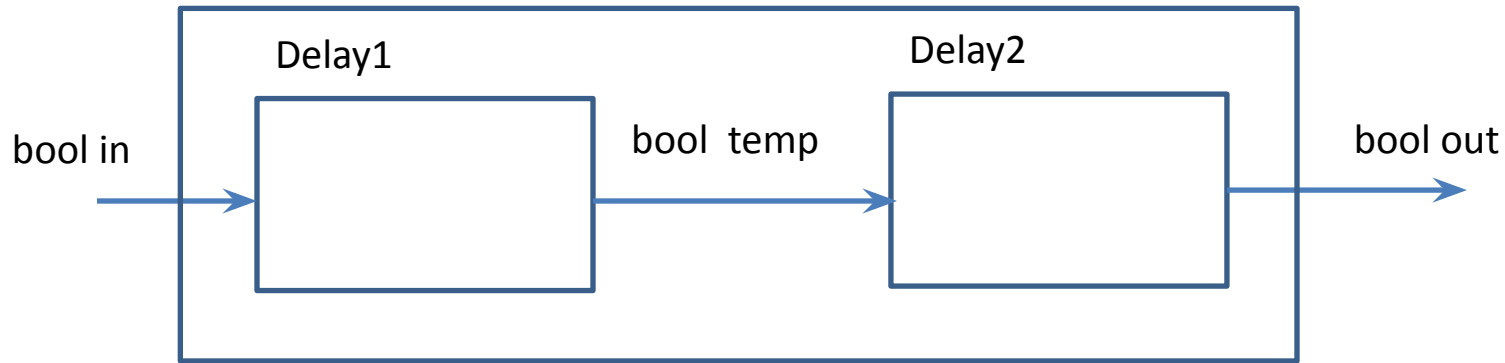
- ❑ Structured modeling
 - How do we build complex models from simpler ones
 - What are basic operations on components?

DoubleDelay



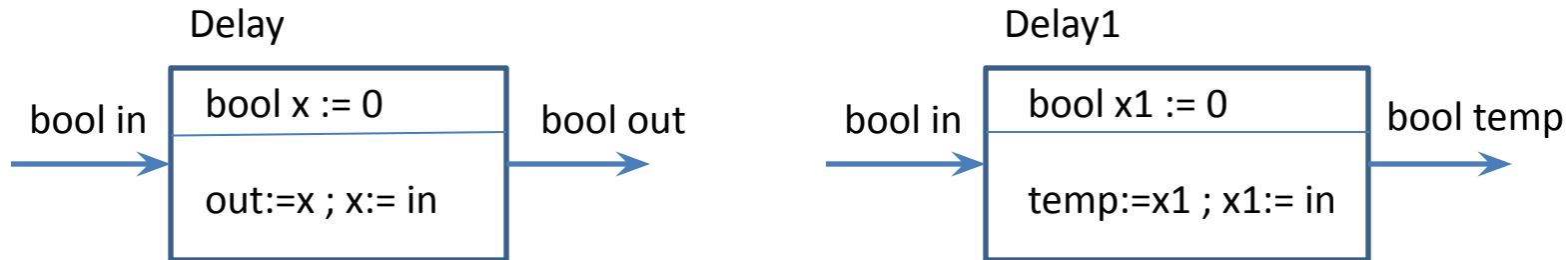
- Design a component with
 - Input: bool in
 - Output: bool out
 - Output in round n should equal input in round $n-2$

DoubleDelay



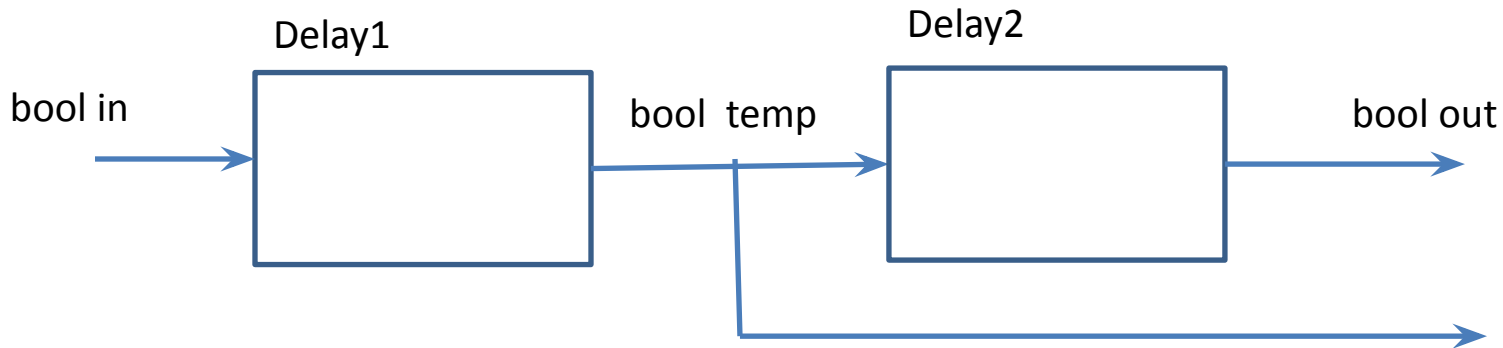
- ❑ Instantiation: Create two instances of Delay
 - Output of Delay1 = Input of Delay2 = Variable temp
- ❑ Parallel composition: Concurrent execution of Delay1 and Delay2
- ❑ Hide variable temp: Encapsulation

Instantiation / Renaming



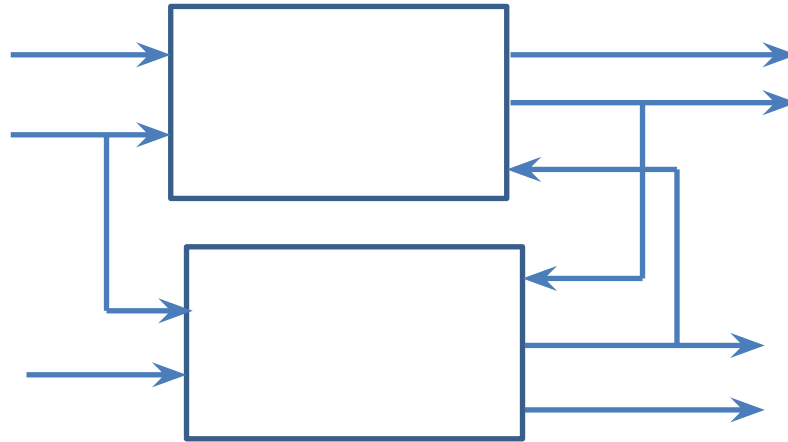
- $\text{Delay1} = \text{Delay}[\text{out} \rightarrow \text{temp}]$
 - Explicit renaming of input/output variables
 - Implicit renaming of state variables
 - Components (I,O,S,Init,React) of Delay1 derived from Delay
- $\text{Delay2} = \text{Delay}[\text{in} \rightarrow \text{temp}]$

Parallel Composition



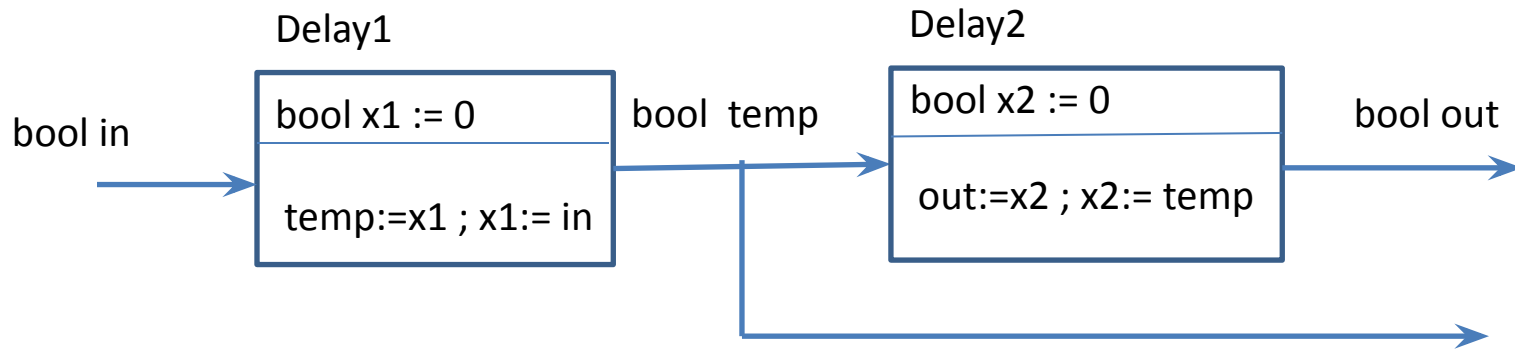
- ❑ $\text{DDelay} = \text{Delay1} \parallel \text{Delay2}$
 - Execute both concurrently
- ❑ When can two components be composed?
- ❑ How to define parallel composition precisely?
 - Input/output/state variables, initialization, and reaction description of composite defined in terms of components
 - Can be viewed as an “algorithm” for compilation

Compatibility of components C1 and C2



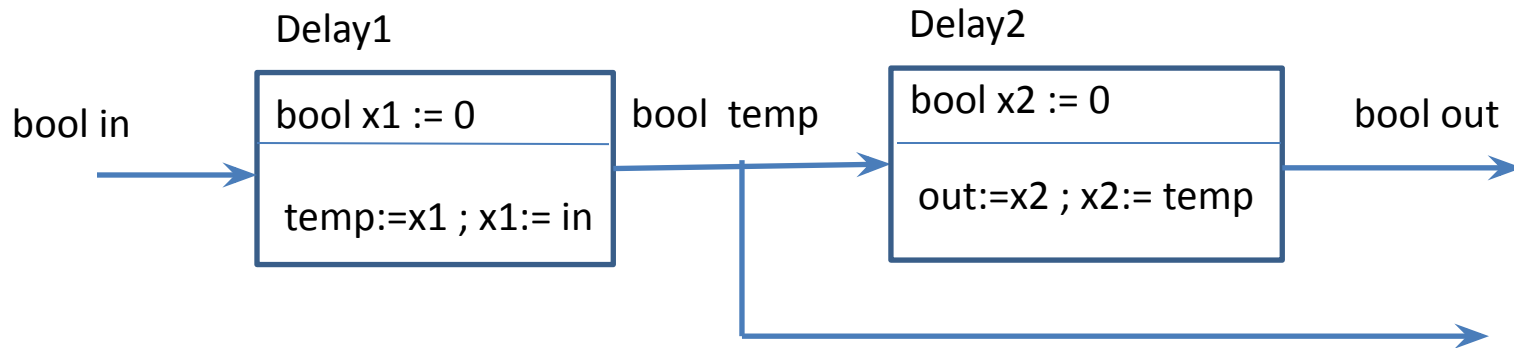
- ☐ Can have common input variables
- ☐ Cannot have common output variables
 - A unique component responsible for values of any given variable
- ☐ Cannot have common state variables
 - State variables can be implicitly renamed to avoid conflicts
- ☐ Input variable of one can be output of another, and vice versa

Outputs of Product



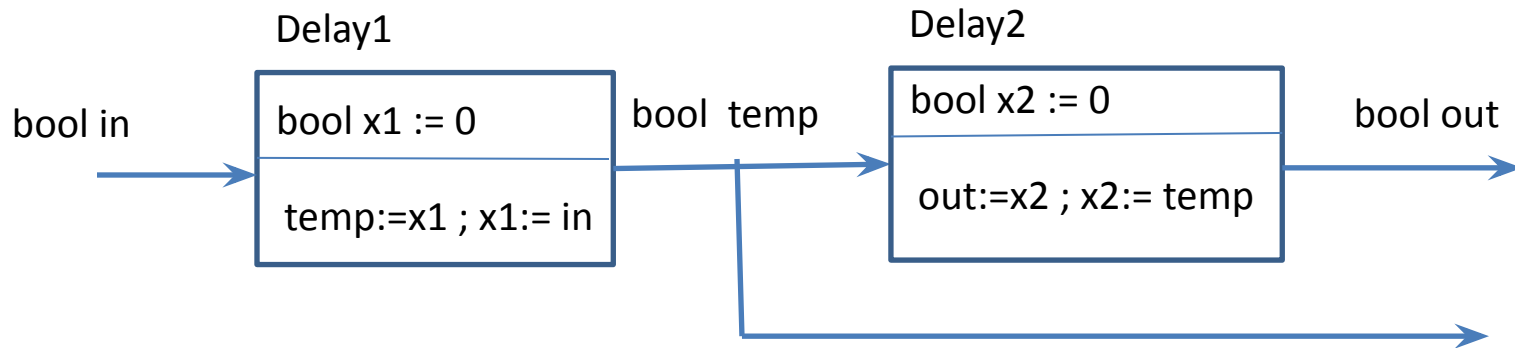
- ❑ Output variables of $\text{Delay1} \parallel \text{Delay2}$ is $\{\text{temp}, \text{out}\}$
 - Note: By default, every output is available to outside world
- ❑ If $C1$ has output vars $O1$ and $C2$ has output vars $O2$ then the product $C1 \parallel C2$ has output vars $O1 \cup O2$

Inputs of Product



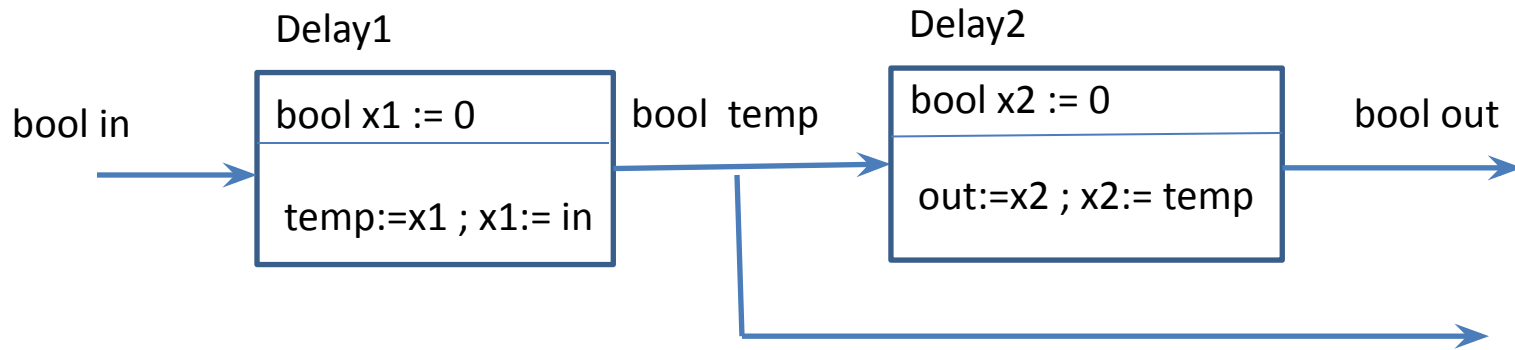
- ❑ Input variables of Delay1 || Delay2 is {in}
 - Even though temp is input of Delay2, it is not an input of product
- ❑ If C1 has input vars I1 and C2 has input vars I2 then the product C1 || C2 has input vars $(I1 \cup I2) \setminus (O1 \cup O2)$
 - A variable is an input of the product if it is an input of one of the components, and not an output of the other

States of Product



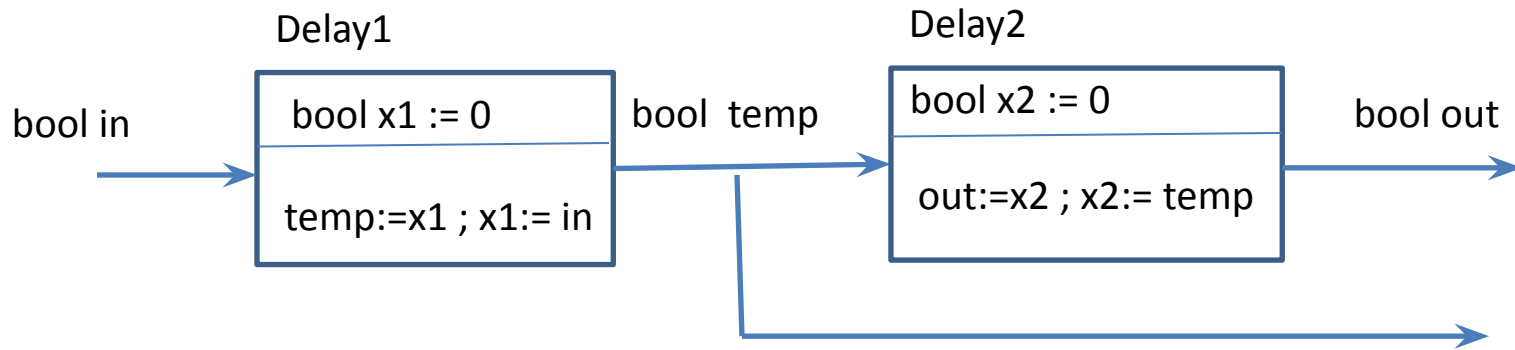
- State variables of Delay1 || Delay2 : {x1, x2}
- If C1 has state vars S1 and C2 has state vars S2 then the product has state vars (S1 U S2)
 - A state of the product is a pair (s1, s2), where s1 is a state of C1 and s2 is a state of C2
 - If C1 has n1 states and C2 has n2 states then the product has (n1 x n2) states

Initial States of Product



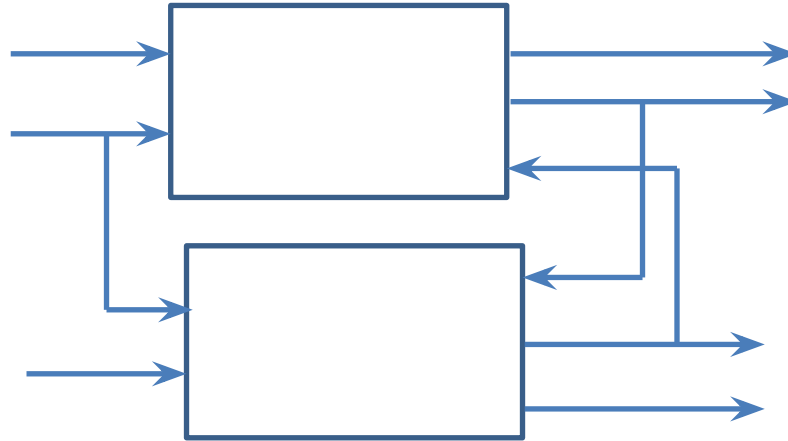
- ❑ The initialization code Init for Delay1 || Delay2 is "x1:=0;x2:=0"
 - Initial state is (0,0)
- ❑ If C1 has initialization Init1 and C2 has initialization Init2 then the product C1 || C2 has initialization Init1; Init2
 - Order does not matter
 - [Init] is the product of sets [Init1] × [Init2]

Reactions of Product



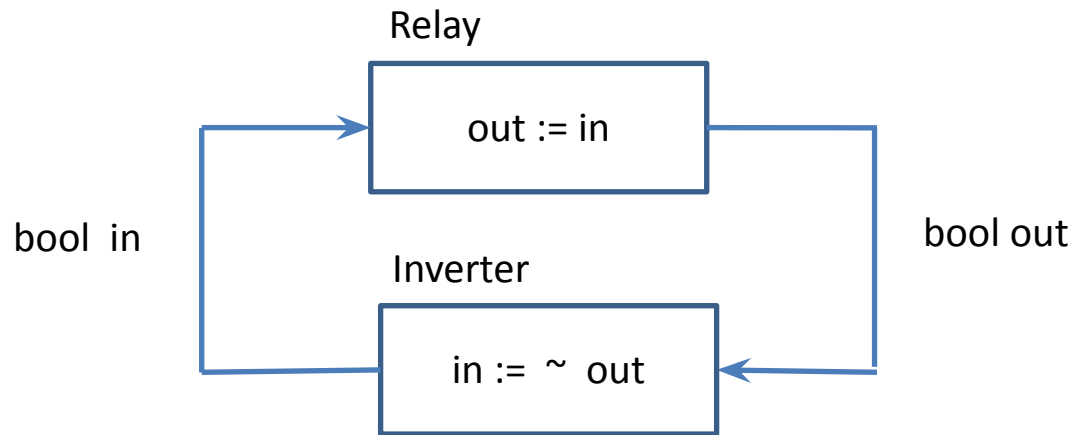
- ❑ Execution of Delay1 || Delay2 within a round
 - Environment provides input value for variable in
 - Execute code "temp:=x1; x1:=in" of Delay1
 - Execute code "out:=x2; x2:=temp" of Delay2

Feedback Composition



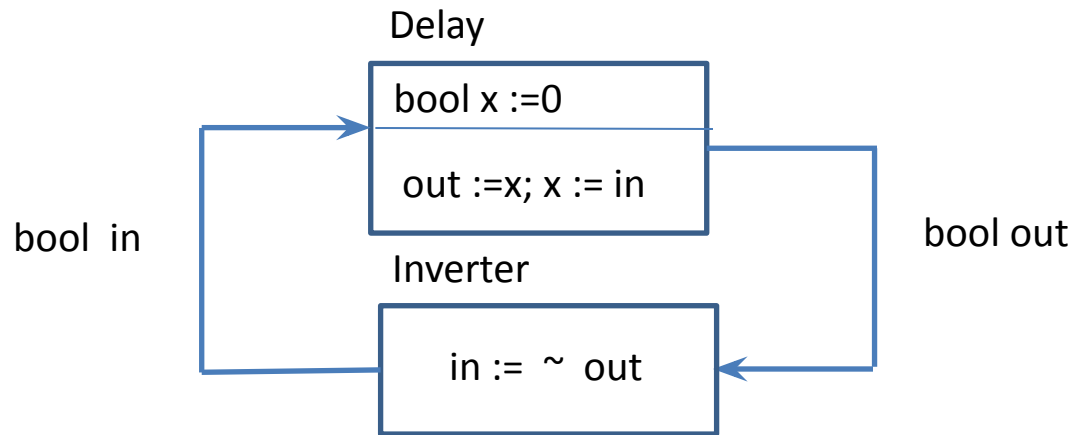
- ☐ When some output of $C1$ is an input of $C2$, and some output of $C2$ is an input of $C1$, how do we order the executions of reaction descriptions React1 and React2 ?
- ☐ Should such composition be allowed at all?

Feedback Composition



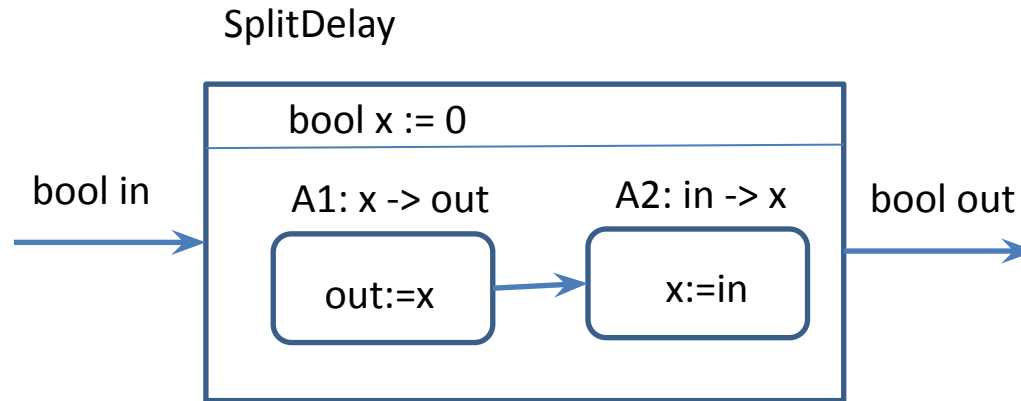
- ❑ For Relay, its output out “awaits” its input in
- ❑ For Inverter, its output in “awaits” its input out
- ❑ In product, cannot order the execution of the two
- ❑ In presence of such cyclic dependency, composition is disallowed
 - Intuition: Combinational cycles should be avoided

Feedback Composition



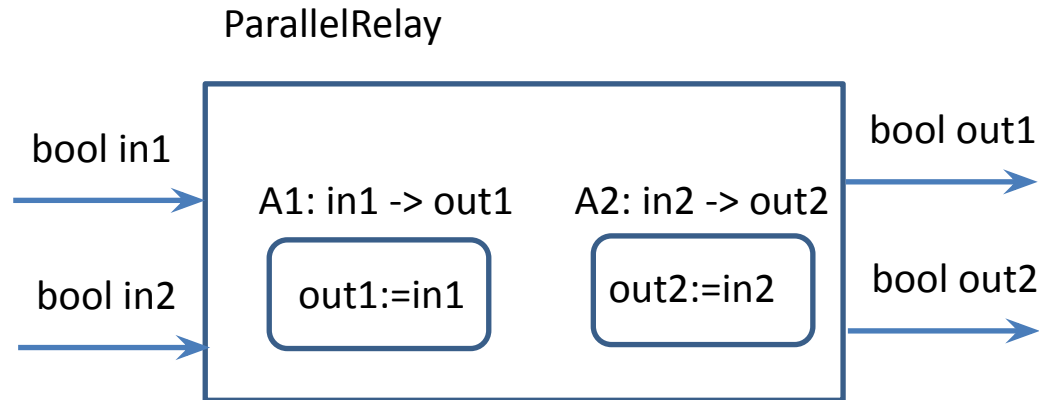
- ❑ For Delay, possible to produce output without waiting for its input by executing the assignment "out := x"
- ❑ Reaction code for product can be "out:=x; in := ~out; x := in"
- ❑ Goal: Refine specification of reaction description so that "await" dependencies among output-input variables are easy to detect
 - Ordering of code-blocks during composition should be easy

Splitting Reaction code into Tasks



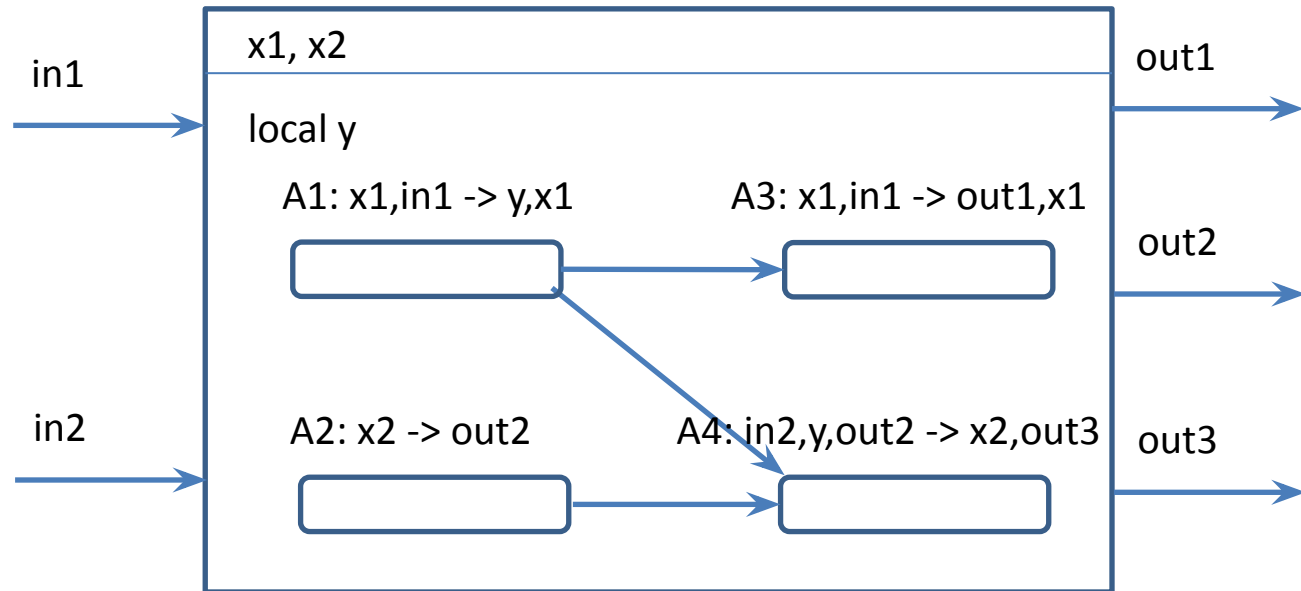
- ❑ A1 and A2 are tasks (atomic blocks of code)
 - Each task specifies variables it reads and writes
 - A1 reads x and writes out
- ❑ Task Graph: Vertices are tasks and edges denote precedence
 - $A1 < A2$ means that A1 should be executed before A2
 - Graph should be acyclic

Example Task Graph



- ❑ Tasks A1 and A2 are unordered
 - Possible “schedules” (linear ordering of tasks): A1, A2 and A2, A1
 - All consistent schedules give the same result
- ❑ I/O await dependencies: out1 awaits in1, out2 awaits in2

Example Task Graph



- ☐ What are possible schedules consistent with precedence constraints?
- ☐ What are I/O await dependencies?

Task Graphs: Definition

- ❑ For a synchronous reactive component C with input vars I , output vars O , state vars S , and local vars L , reaction description is given by a set of tasks, and precedence edges $<$ over these tasks
- ❑ Each task A is specified by:
 1. Read-set R
 - must be a subset of $I \cup S \cup O \cup L$
 2. Write-set W
 - must be a subset of $O \cup S \cup L$
 3. Update: code to write vars in W based on values of vars in R
 - [Update] is a subset of $Q_R \times Q_W$

Requirements on Task Graph (1)

The precedence relation $<$ must be acyclic

- ❑ Notation: $A' <^+ A$ means that there is a path from task A' to task A in the task graph using precedence edges
- ❑ $<^+$ denotes the "transitive closure" of the relation $<$
- ❑ Task schedule: Total ordering A_1, A_2, \dots, A_n of all the tasks consistent with the precedence edges
 - If $A' < A$, then A' must appear before A in the ordering
 - Multiple schedules possible
- ❑ If $A' <^+ A$ then A' must appear before A in every schedule
- ❑ Acyclicity means that there is at least one task schedule

Requirements on Task Graph (2)

Each output variable is in the write-set of exactly one task

- ❑ If output y is in write-set of task A , then as soon as A executes the output y is available to the rest of the system
- ❑ If task A writes output y , then y awaits an input variable x , denoted $y \succ x$, if
 - either the task A reads x
 - or another task A' reads x such that $A' \prec^+ A$
- ❑ y awaits x means that y cannot be produced before x is supplied

Requirements on Task Graph (3)

Output/local variables are written before being read:

- If an output or a local variable y is in the read-set of a task A , then y must be in the write-set of some task A' such that $A' \prec^+ A$

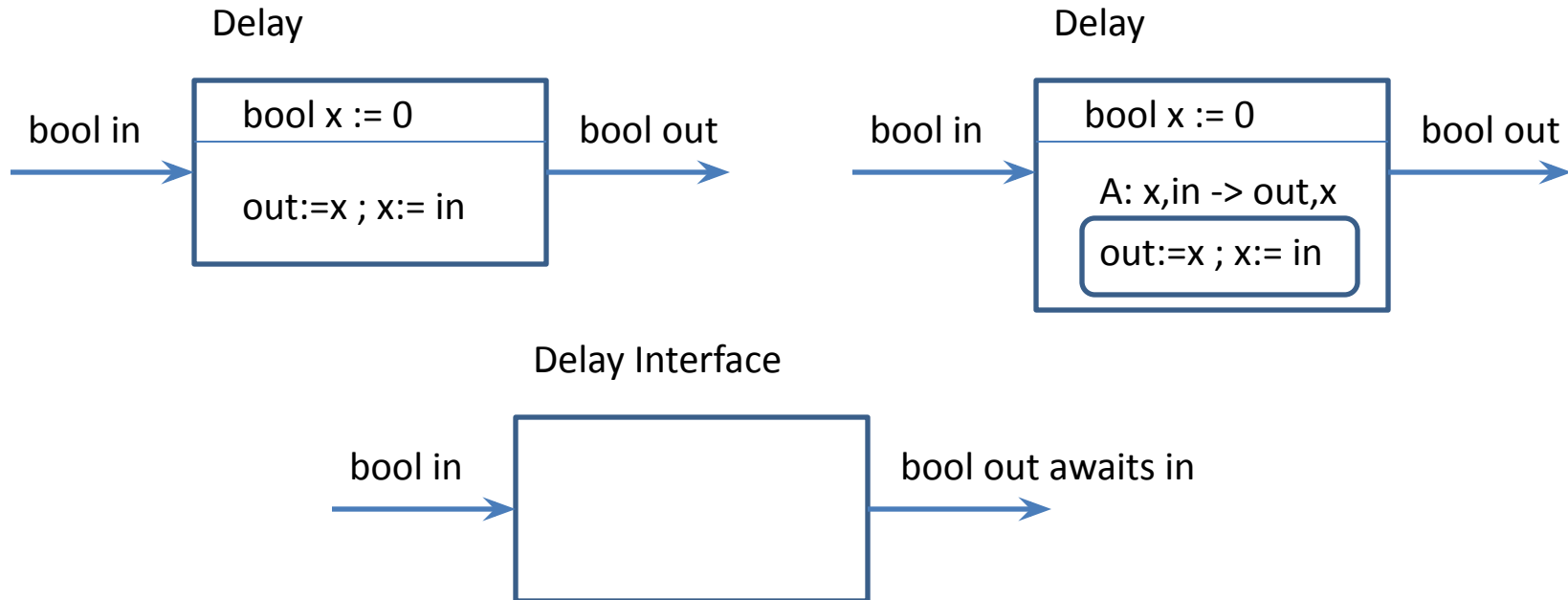
Requirements on Task Graph (4)

- ❑ Write-conflict between tasks A and A' :
 - There exists a variable that A writes and is either read or written by A'
- ❑ If A and A' have write-conflict, then the result depends on whether A executes before A' or vice versa.
 - Example: Update of A is $x := x+1$; Update of A' is $out := x$
- ❑ Requirement: Tasks with a write conflict must be ordered:
 - If tasks A and A' have write-conflict then either $A \prec^+ A'$ or $A' \prec^+ A$
- ❑ The set of reactions resulting from executing all the tasks do not depend on the task schedule

Properties of Tasks

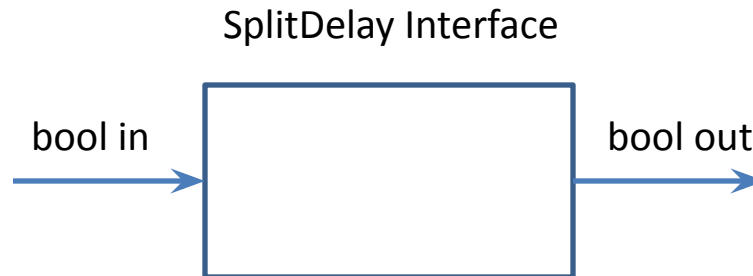
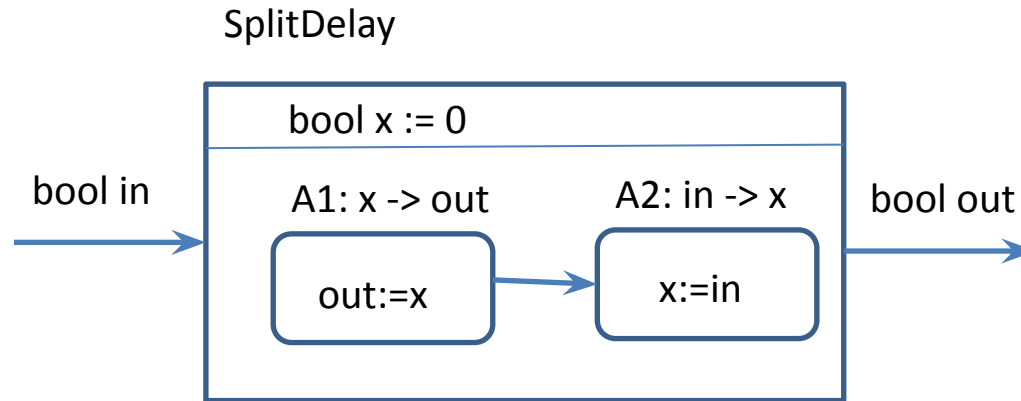
- ❑ Task $A = (R, W, \text{Update})$ is deterministic if for every value u in Q_R there is a unique value v in Q_W such that (u,v) is in $[\text{Update}]$
- ❑ If all tasks of a component are deterministic, what can we conclude about the component itself?
- ❑ Task $A = (R, W, \text{Update})$ is input-enabled if for every value u in Q_R there exists at least one value v in Q_W such that (u,v) is in $[\text{Update}]$
- ❑ If all tasks of a component are input-enabled, what can we conclude about the component itself?

Interfaces



□ Interface = Input variables, Output variables, Await dependencies

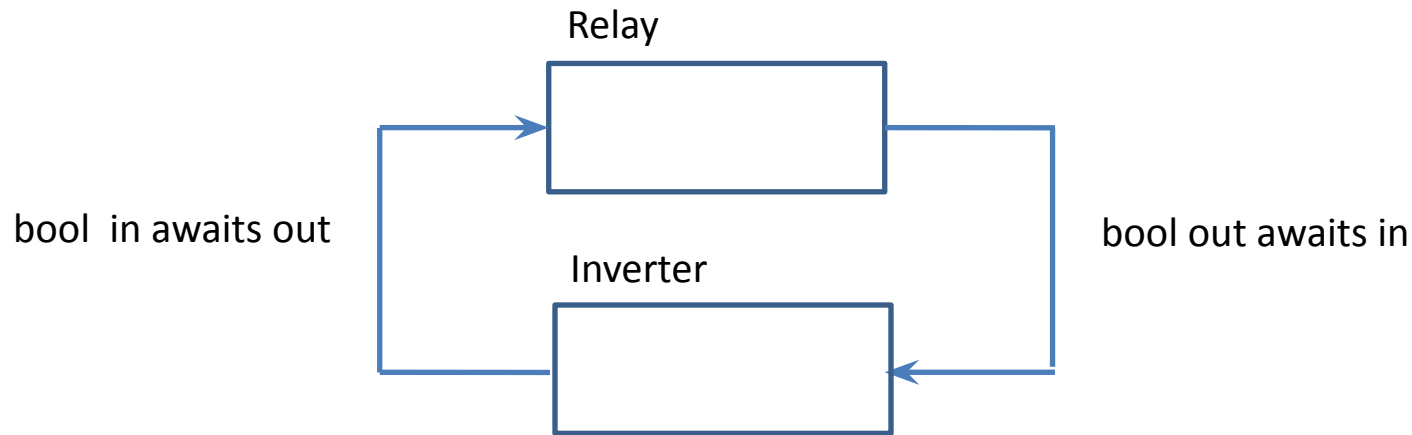
Interface: SplitDelay



Example Interface

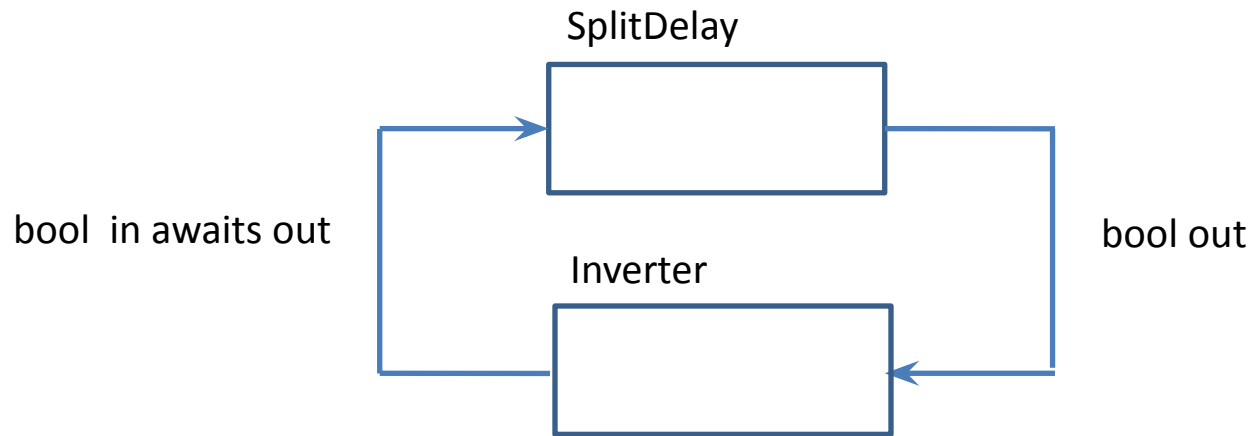


Back to Parallel Composition



- ❑ Relay and Inverter are not compatible since there is a cycle in their combined await dependencies

Composing SplitDelay and Inverter

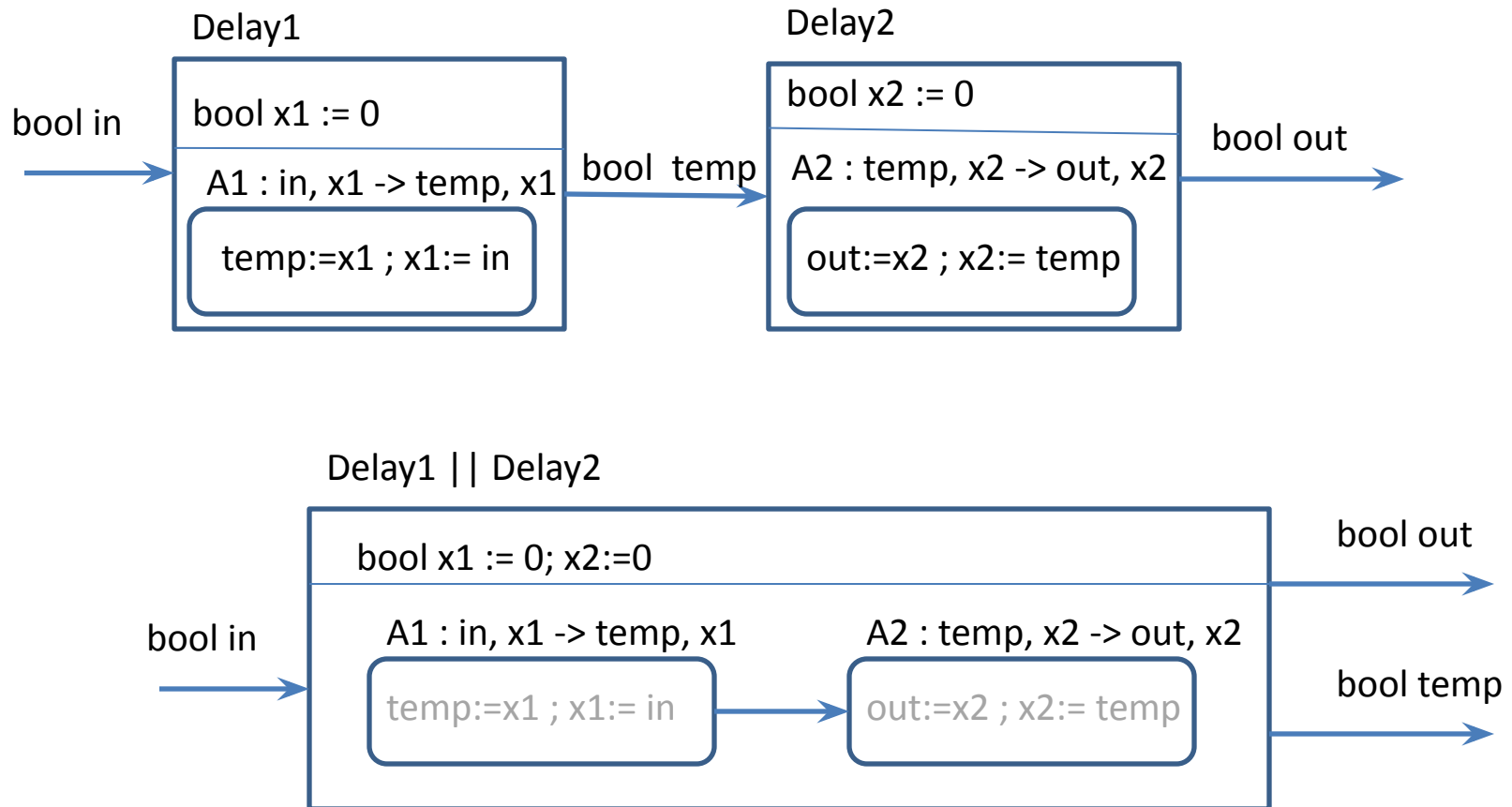


- ❑ SplitDelay and Inverter are compatible since there is no cycle in their combined await dependencies
- ❑ Note: Delay and Inverter are **not** compatible

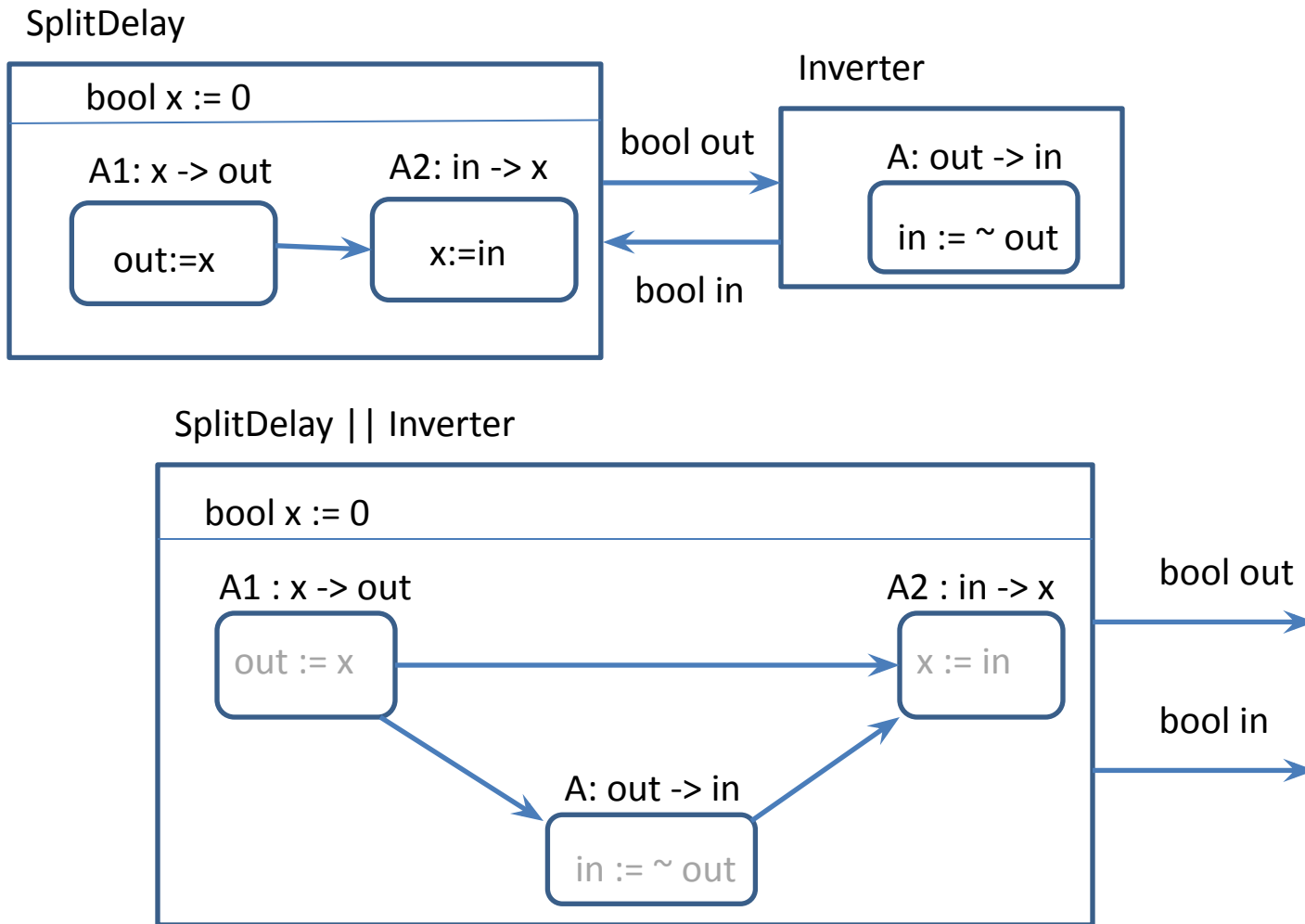
Component Compatibility Definition

- Given:
 - Component $C1$ with input vars $I1$, output vars $O1$, and awaits-dependency relation \succ_1
 - Component $C2$ with input vars $I2$, output vars $O2$, and awaits-dependency relation \succ_2
- The components $C1$ and $C2$ are compatible if
 - No common outputs: sets $O1$ and $O2$ are disjoint
 - The relation $(\succ_1 \cup \succ_2)$ of combined await-dependencies is acyclic
- Parallel Composition is allowed only for compatible components

Defining the Product



Composing SplitDelay and Inverter



Parallel Composition Definition

- Given compatible components $C1 = (I1, O1, S1, Init1, React1)$ and $C2 = (I2, O2, S2, Init2, React2)$, what's the definition of product $C = C1 \parallel C2$?
- We already defined I , O , S , and $Init$ for C
- Suppose $React1$ specified using local variables $L1$, set of tasks Π_1 , and precedence \prec_1 , and $React2$ given using local vars $L2$, set of tasks Π_2 , and precedence \prec_2
- Reaction description for product C has
 - Local variables $L1 \cup L2$
 - Set of tasks $\Pi_1 \cup \Pi_2$
 - Precedence edges: Edges in $\prec_1 + \text{Edges in } \prec_2 + \text{Edge between tasks } A1 \text{ and } A2 \text{ of different components if } A2 \text{ reads a var written by } A1$

Parallel Composition Definition

- ❑ Why is the parallel composition operation well-defined?
 - Can the new edges make task graph of the product cyclic?
- ❑ Recall: Await-dependencies among I/O variables of compatible components must be acyclic
- ❑ Proposition 2.1: Awaits compatibility implies acyclicity of product task graph
- ❑ Bottomline: Interfaces capture enough information to define parallel composition in a consistent manner
- ❑ Aside: possible to define more flexible (but complex) notions of awaits dependencies

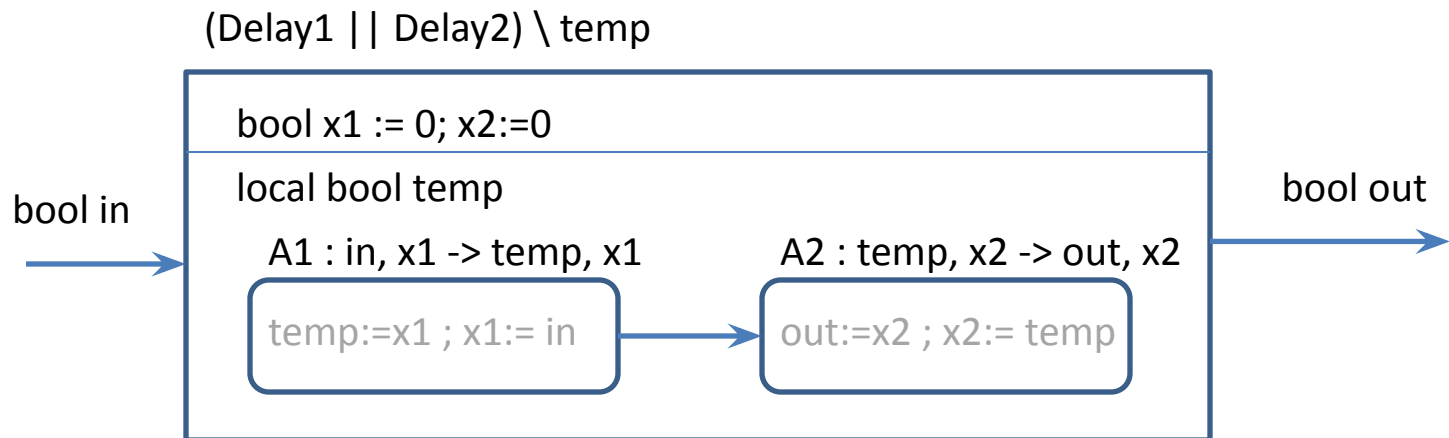
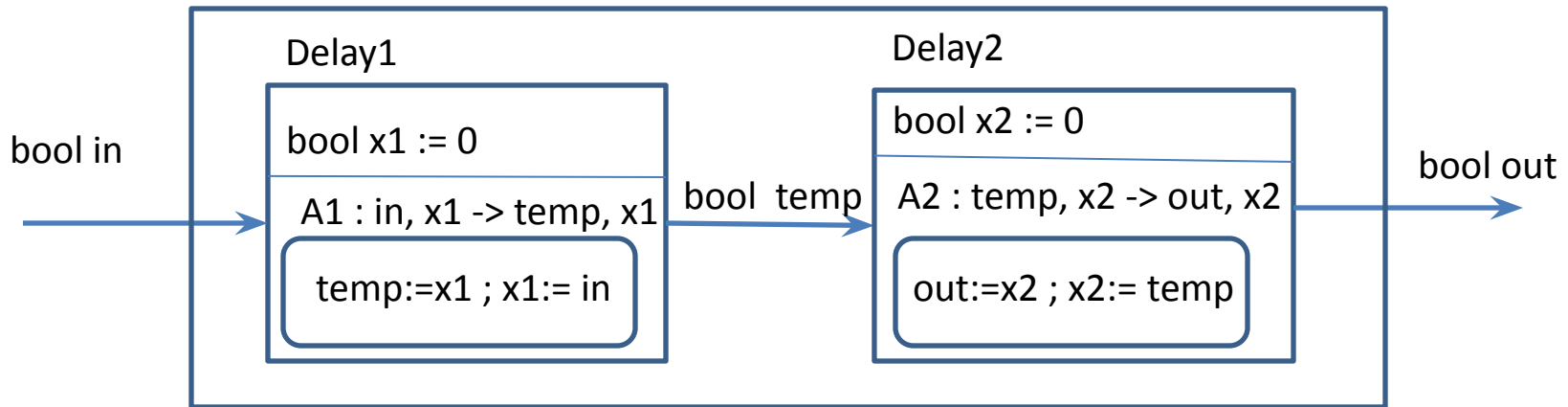
Properties of Parallel Composition

- ❑ Commutative: $C1 \parallel C2$ is same as $C2 \parallel C1$
- ❑ Associative: Given $C1, C2, C3$, all of $(C1 \parallel C2) \parallel C3$, $C1 \parallel (C2 \parallel C3)$, $(C1 \parallel C3) \parallel C2$, ... give the same result
 - If compatibility check fails in one case, will also fail in others
 - Bottomline: Order in which components are composed does not matter
- ❑ If both $C1$ and $C2$ are finite-state, then so is product $C1 \parallel C2$
 - If $C1$ has $n1$ states and $C2$ has $n2$ states then the product has $(n1 \times n2)$ states
- ❑ If both $C1$ and $C2$ are deterministic, then so is product $C1 \parallel C2$
- ❑ If both $C1$ and $C2$ are event-triggered, is it guaranteed that the product $C1 \parallel C2$ is event-triggered??

Output Hiding

- Given a component C , and an output variable y , the result of hiding y in C , written as $C \setminus y$, is basically the same component as C , but y is no longer an output variable, and becomes a local variable
 - Not available to the outside world
 - Useful for limiting the scope (encapsulation)

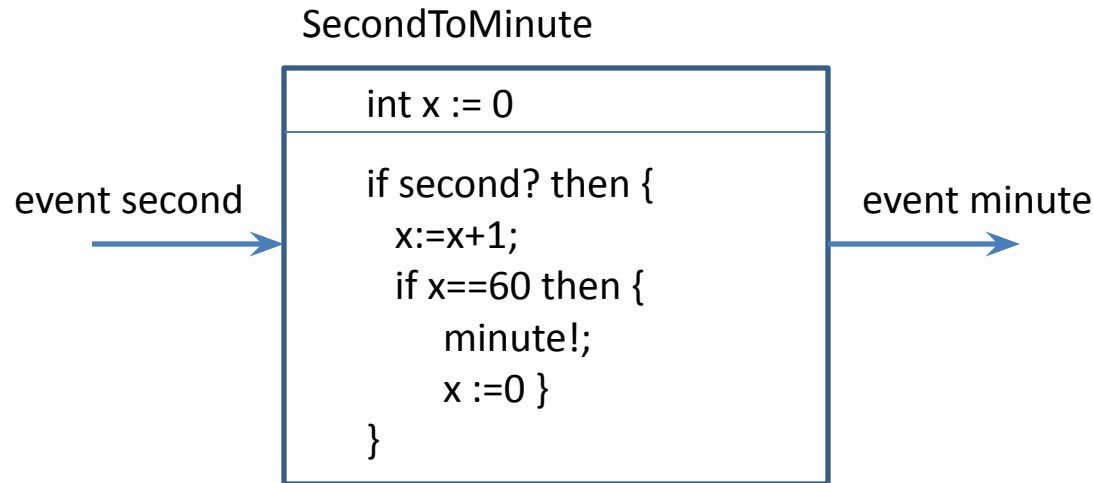
DoubleDelay



Second-To-Minute

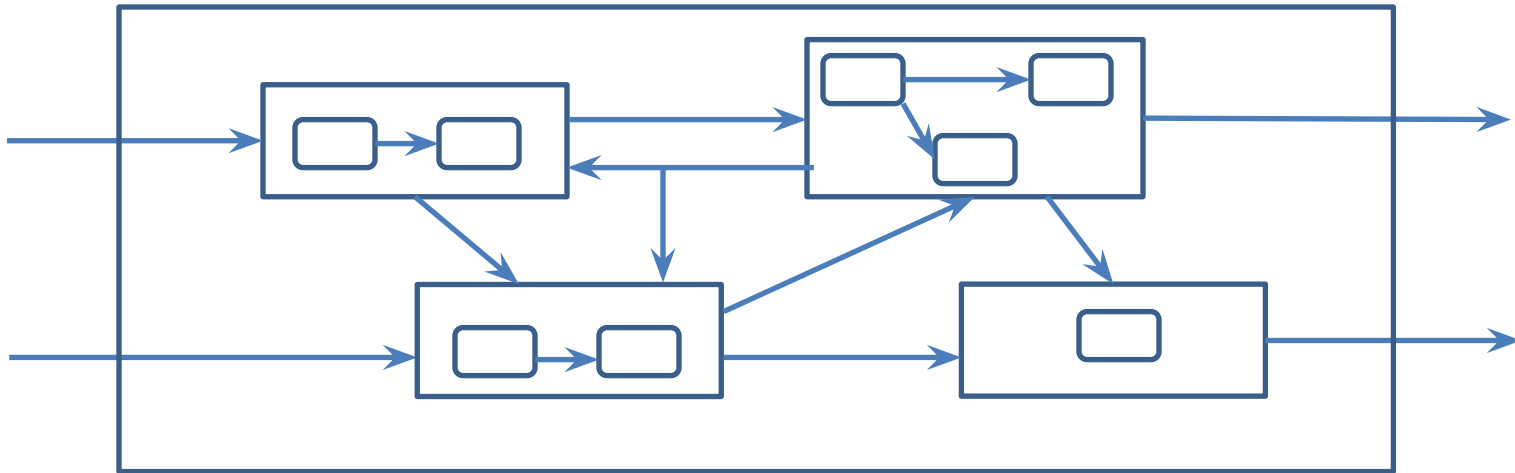
Desired behavior (spec):

Issue the output event every 60th time the input event is present



- Design the component Second-To-Hour such that it issues its output every 3600th time its input event is present

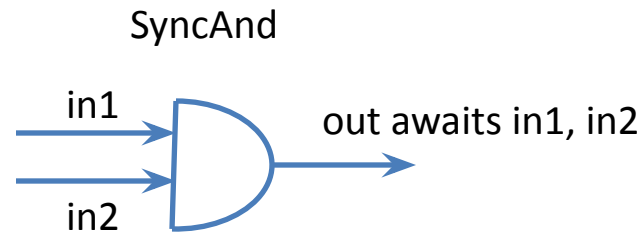
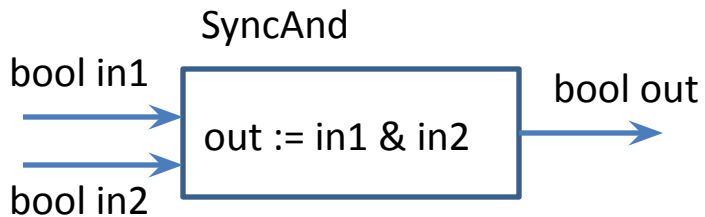
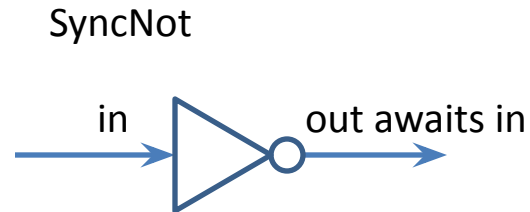
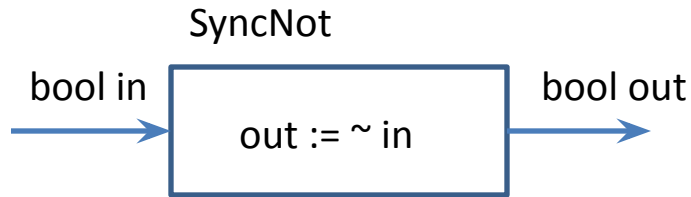
Synchronous Block Diagrams



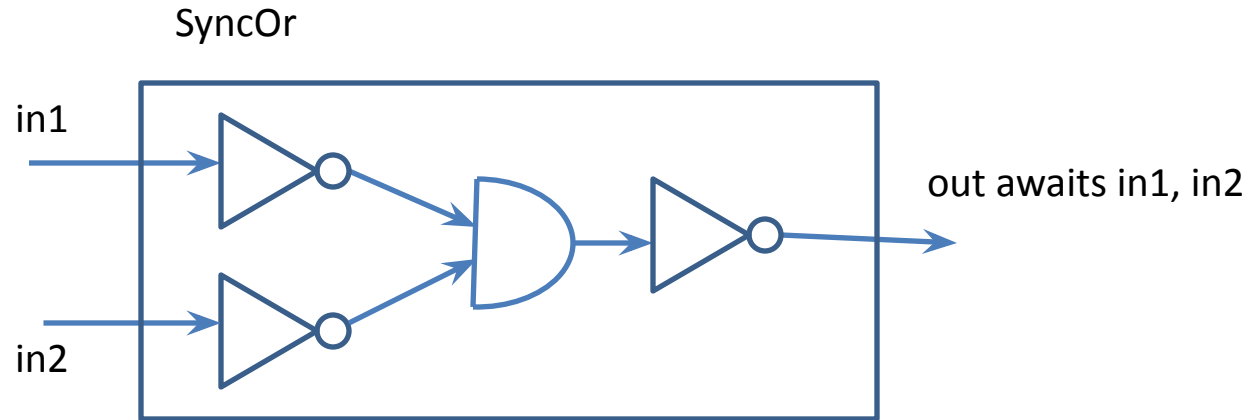
Bottom-Up Design

- ☐ Design basic components
- ☐ Compose existing components in block-diagrams to build new components
- ☐ Maintain a library of components, and try to reuse at every step
- ☐ Canonical example: Synchronous circuits

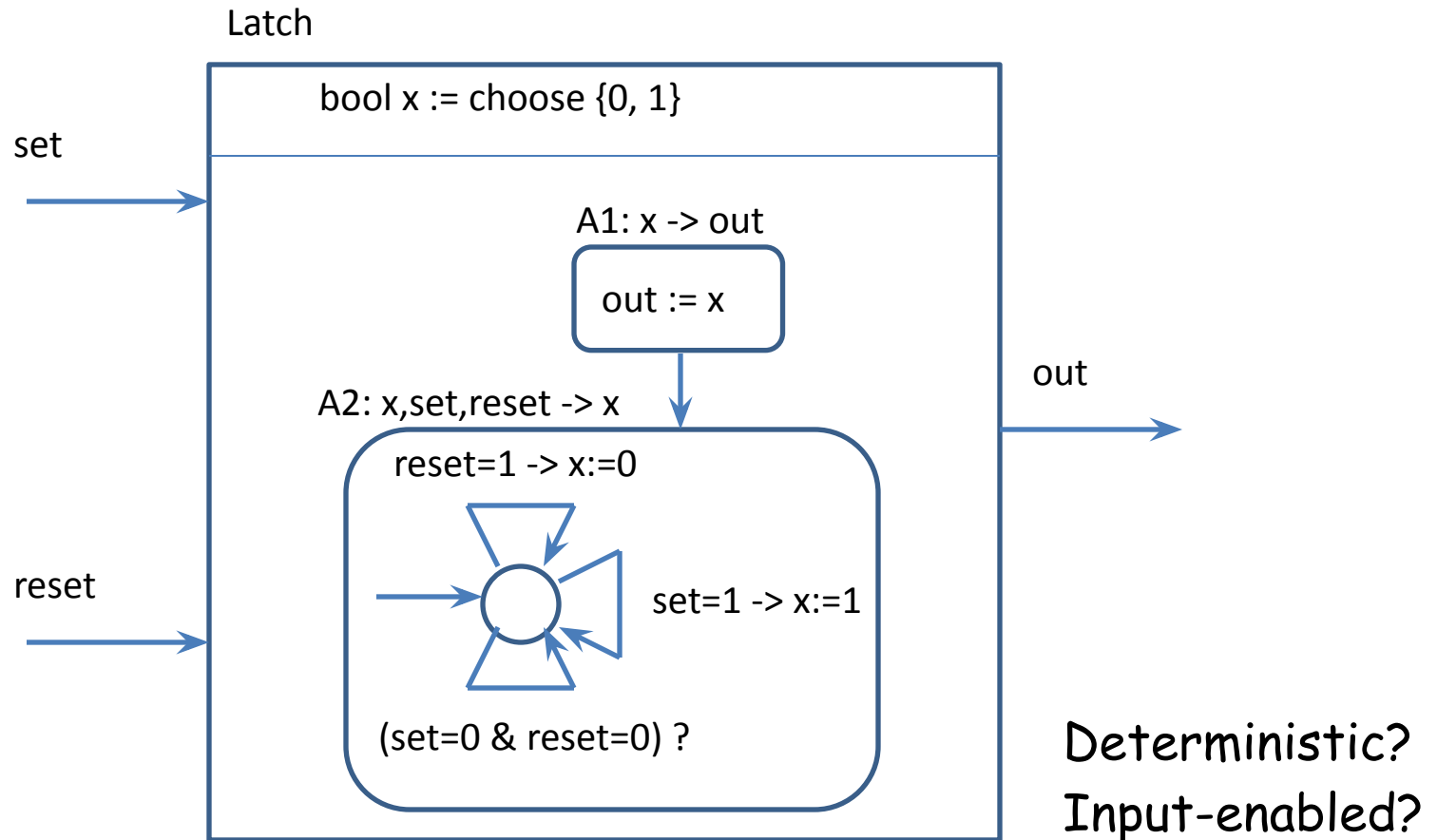
Combinational Circuits



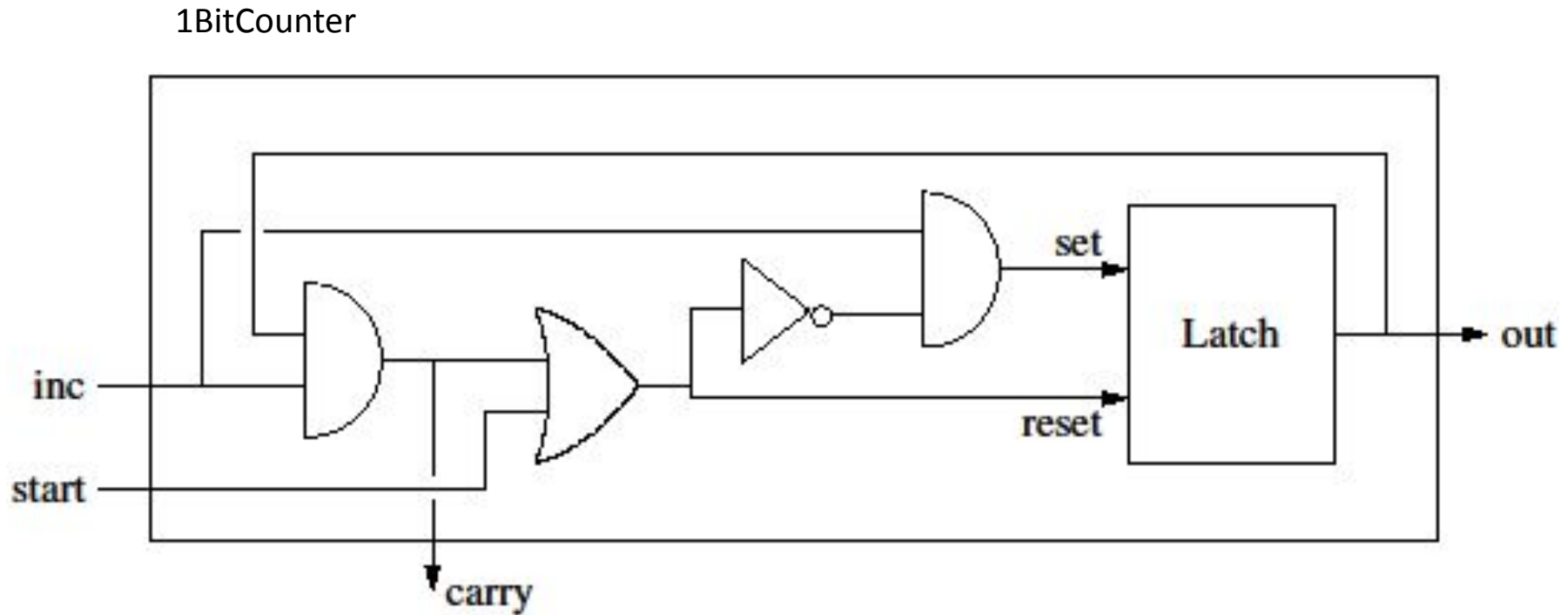
Design OR gate



Synchronous Latch



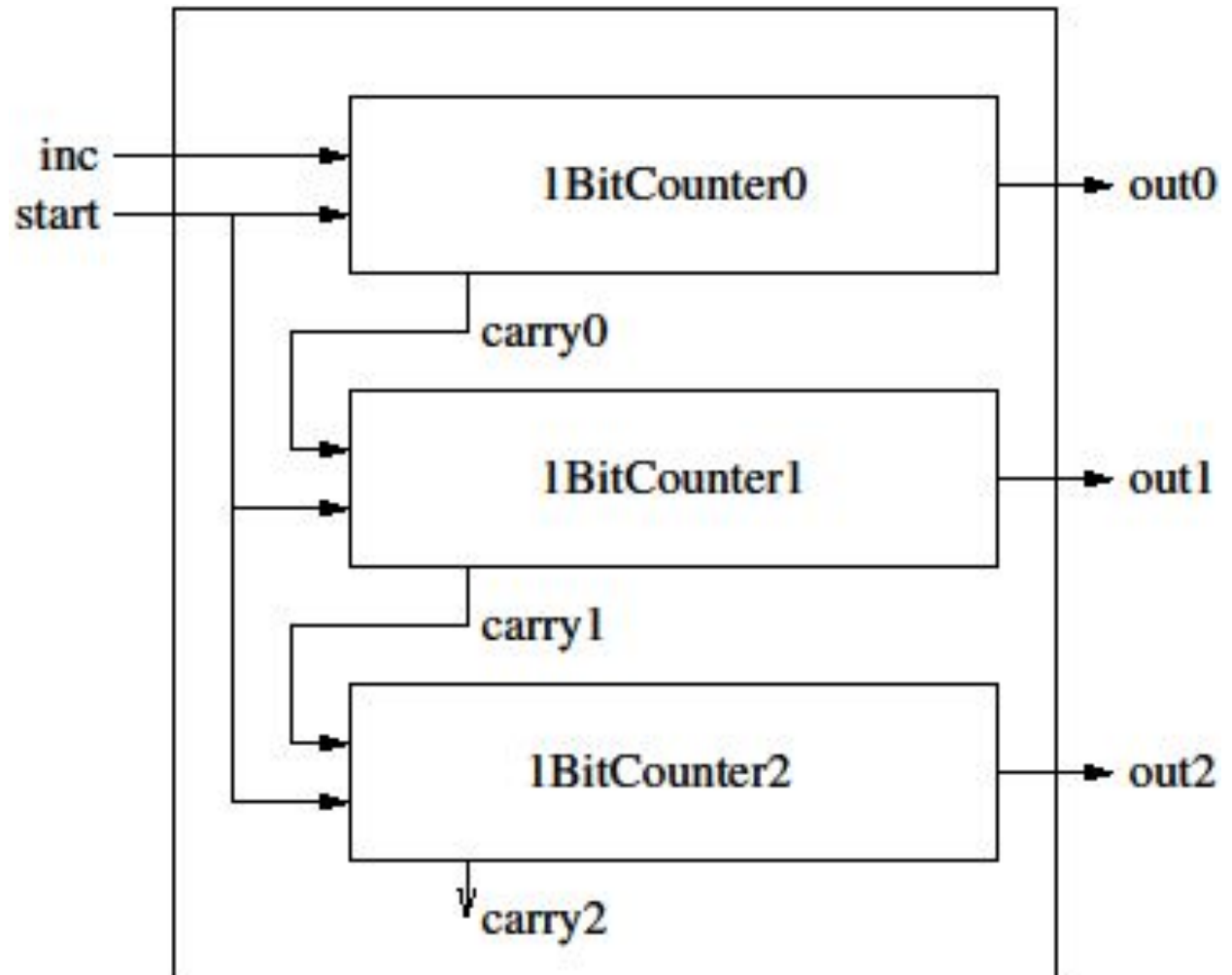
Designing Counter Circuit (1)



□ Are await-dependencies acyclic?

Designing Counter Circuit (2)

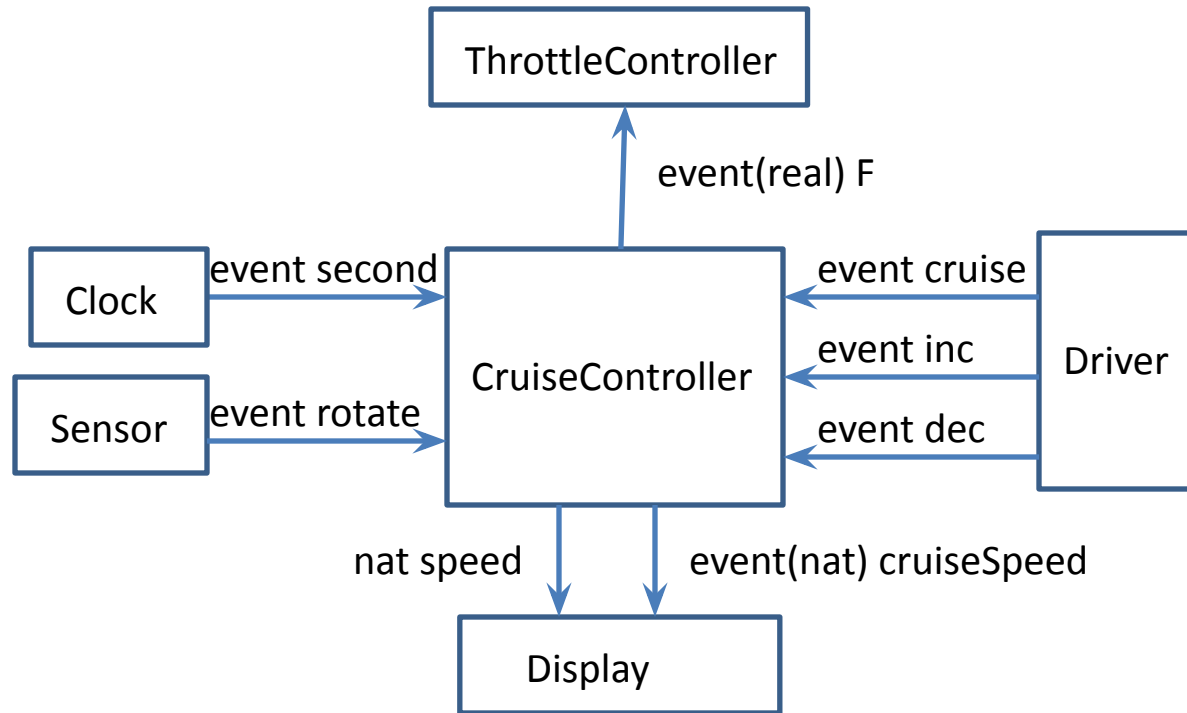
3BitCounter



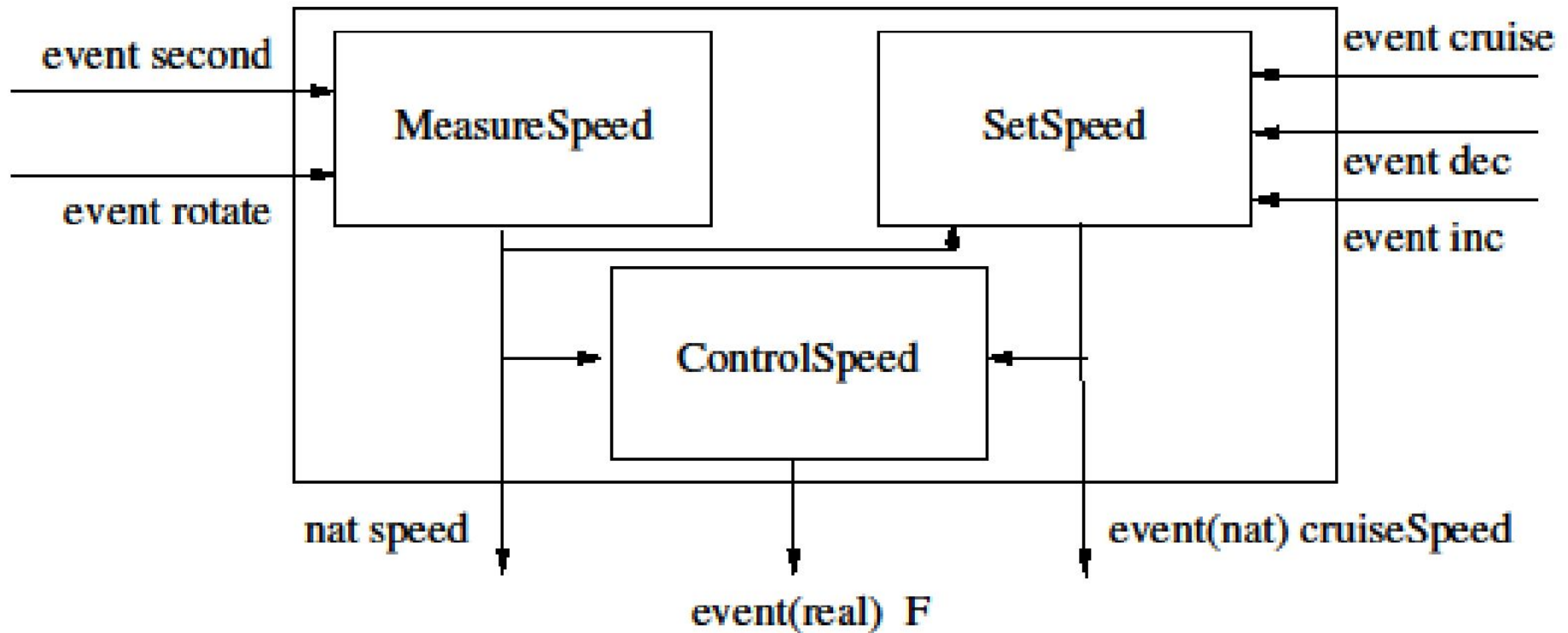
Top-Down Design

- ❑ Starting point: Inputs and outputs of desired design C
- ❑ Models/assumptions about the environment in which C operates
- ❑ Informal/formal description of desired behavior of C
- ❑ Example: Cruise Controller

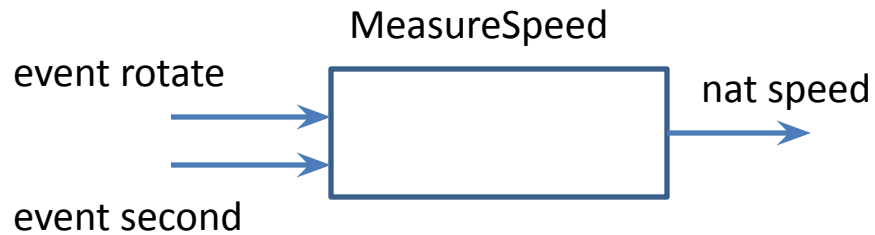
Top-Down Design of a Cruise Controller



Decomposing CruiseController

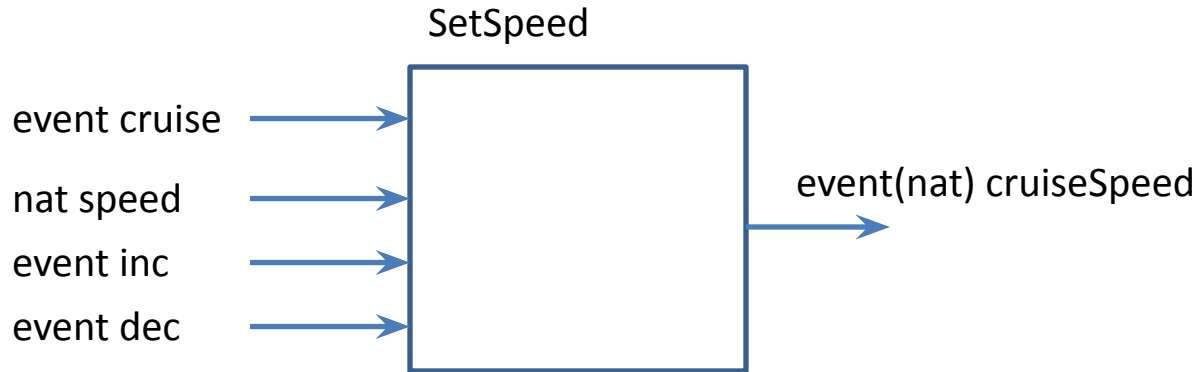


Tracking Speed



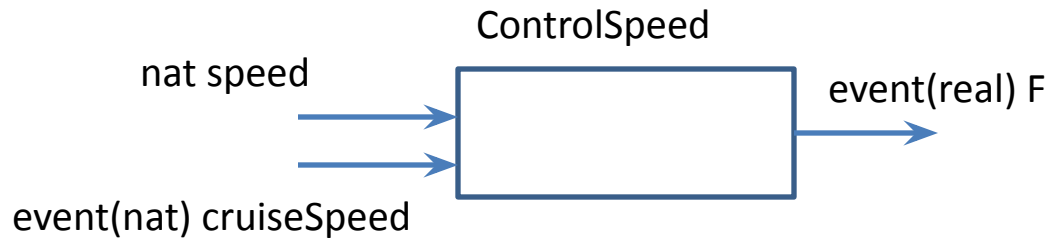
- ❑ Inputs: Events rotate and second
- ❑ Output: current speed
- ❑ Computes the number of rotate events per second (see notes)

Tracking Cruise Settings



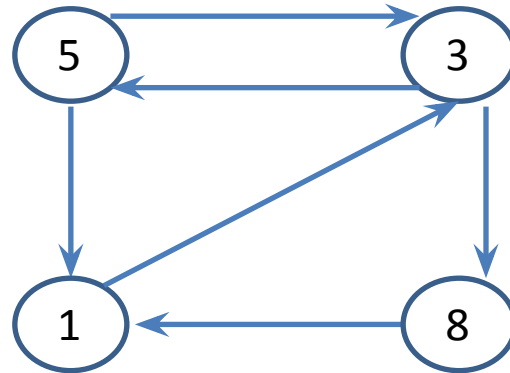
- ☐ Inputs from the driver: Commands to turn the cruise-control on/off and to increment/decrement desired cruising speed from driver
- ☐ Input: Current speed
- ☐ Output: Desired cruising speed
- ☐ What assumptions can we make about simultaneity of events?
- ☐ Should we include safety checks to keep desired speed within bounds?

Controlling Speed



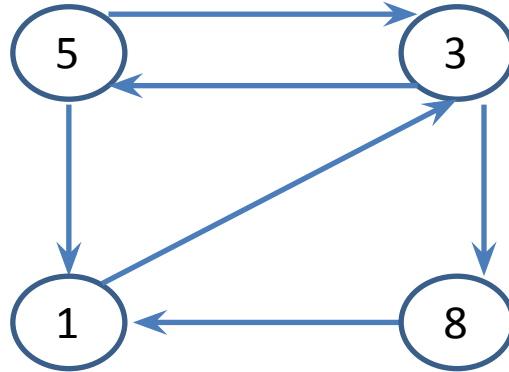
- ❑ Inputs: Actual speed and desired speed
- ❑ Output: Pressure on the throttle
- ❑ Goal: Make actual speed equal to the desired speed (while maintaining key physical properties such as stability)
- ❑ Design relies on theory of dynamical systems (Chapter 6)

Synchronous Networks



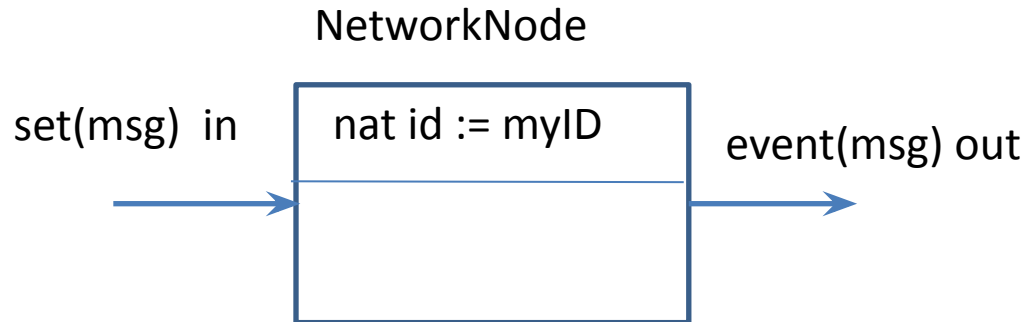
- ☐ Time divided into slots, with all nodes synchronized
- ☐ In one round, each node can get a message from each neighbor
- ☐ Design abstraction for simplicity
- ☐ Some implementation platforms directly support such a "time-triggered" network: WirelessHART (control), CAN (automotive)

Modeling Synchronous Networks



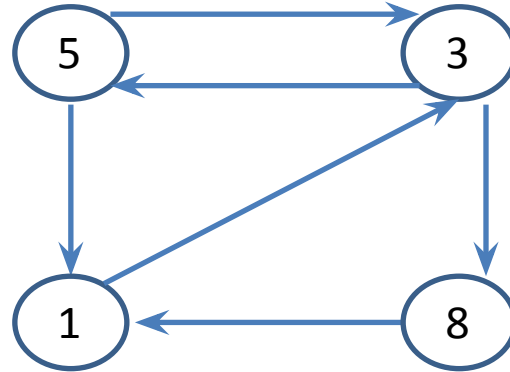
- ❑ Assume: Each link is directed and connects two nodes
 - Alternative: Broadcast communication (everyone can listen)
- ❑ Assume: Communication is reliable
 - Alternative: Messages may be lost, collisions in broadcast
- ❑ Network is a directed graph
 - Each link can carry one message in each slot

Component for a network node

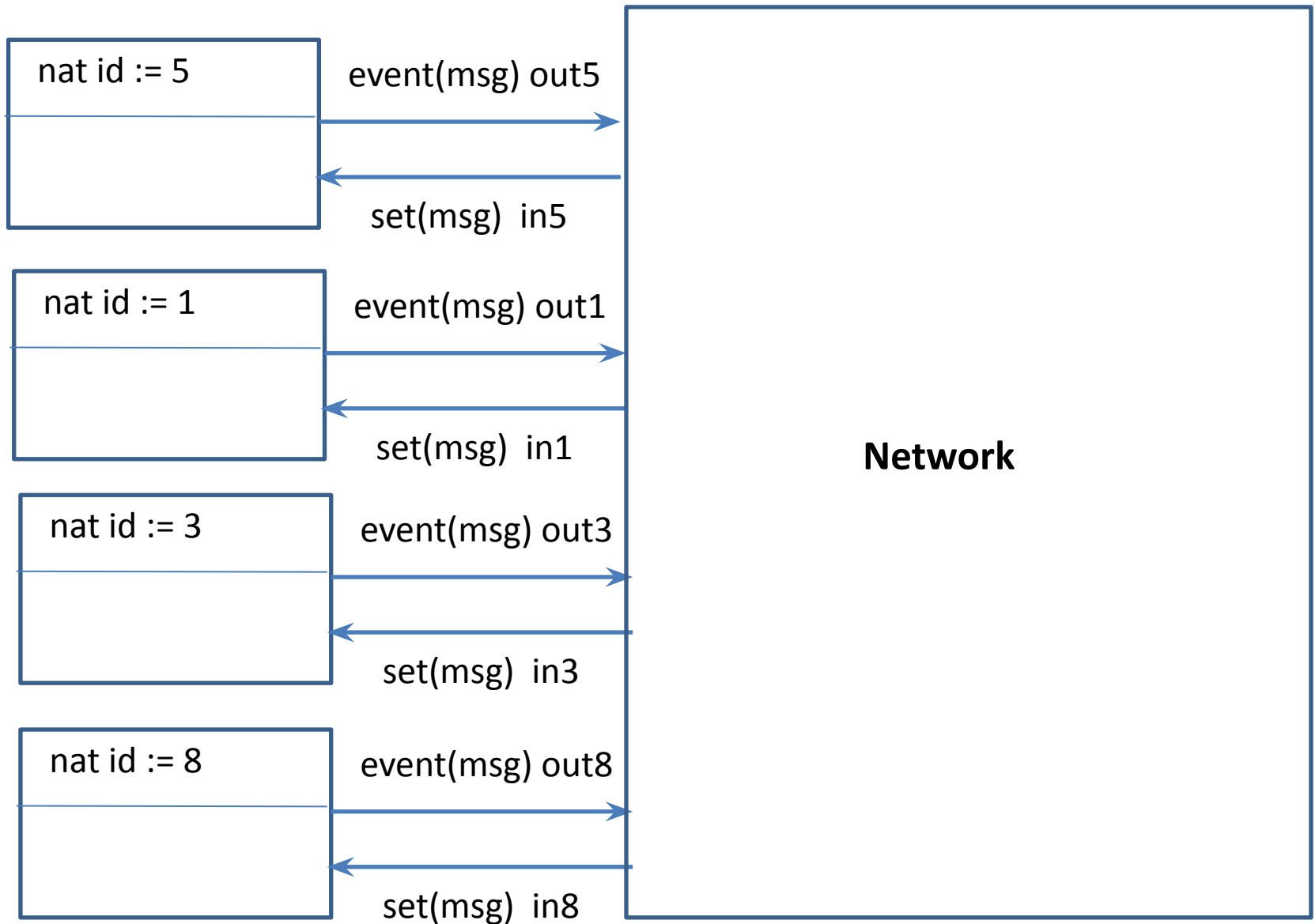


- ❑ A node does not know network topology
 - Each node has unique identifier, myID
 - Does not know which nodes it is connected to
 - Useful for "network identification" problems
- ❑ Interface for each node
 - Output is an event carrying msg (may be absent in some rounds)
 - Input is a set of messages (delivered by the network)
 - Output should not await input

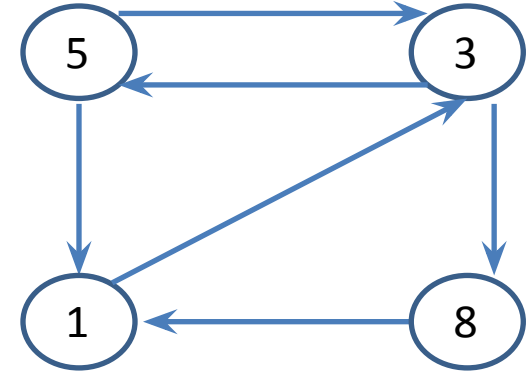
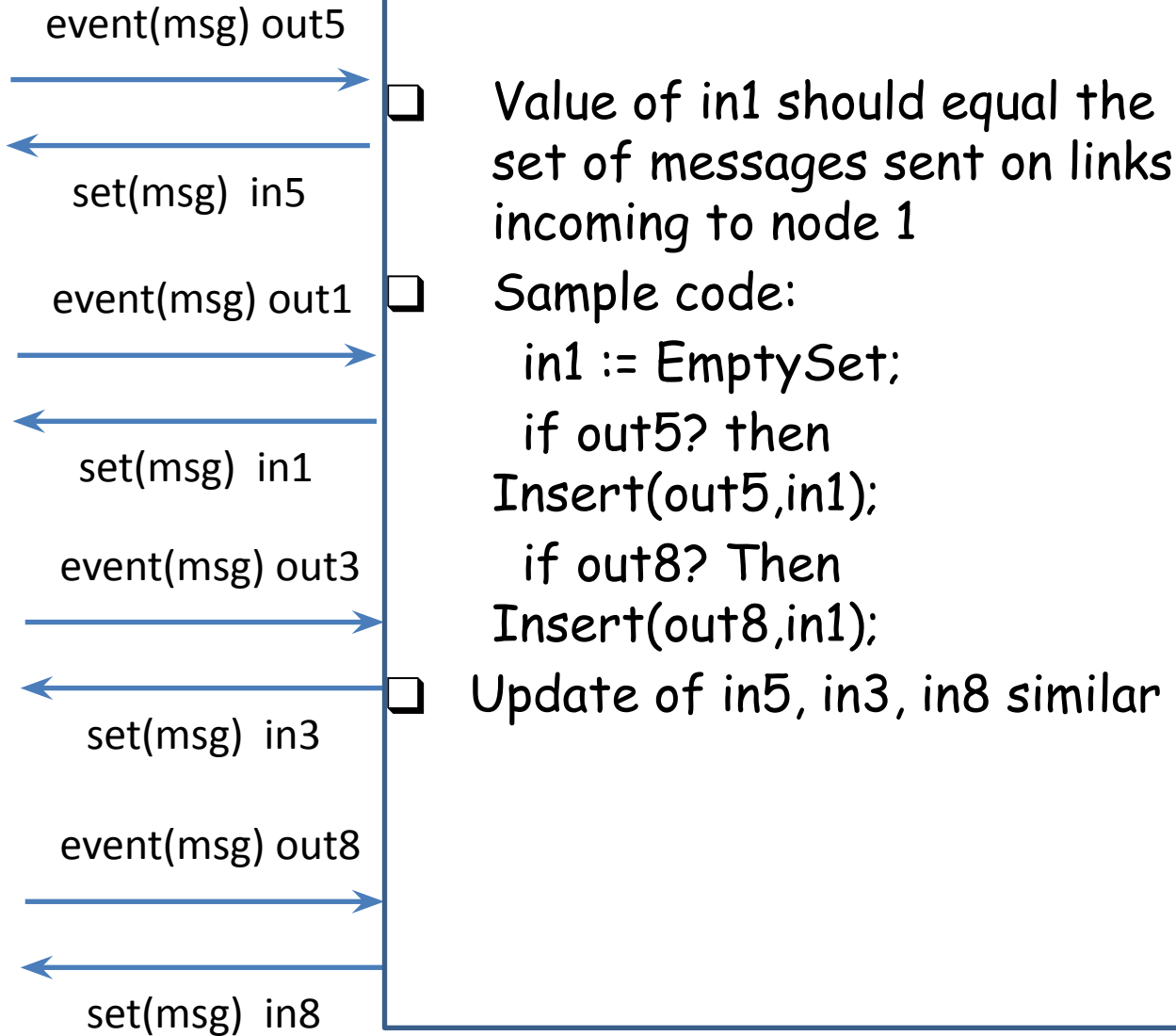
Modeling Synchronous Networks



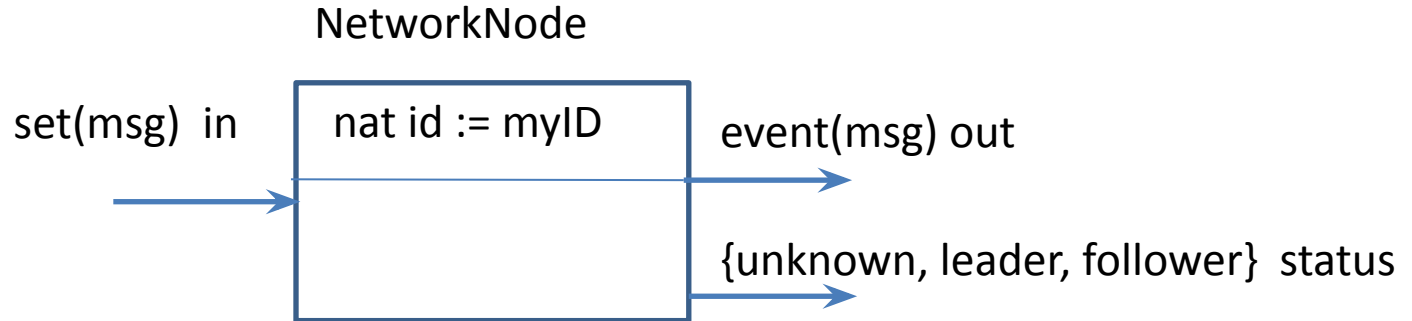
- ❑ Description of each node does not depend on the network
- ❑ Network itself is modeled as a synchronous component
- ❑ Description of Network depends on the network graph
- ❑ Input variables: for each node n , out_n of type $event(msg)$
- ❑ Output variables: for each node n , in_n of type $set(msg)$
- ❑ Network is a combinational component (simply routes messages)



Network



Leader Election

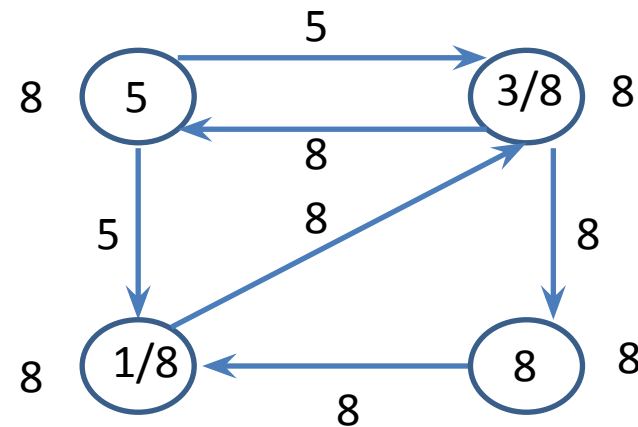
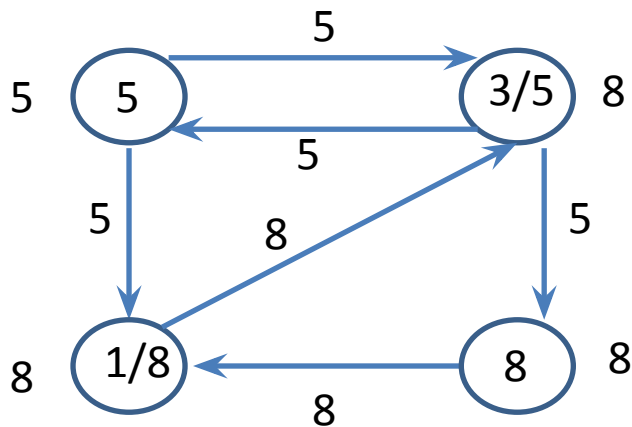
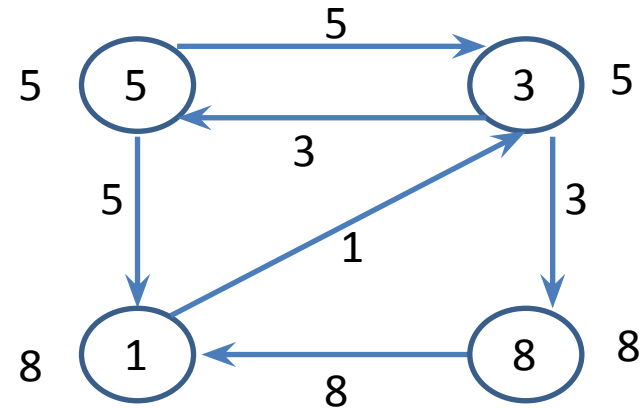
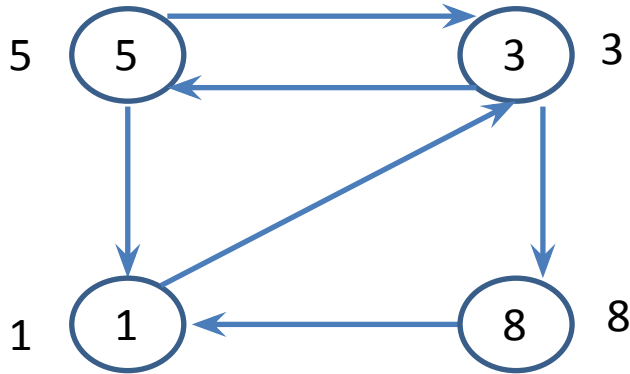


- ❑ Classical coordination problem: Elect a unique node as a leader
 - Exchange messages to find out which nodes are in network
 - Output the decision using the variable status
- ❑ Requirements
 1. Eventually every node sets status to either leader or follower
 2. Only one node sets status to leader

Leader Election: Flooding Algorithm

- ❑ Goal: Elect the node with highest identifier as the leader
- ❑ Strategy: Transmit highest id you have encountered so far to your neighbors
- ❑ Implementation:
 - Maintain a state variable, id, initialized to your own identifier
 - In each round, transmit value of id on output
 - Receive input values from the network
 - If a value higher than id received, then update id

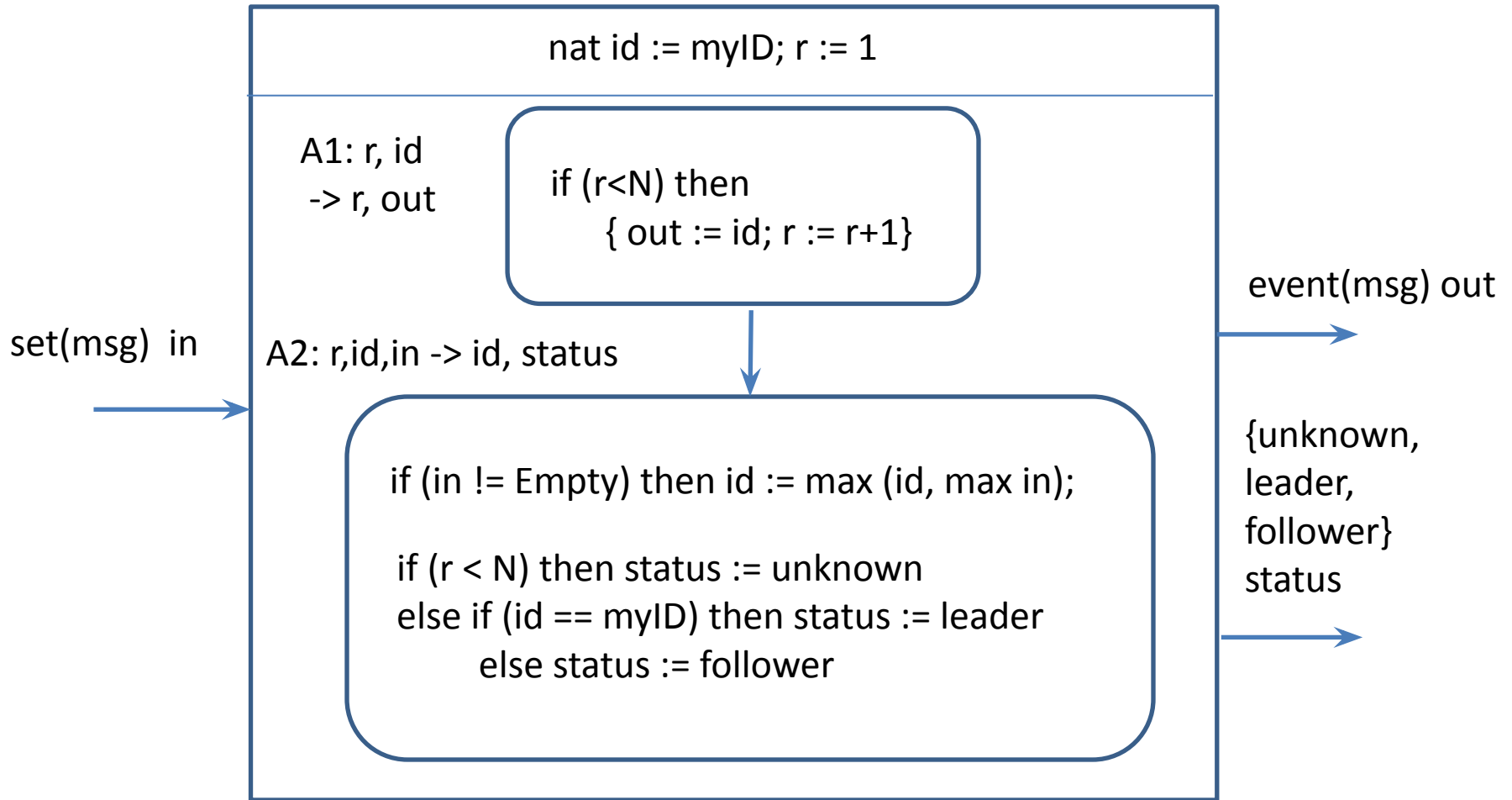
Execution of Leader Election



Leader Election

- ❑ When should a node stop and make a decision?
- ❑ When it knows that enough rounds have elapsed for message from every node to reach every other node
- ❑ Correctness depends on following assumptions:
 1. Network is strongly connected: for every pair of nodes m and n , there is a directed path from node m to node n
 2. Each node knows an upper bound N on total number of nodes
- ❑ Implementation of decision rule:
 - Maintain a state variable r to count rounds, initially 1
 - In each round, r is incremented
 - When $r = N$, decide
- ❑ What should the decision be?

Node Component for Leader Election



Leader Election

- ☐ Does a node really have to wait for N rounds?
- ☐ If a node receives a value higher than its own identifier, can it stop participating (i.e. not transmit any more messages)?
- ☐ Does a node have to transmit in each round? When can it choose to skip a round without affecting correctness?