CS:4980
Foundations of Embedded Systems
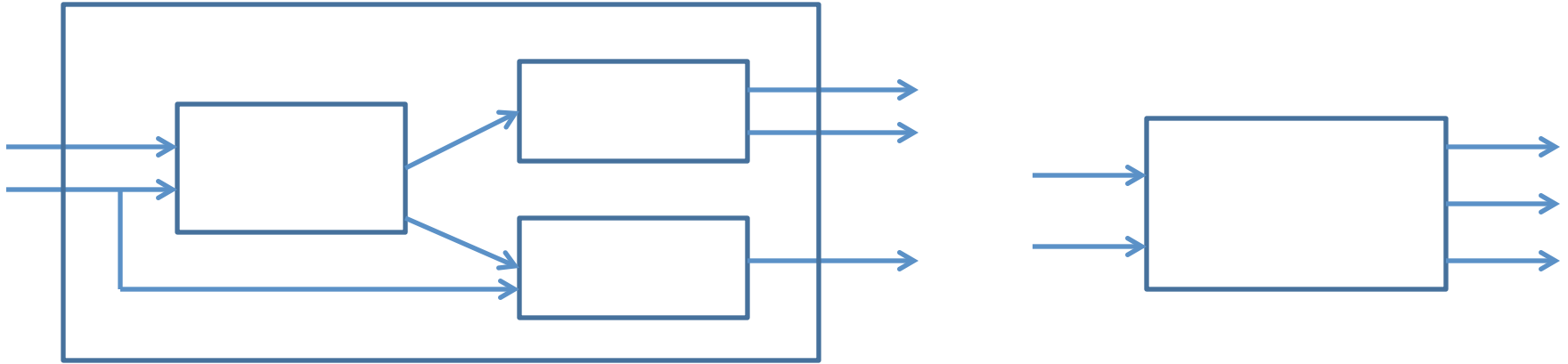
Synchronous Model
Part I

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

❑ **Functional model** of computation (classical one):

- 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
- Canonical model: Turing machines
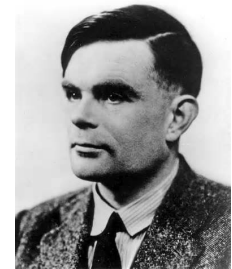
❑ **Reactive model** of computation:

- System continually interacts with its environment via inputs and outputs
- Desired behavior described by sequences of observed input/output interactions
- Example: cruise controller in a car

Sequential vs. Concurrent Computation

❑ Sequential model of computation (classical):

- A computation is a sequence of instructions executed one at a time
- Well understood and canonical model: Turing machines

❑ Concurrent model of 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
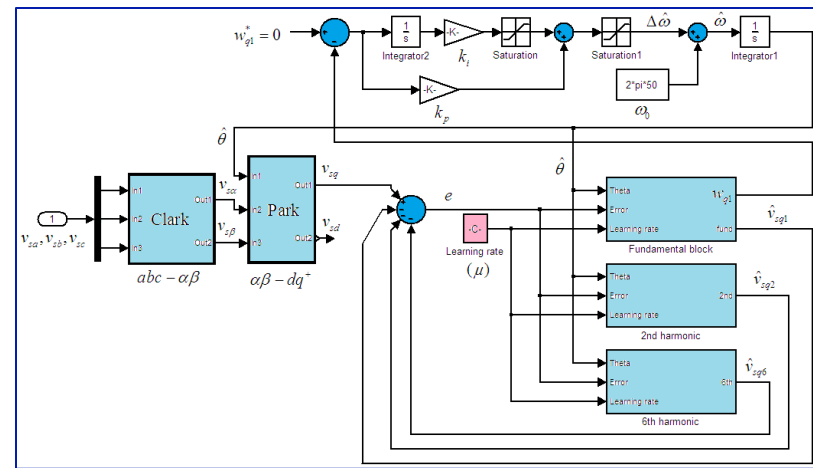
❑ All components execute in a sequence of (logical) rounds, in lock-step

❑ 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

```
bool in  ┌──────────────────────────┐  bool out
────────►│      bool x := 0         │────────►
         ├──────────────────────────┤
         │   out := x ; x := in     │
         └──────────────────────────┘
```
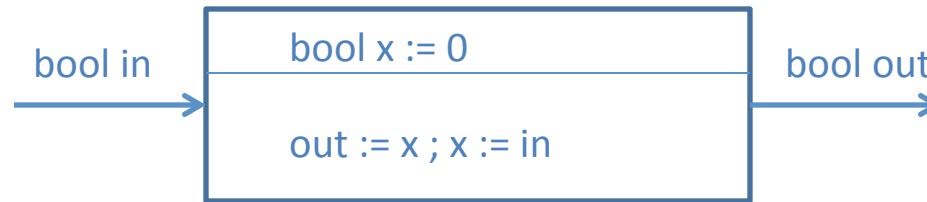
❑ *Input* variable: in of Boolean type ({0,1})

❑ *Output* variable: out of Boolean type

❑ *State* variable: x of Boolean type

❑ *Initialization* of state variables (x := 0)

❑ *Execution*: in each round, in response to an input,
- produce output (out := x) and
- update state (x := in)

Delay: Round-based Execution
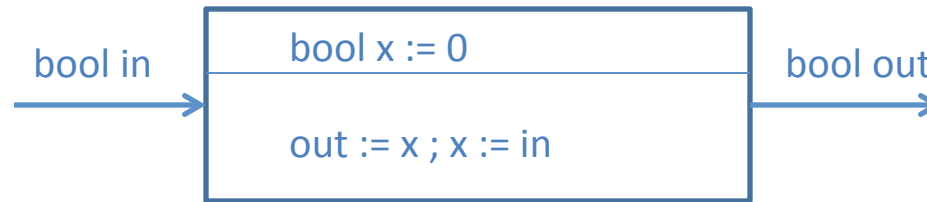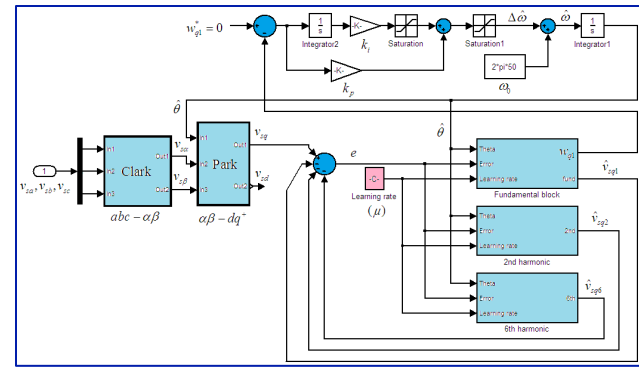

1. Initialize state x to 0

2. Repeatedly execute rounds. In each round:
   a. Read current value of the input variable in
   b. Execute the update code to produce output out and change state

Sample execution:

$$0 \xrightarrow{\ 1\,/\,0\ } 1 \xrightarrow{\ 1\,/\,1\ } 1 \xrightarrow{\ 0\,/\,1\ } 0 \xrightarrow{\ 0\,/\,0\ } 0 \xrightarrow{\ 1\,/\,0\ } 1 \dashrightarrow$$
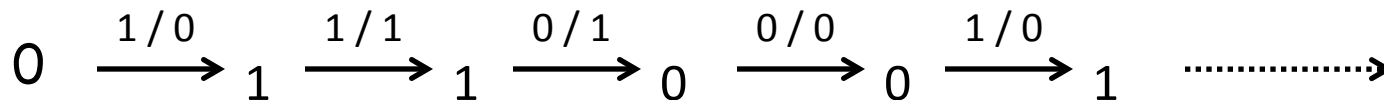
Synchrony Hypothesis

❏ Assumption: Time needed to execute the update code is negligible compared to delay between successive input arrivals

❏ Logical abstraction:

  ▪ Execution of update code takes zero time
  ▪ Reception of inputs and production of outputs occur simultaneously

❏ Composition: when multiple components are composed, all execute synchronously and simultaneously

❏ Implementation: must ensure that this design-time assumption is valid

Components in an Automobile
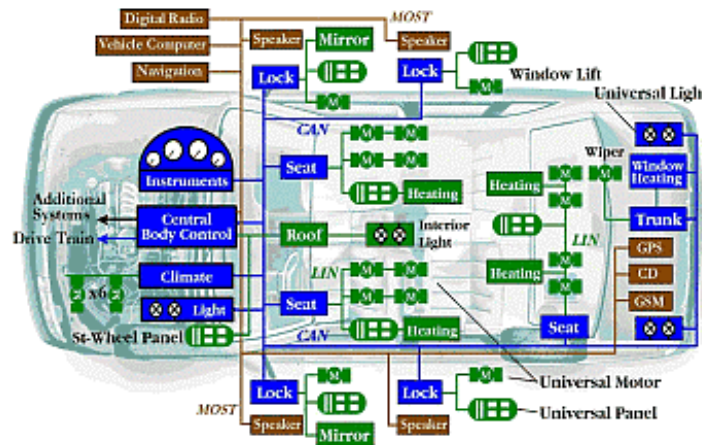
❑ Components need to communicate and coordinate over a shared bus

❑ 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

Model Definition

❑ Syntax: How to describe a component?

- Variable declarations, types, state updates, …

❑ Semantics: What does the description mean?

- Defined using mathematical concepts such as sets, functions, …

❑ Formal foundations: 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

Model Definition

Our modeling language: Synchronous Reactive Components

- Representative of many academic proposals
- Foundations of Industrial-strength synchronous languages: Esterel, Lustre, VHDL, Verilog, Stateflow, …

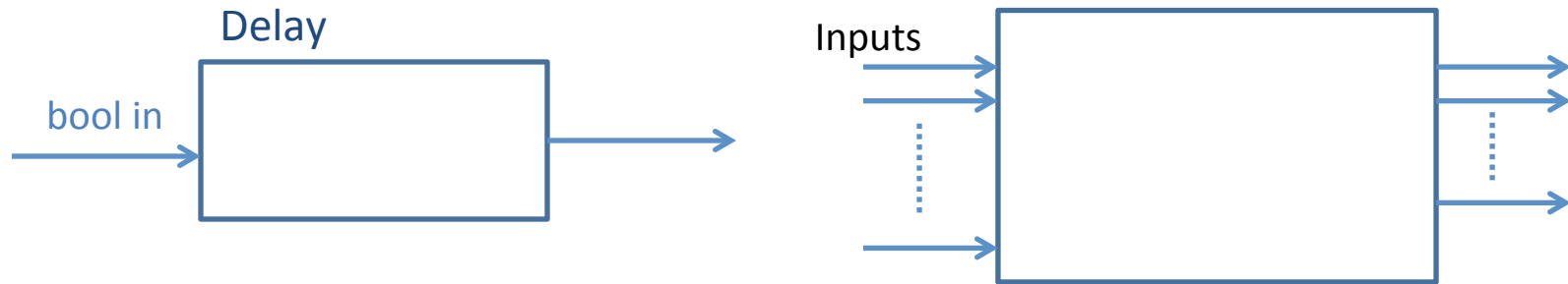# Synchronous Reactive Component

Delay

bool in → [bool x := 0 / out :=x ; x := in] → bool out

Inputs

State variables:
    Declaration + Initialization

Update code:
    To be executed in each round

Outputs
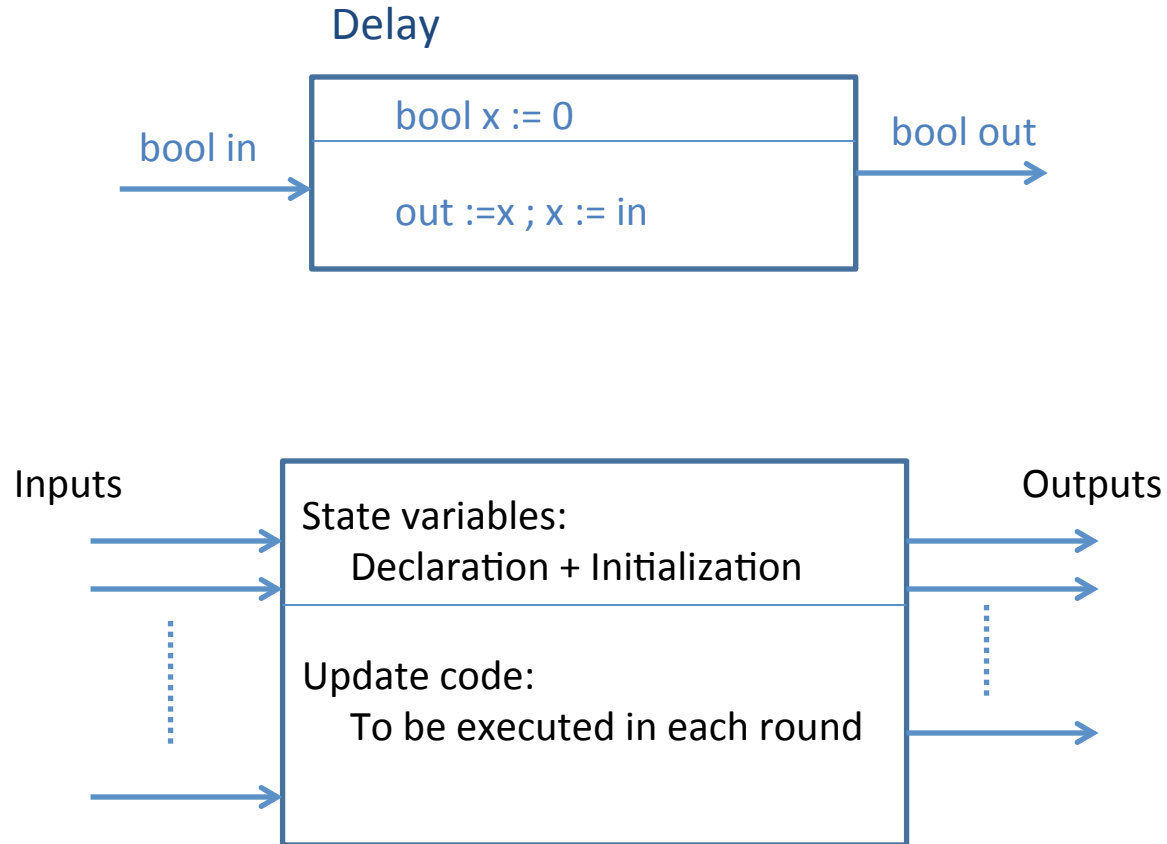
SRC Definition (1): Inputs

Delay

bool in

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 { {in := 0}, {in := 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
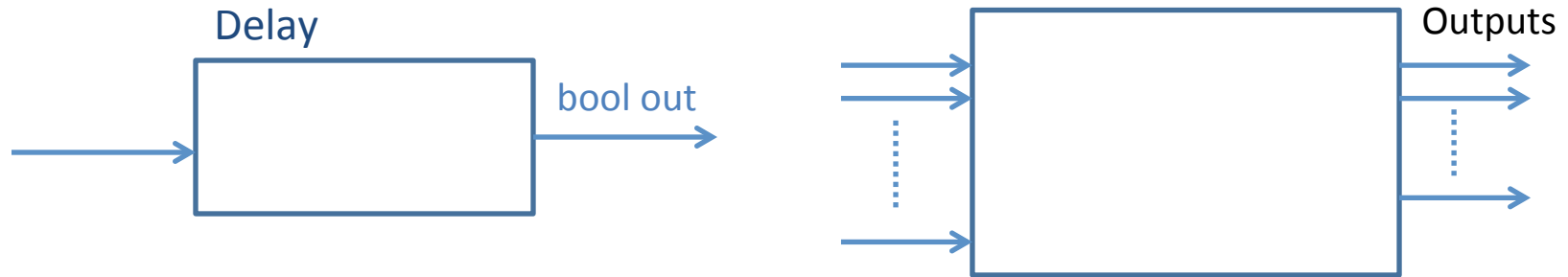
❑ Each component has a set O of typed output variables

❑ 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 { {out := 0}, {out := 1} }

SRC Definition (3): States
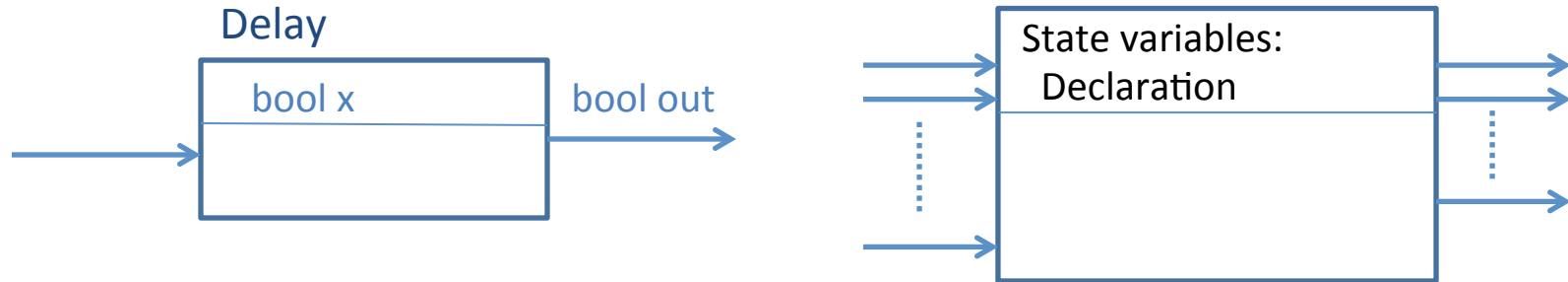
❑ Each component has a set S of typed state variables

❑ 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 { {x := 0}, {x := 1} }

❑ State is internal and maintained across rounds

SRC Definition (4): Initialization

Delay

bool x := 0   bool out

State variables:
  Declaration + Initialization
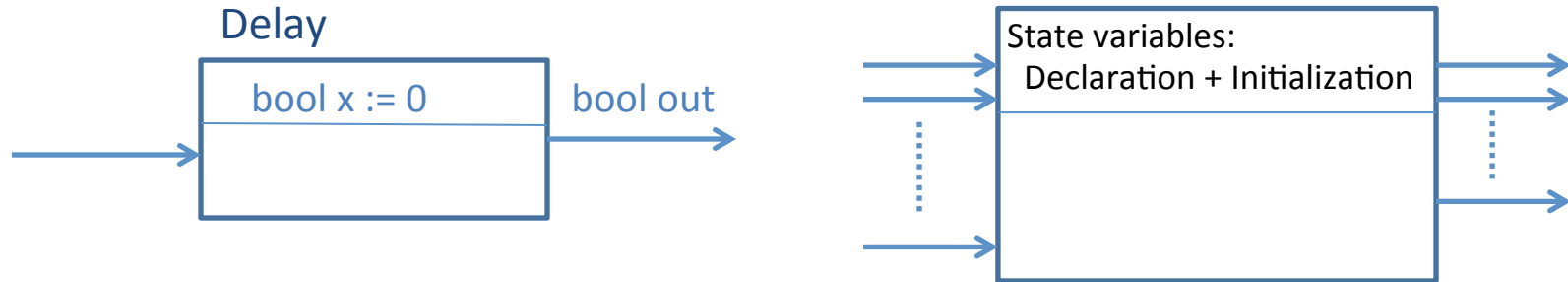
❑ Initialization of state variables specified by Init

- 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 { {x := 0} }

❑ Component can have multiple (alternative) initial states

- Example: bool x := choose {0, 1}

SRC Definition (5): Reactions
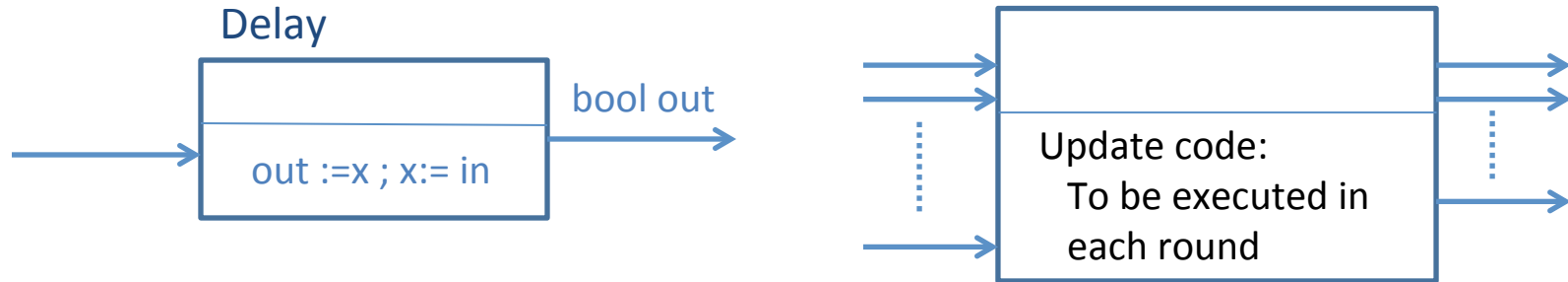
❑ Execution in each round given by code fragment React

- 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 -> (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 -> 0; 0 - 1/0 -> 1; 1 - 0/1 -> 0; 1 - 1/1 -> 1

Multiple Reactions

bool in → [bool x := 0 | out := x ; x := choose { in, x }] → bool out
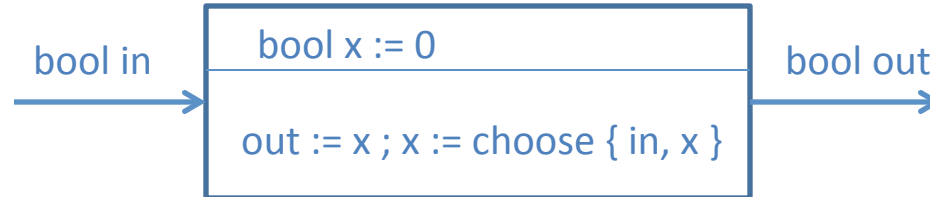
❑ During update, x is either updated to input in or is left unchanged
  ▪ Motivation: models the possibility that an input may be lost
❑ Nondeterministic reactions
  ▪ Given (old) state and input, output/new state need not be unique
  ▪ The set [React] of reactions now contains

    0 - 0/0 -> 0

    0 - 1/0 -> 1; 0 - 1/0 -> 0

    1 - 0/1 -> 0; 1 - 0/1 -> 1

    1 - 1/1 -> 1

Multiple Reactions

```
                ┌────────────────────────────┐
  bool in       │       bool x := 0          │        bool out
 ─────────────▶ ├────────────────────────────┤ ─────────────▶
                │  if x != in then           │
                │    { out := x ; x := in }  │
                │                            │
                └────────────────────────────┘
```
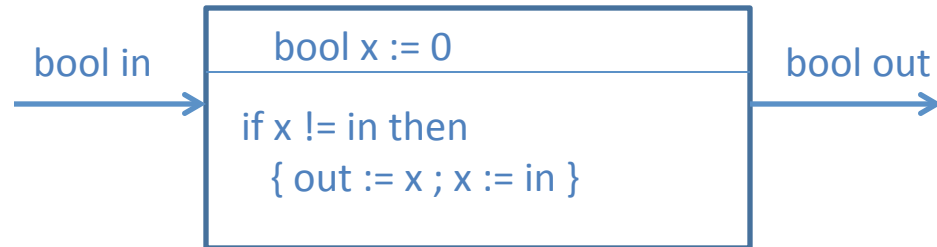
❑ A component may not accept all inputs in all states

  ▪ 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

bool in → [bool x := 0 | x := out; out := in] → bool out
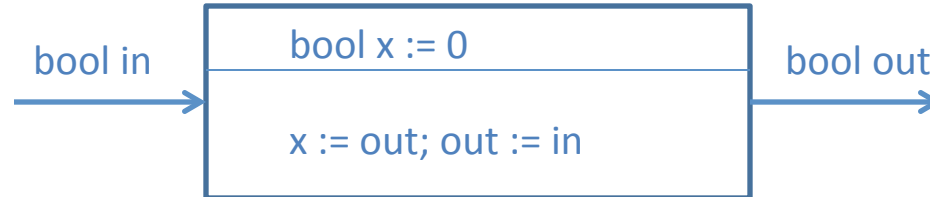
❑ Update code expected to satisfy a number of requirements:
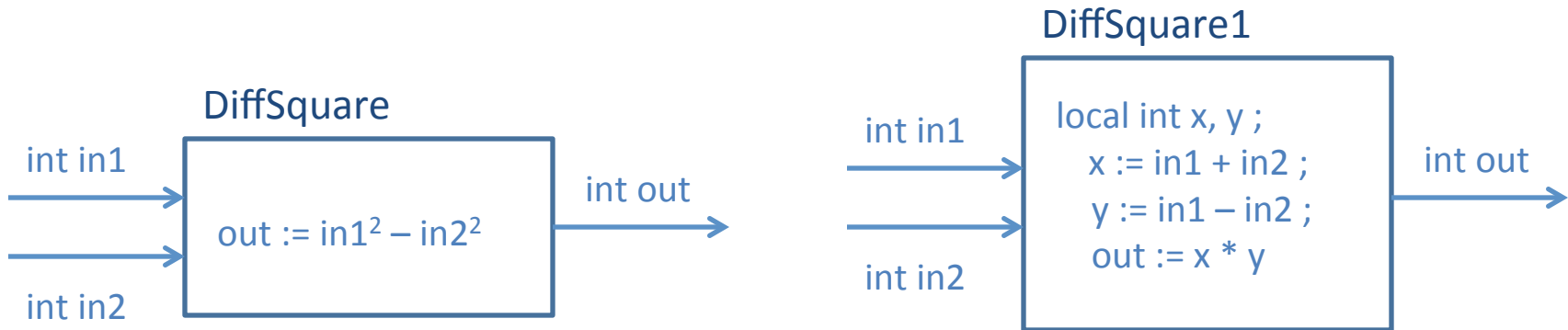 - Types of variables and expressions should match
  - Output variables must first be written before being read
  - Output variable must be explicitly assigned a value

❑ Otherwise, then no reaction possible
  - In above: set [React] of reactions is the empty set

Semantic Equivalence

DiffSquare

int in1

int in2

$$out := in1^2 - in2^2$$

int out

DiffSquare1

int in1

int in2

local int x, y ;
    x := in1 + in2 ;
    y := in1 − in2 ;
    out := x * y

int out

❑ 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, Init, React), what are its executions?

❑ Initialize state to some state s_0 in [Init]
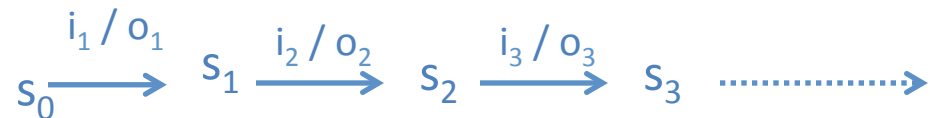
❑ Repeatedly execute rounds. In each round n = 1, 2, 3, …
  Choose an input value i_n in Q_I
  Execute React to produce output o_n and change state to s_n
    that is, $s_{n-1} - i_n / o_n \rightarrow s_n$ must be in [React]

❑ Sample execution:

$$s_0 \xrightarrow{i_1 / o_1} s_1 \xrightarrow{i_2 / o_2} s_2 \xrightarrow{i_3 / o_3} s_3 \cdots\cdots\cdots\rightarrow$$

What does this component do ?

bool in →

bool x := 0 ; y := 0
if y then out := x
else out := 0 ;
x := in ;
y := ¬y

→ bool out
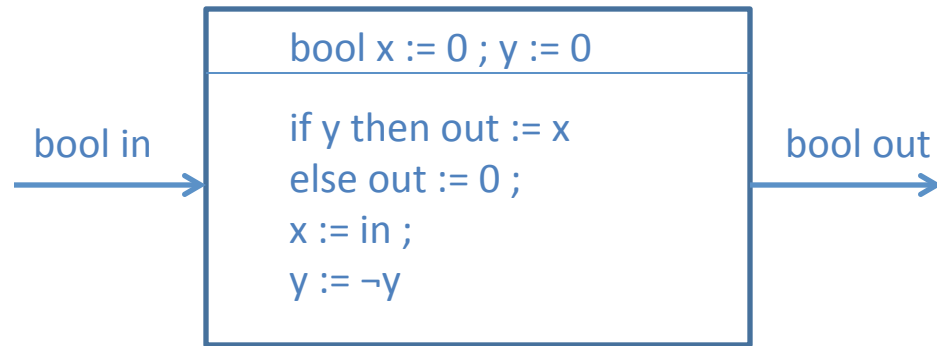
Extended State Machines

Input: bool press

(press = 0) ?

(press = 0 & x < 10) -> x := x+1

(press = 1) ?

int x := 0

off

on
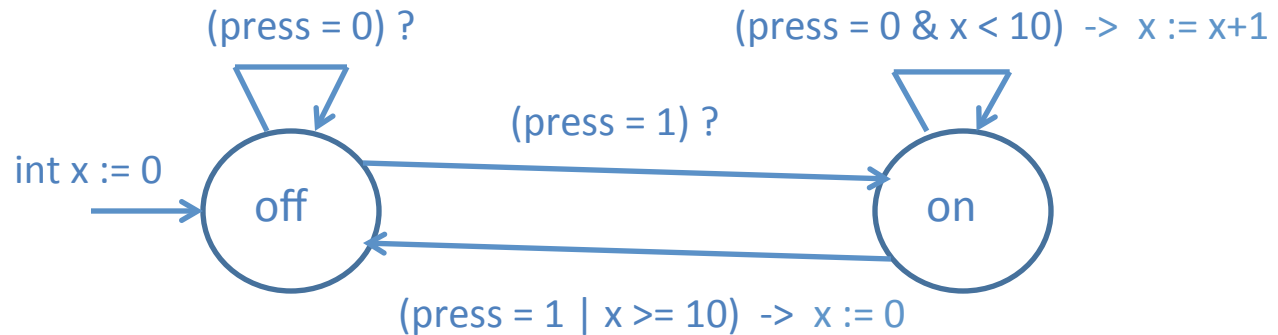
(press = 1 | x >= 10) -> x := 0

mode is an implicit 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* x := 0

Executing ESMs: Switch

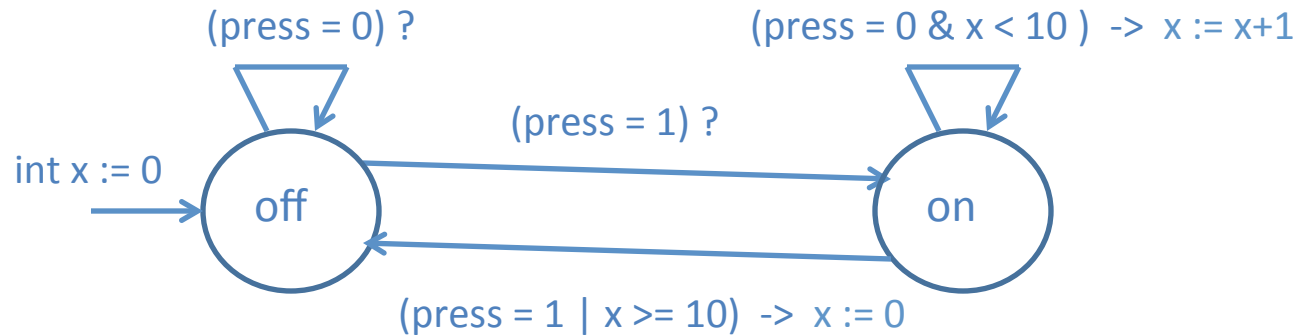Input: bool press

(press = 0) ?   (press = 0 & x < 10) -> x := x+1
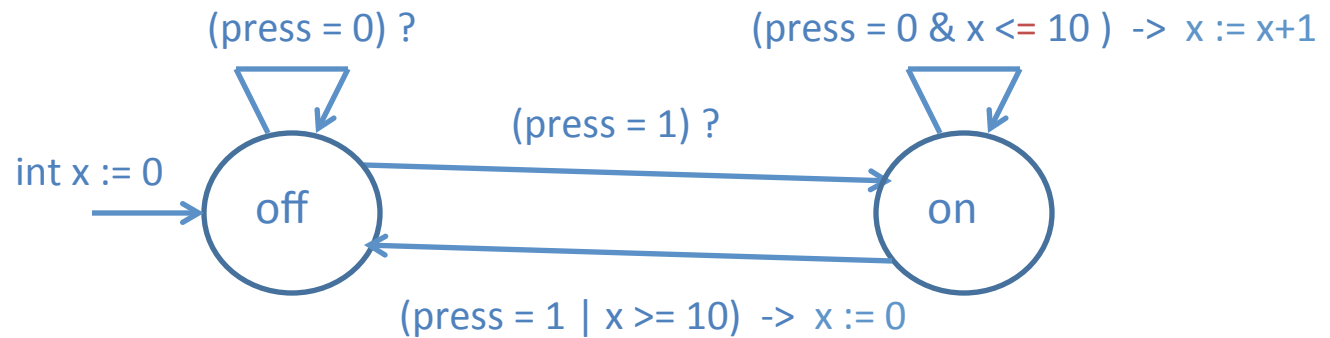
(press = 1) ?

int x := 0   off   on

(press = 1 | x >= 10) -> x := 0

❑ State of the component Switch assigns values to mode and x

❑ Initial state: (off, 0) (i.e., {mode := off, x := 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

Exercise: ESM to SRC

Input: bool press

(press = 0) ?

(press = 0 & x < 10) -> x := x+1

(press = 1) ?

int x := 0

off

on

(press = 1 | x >= 10) -> x := 0
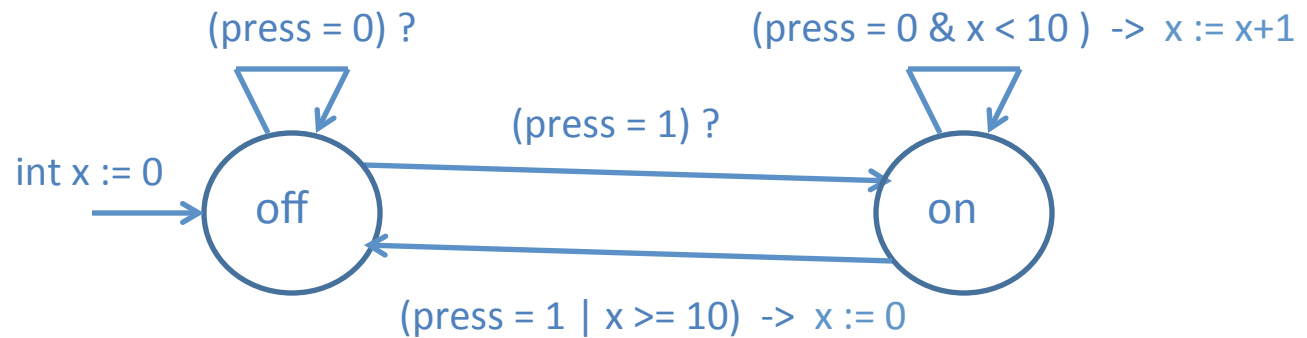
Rewrite this ESM as a synchronous reactive component

Finite-State Components

Delay

bool in → | bool x := 0

 out := x ; x := in | → bool out

DiffSquare

int in1 →
 int in2 → | out := $in1^2 - in2^2$ | → int out
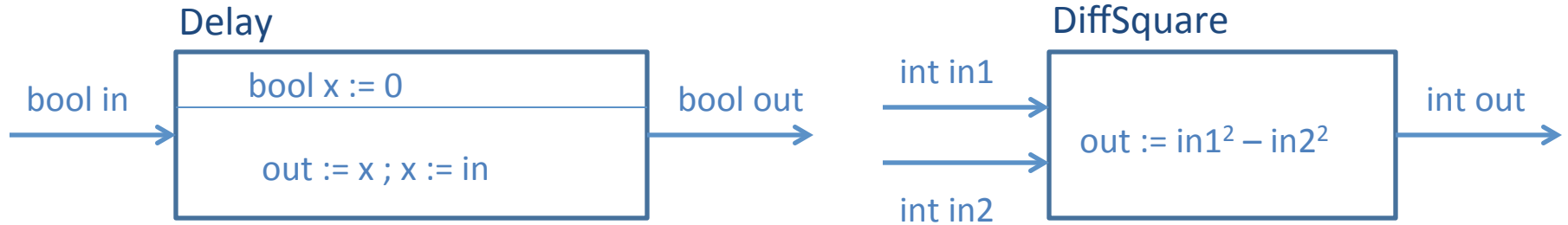
A component is *finite-state* if all its variables range over finite types

- 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)
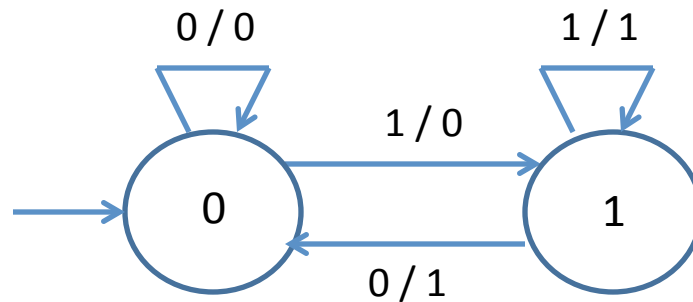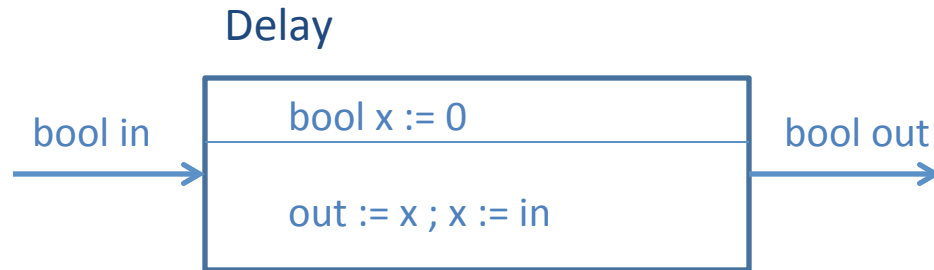

Finite-state components are amenable to exact, algorithmic analysis

Switch: Is it finite-state?

Input: bool press

(press = 0) ?

(press = 0 & x < 10) -> x := x+1

(press = 1) ?

~~int x := 0~~

off

on

int[0,10] x := 0

(press = 1 | x >= 10) -> x := 0

Combinational Components

Delay

bool in

bool x := 0
out := x ; x := in

bool out

DiffSquare

int in1

int in2

$$out := in1^2 - in2^2$$

int out
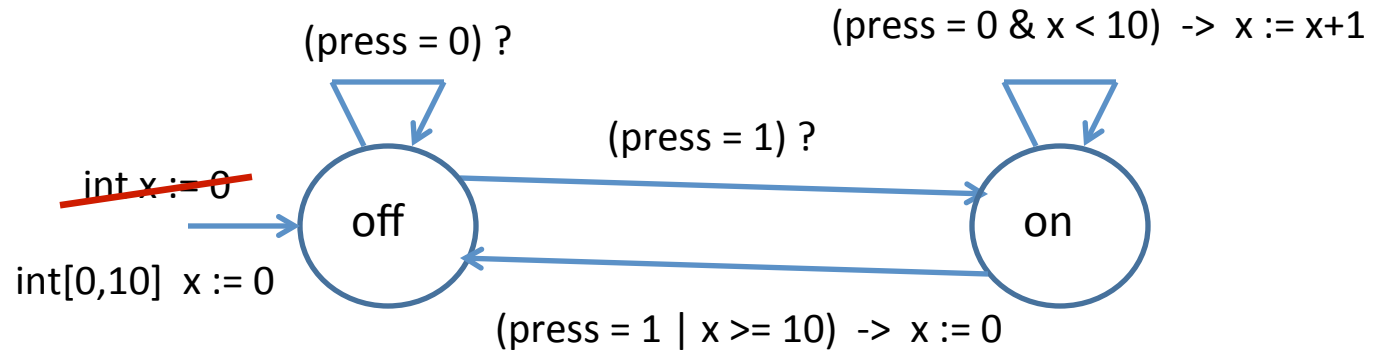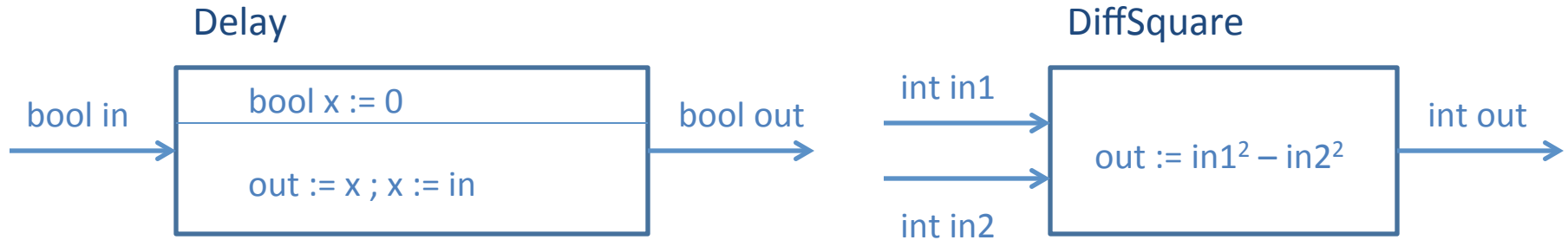
A component is *combinational* if it has no state variables

- DiffSquare is combinational, but Delay is not
- Hardware gates are combinational components

Events

❑ Input/output variable can be of type event

❑ Motivation: notion of clock can be different for different components

❑ An event can be absent, or present, in which case it has a value

  ▪ event x means x ranges over {present, $\perp$}
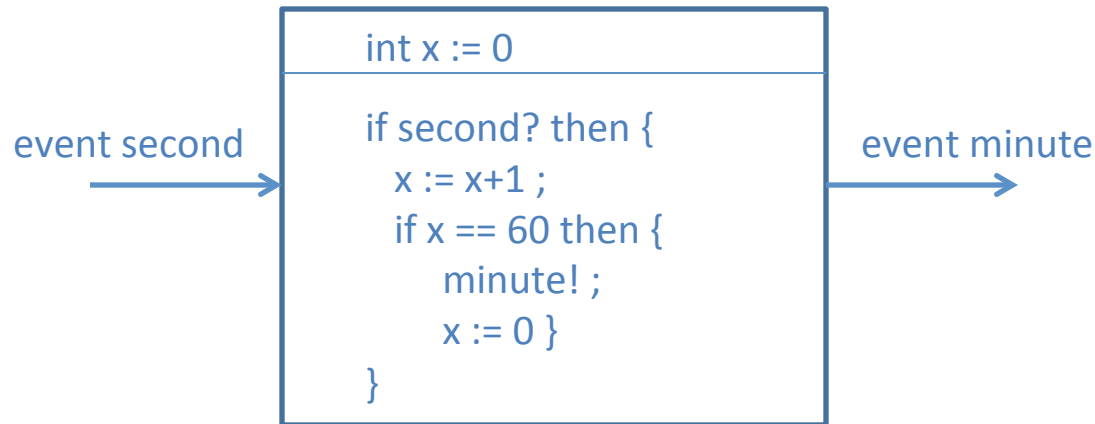
  ▪ event(bool) x means x ranges over {0, 1, $\perp$}

  ▪ event(nat) x means x ranges over {$\perp$, 0, 1, 2, …}

❑ Syntax: x? is a short form for the test (x != $\perp$)

❑ Syntax: x!v is a short form for 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)

Second-To-Minute

Desired behavior (spec):

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

event second →

```
int x := 0

if second? then {
    x := x+1 ;
    if x == 60 then {
        minute! ;
        x := 0 }
}
```

→ event minute

Event-Triggered Components
- No need to execute in a round where triggering input events absent
- See textbook for formal definition

Deterministic Components

Delay

bool x := 0
out := x ;   x := in

bool in →

bool out →

LossyDelay

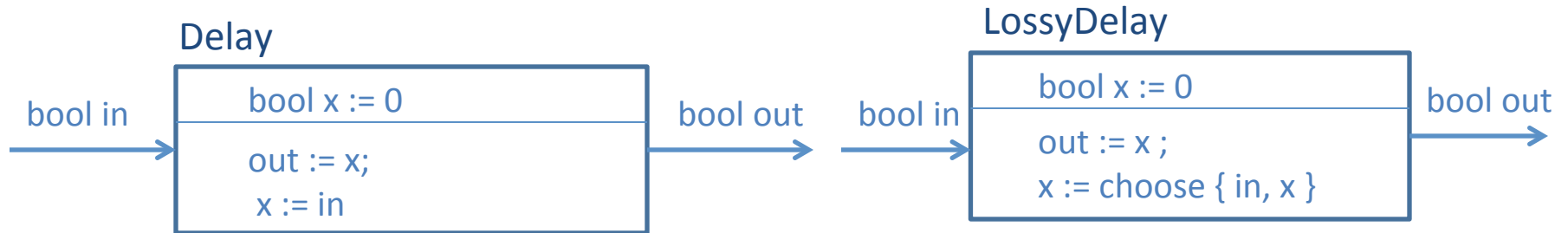| bool x := 0 |
| --- |
| out := x ;
x := choose { in, x } |

bool in →

bool out →
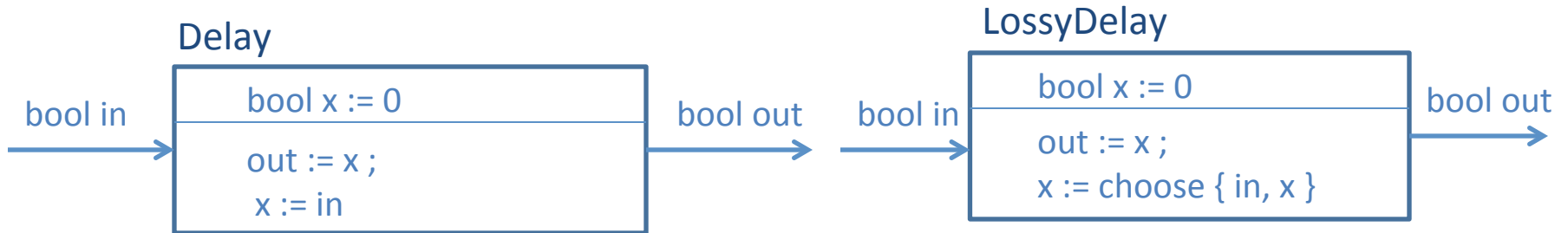
A component is *deterministic* if

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 – i/o -> t is a reaction

Delay is deterministic, but LossyDelay is not

Deterministic Components

Delay

```
bool x := 0

out := x;
 x := in
```

bool in → [Delay] → bool out

LossyDelay

```
bool x := 0

out := x ;
x := choose { in, x }
```

bool in → [LossyDelay] → bool out

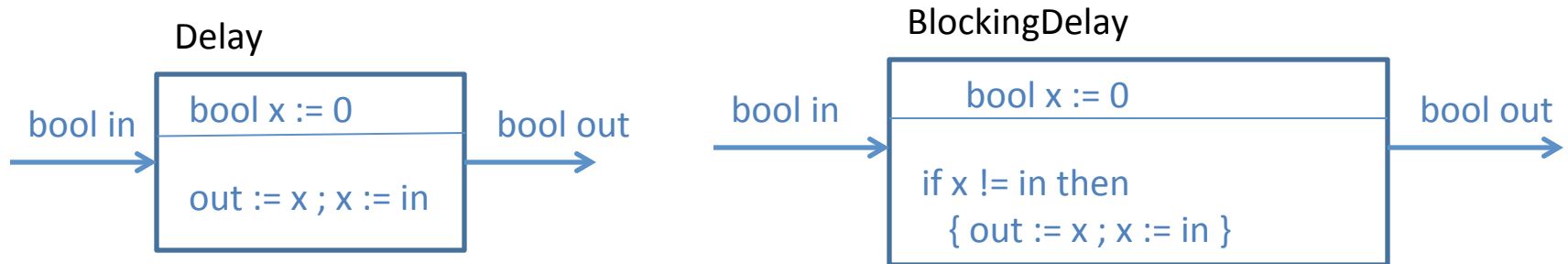- ❑ Deterministic: same sequence of inputs supplied, same outputs observed (predictable, repeatable behavior)

- ❑ Nondeterminism is useful in modeling uncertainty /unknown/ abstraction
- ❑ Nondeterminism is different from probabilistic (or random) choice

What does this component do?

Arbiter

event req1

event req2

req1? -> grant1!

req2? -> grant2!

¬req1? & ¬req2?

event grant1

event grant2
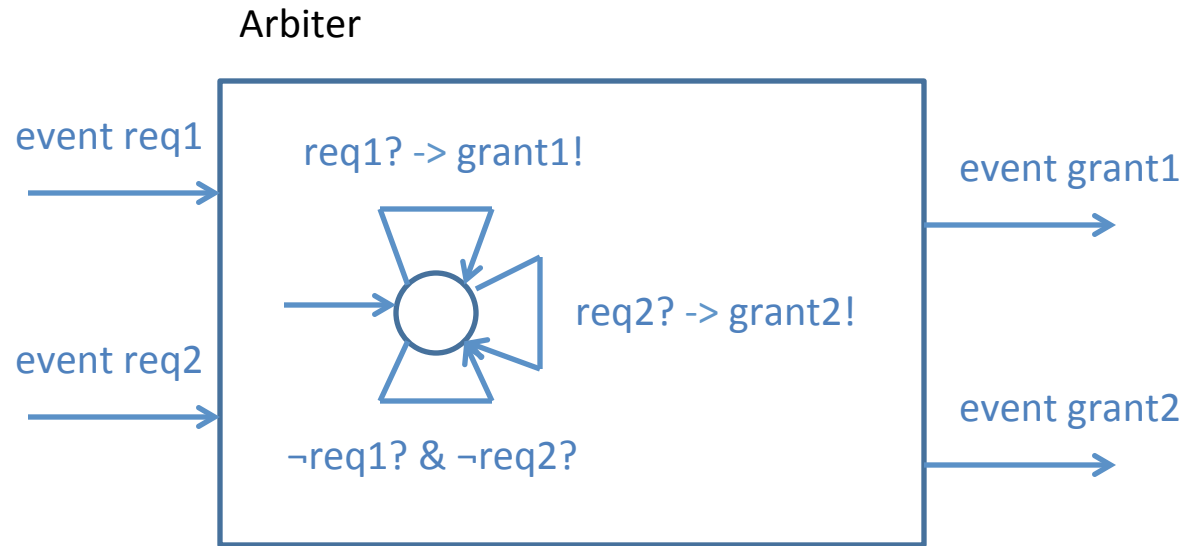
Input Enabled Components

Delay

| bool in → | bool x := 0 | → bool out |
| out := x ; x := in |

BlockingDelay

| bool in → | bool x := 0 | → bool out |
| if x != in then { out := x ; x := in } |

❑ 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 – 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

Splitting Reaction Code into Tasks

SplitDelay
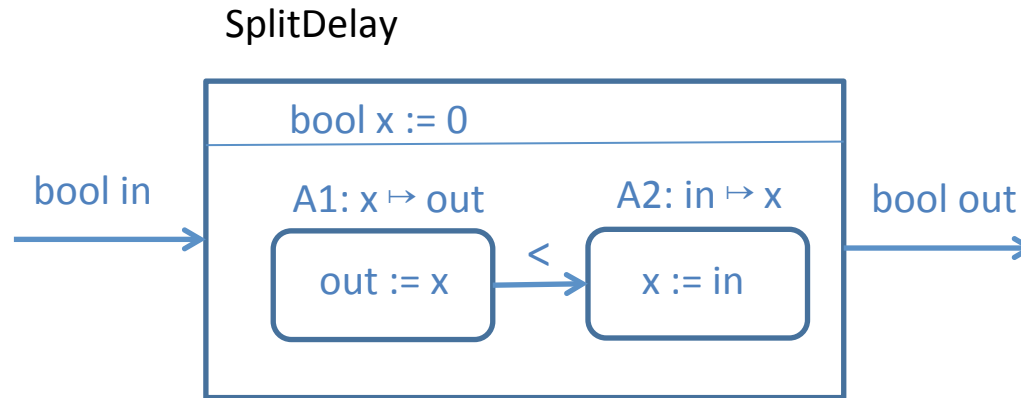

❑ A1 and A2 are tasks (atomic blocks of code)
 ▪ Each task specifies variables it reads and writes
 ▪ A1 reads x and writes out

❑ *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

ParallelRelay
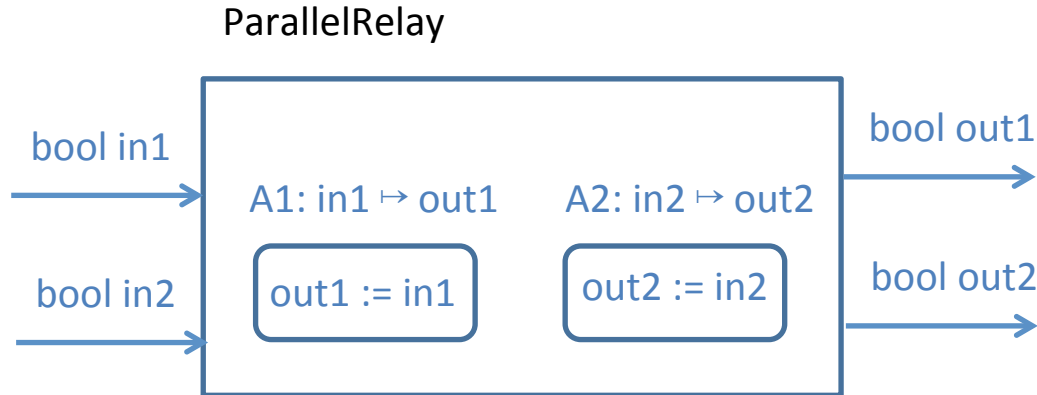
❑ Tasks A1 and A2 are unordered

  ▪ 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
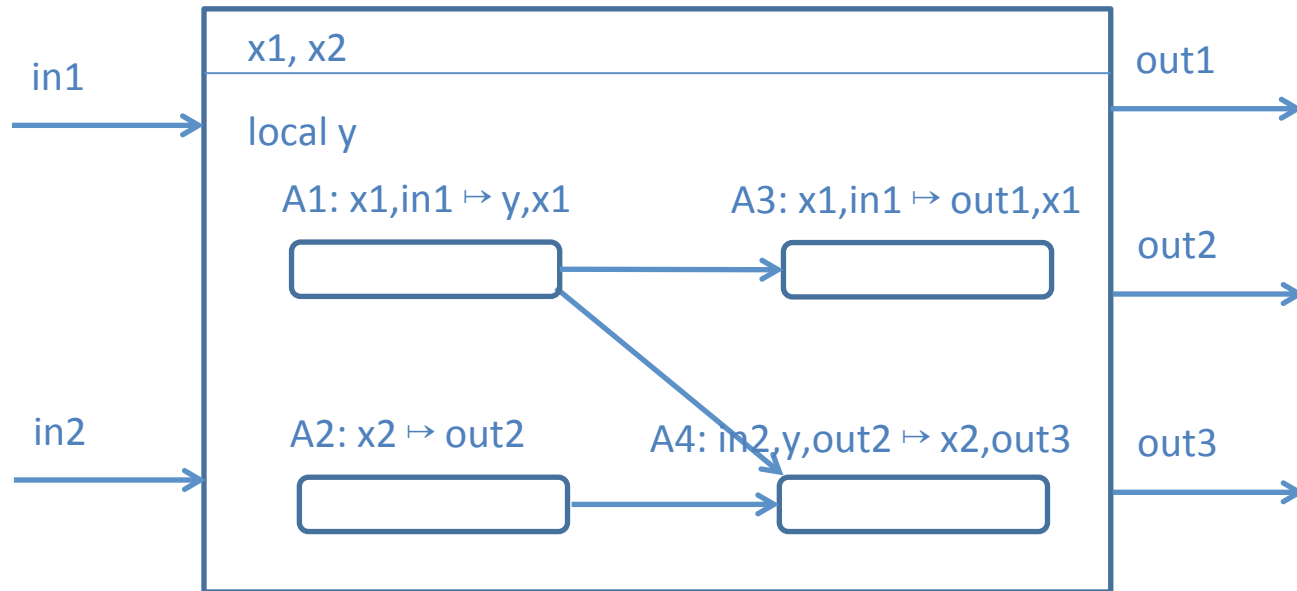
❑ What are possible schedules consistent with precedence constraints?

❑ 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   local vars L

the 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

❑ Relation $<^+$ is the *transitive closure* of the relation $<$

❑ Task schedule: Total ordering A_1, A_2, …, A_n of all the tasks that is consistent with the precedence edges

- If $A < A'$, then A must appear before A' in the ordering

- Multiple schedules are possible

- If $A <^+ A'$ then A must appear before A' in every schedule

❑ Acyclicity implies 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, written y > x, if

either the task A reads x

some another task A' such that A' $<^+$ A reads x

Note: 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' <⁺ A

Requirements on Task Graph (4)

Tasks with a write conflict must be ordered

❑ There is a *write-conflict* between tasks A and A'
   if a variable written by A is read or written by A'

❑ If A and A' have a write-conflict, the result depends on whether A
   executes before A' or vice versa.

 ▪ Example: A update is x := x+1; A' update is out := x

❑ If tasks A and A' have a write-conflict then they must be ordered:
   either A $<^+$ A' or A' $<^+$ A

❑ This way, set of reactions resulting from executing all the tasks do not
   depend on the task schedule

Task Properties

- [ ] 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 $(u,v) \in [\text{Update}]$

- [ ] 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 $(u,v) \in [\text{Update}]$

- [ ] If all tasks of a component are input-enabled, what can we conclude about the component itself?

Credits

Notes based on Chapter 2 of

Principles of Cyber-Physical Systems
by Rajeev Alur
MIT Press, 2015