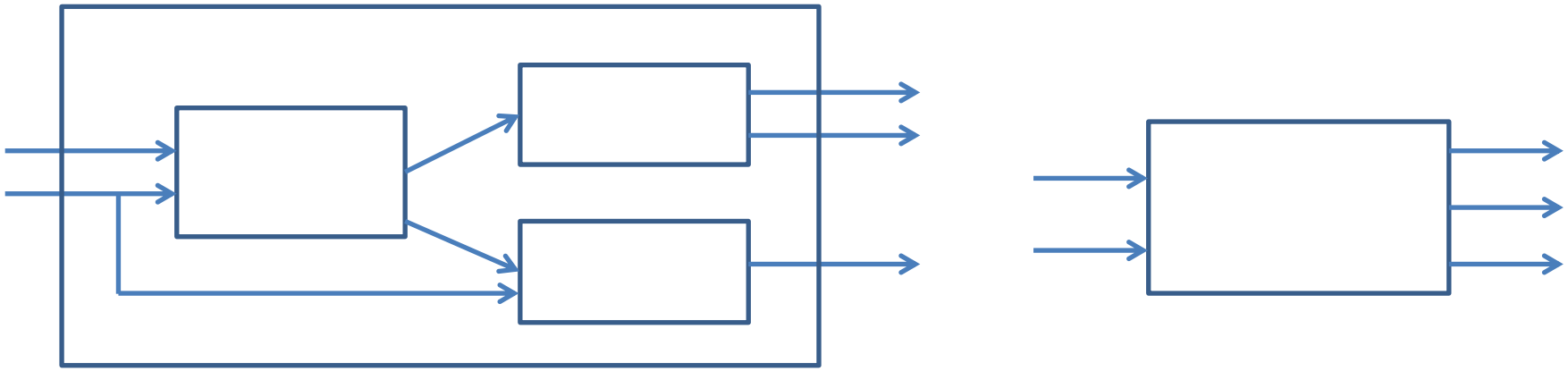
CS:5980
Foundations of Embedded Systems

The 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

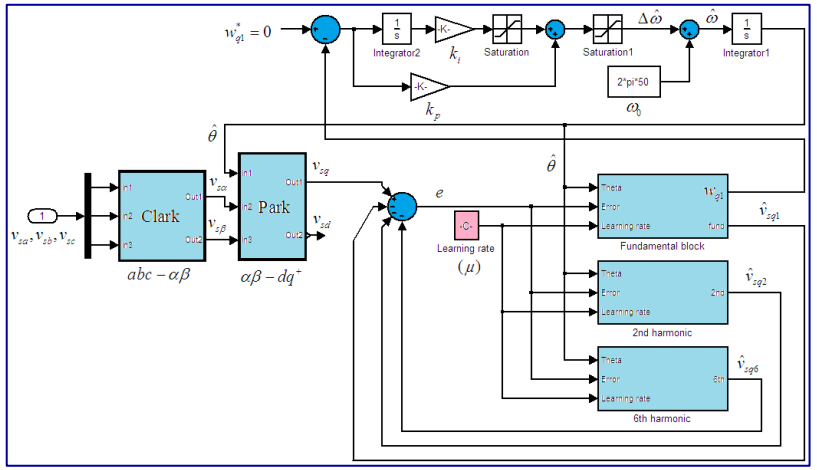
Sequential model of computation (classical one):

- 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

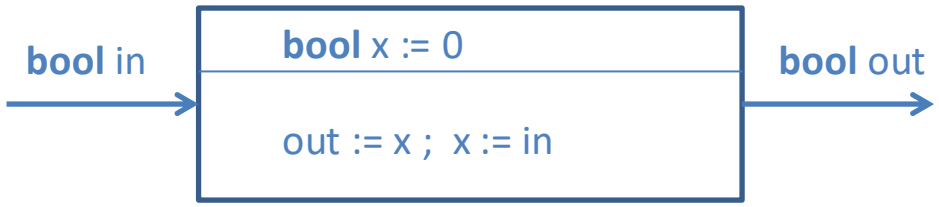
Example: Component blocks in digital hardware circuit

- Clock drives all components in a synchronized manner

Key idea in synchronous languages:

- Design system based on such 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 Boolean type ({0,1})

Output variable: out of Boolean type

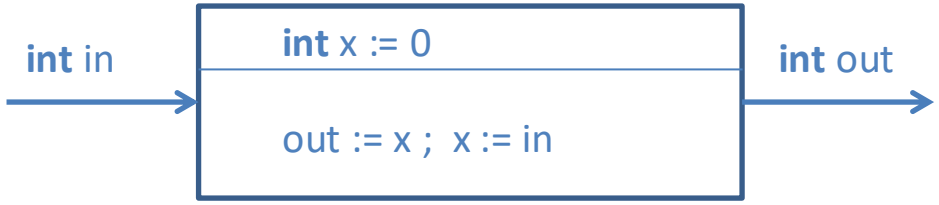
State variable: x of Boolean type

Initialization (of state variables): x := 0

Execution: in each round, in response to the current input,

- produce output (out := x) and
- update state (x := in)

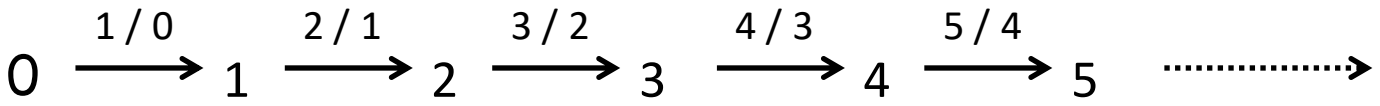
Int Delay: Round-based Execution



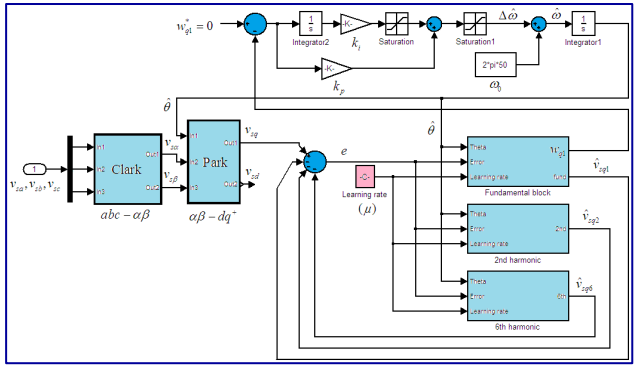
1. Initialize state x to 0

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
   c. Change state as needed

Example execution:



Synchrony Hypothesis



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

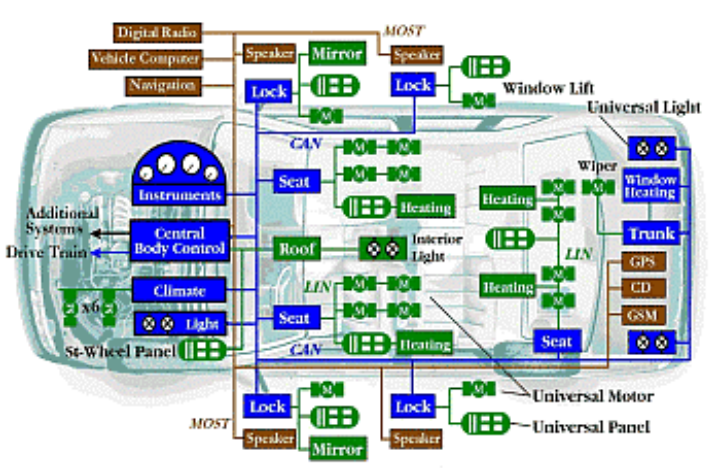
Logical abstraction:

- 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

Example: 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

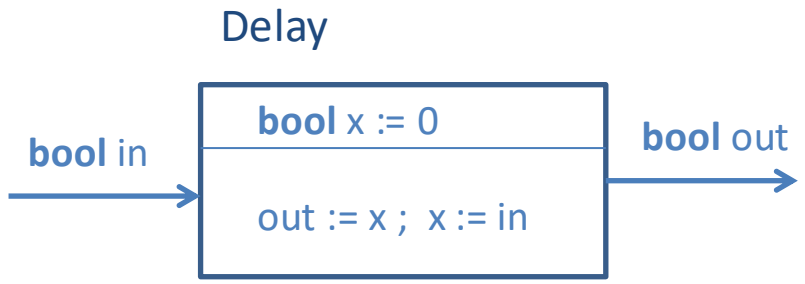
- Necessary for tools for analysis, compilation, verification, …
- Defining formal semantics for a real language is challenging
- But concepts can be illustrated on a toy modeling language

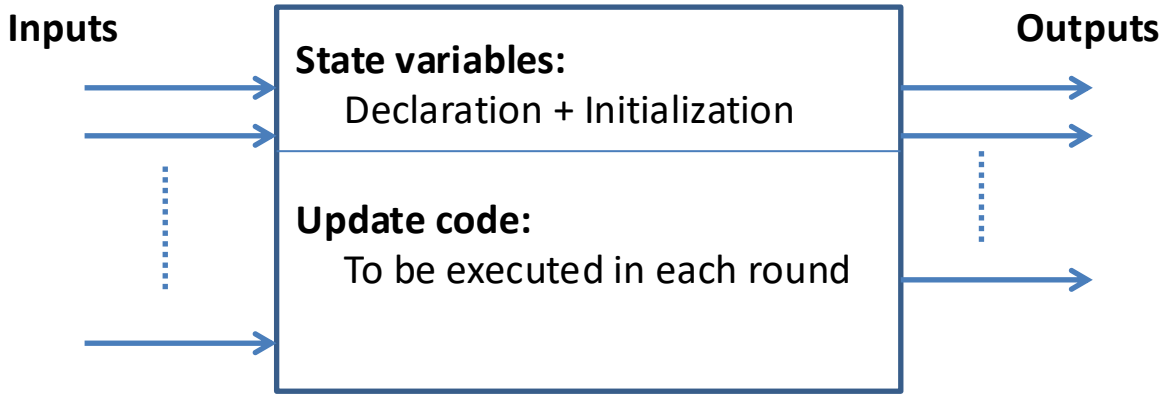
Our Modeling Language

Synchronous Reactive Components

- Representative of many academic languages:
  Signal, Lustre, Lucid, SMV, Lola, Lucent, …
- Foundations of industrial-strength synchronous languages:
  Esterel, SCADE, VHDL, Verilog, Stateflow, …

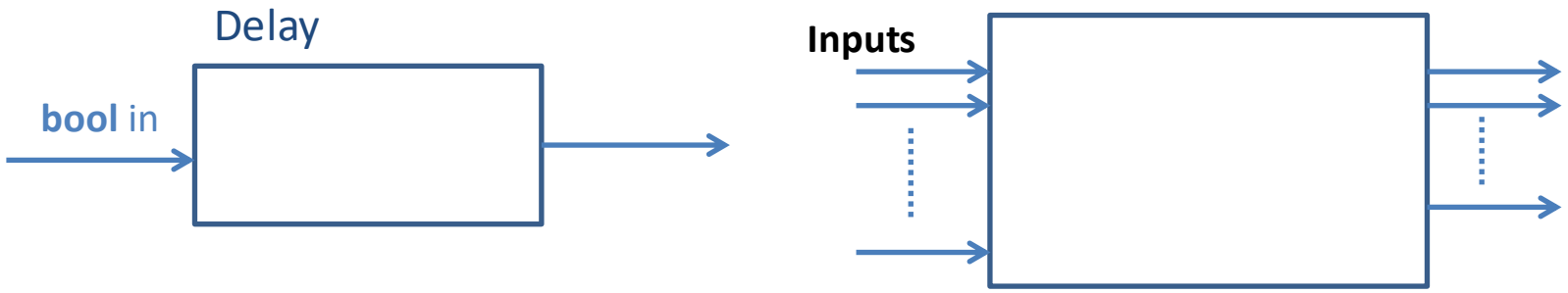
Synchronous Reactive Component





Update code contains no function calls or loops

SRC Definition — Inputs



Each component has a set I of input variables

- Variables have types. E.g., **bool**, **int**, **nat**, **real**, {on, off}, …

Input: Assignment of all the input variables

- The set of possible inputs (input assignments) is Q_I

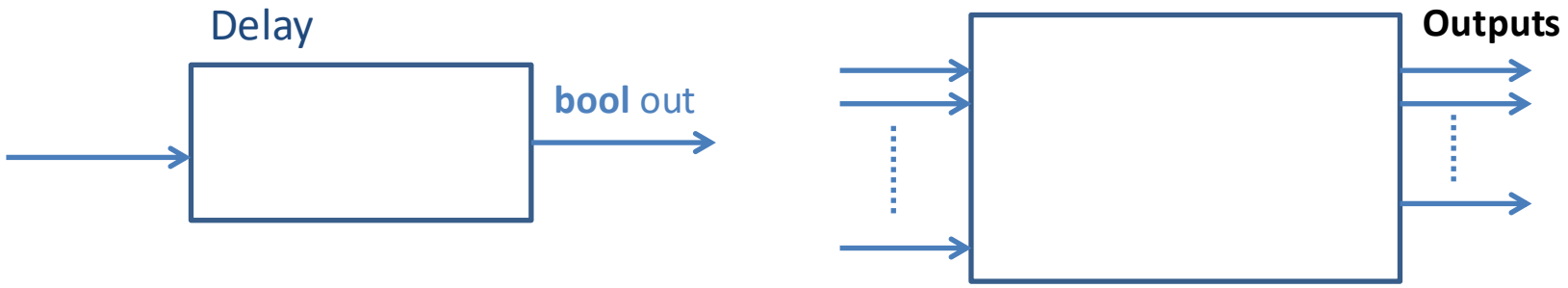
For Delay

- I contains a single variable in of type **bool**
- The set of inputs is { { in := 0 }, { in := 1 } }

Example: I consists of two variables: **int** x , **bool** y

- Each input is a pair: (integer value for x and Boolean value for y)

SRC Definition — Outputs



Each component has a set O of typed output variables

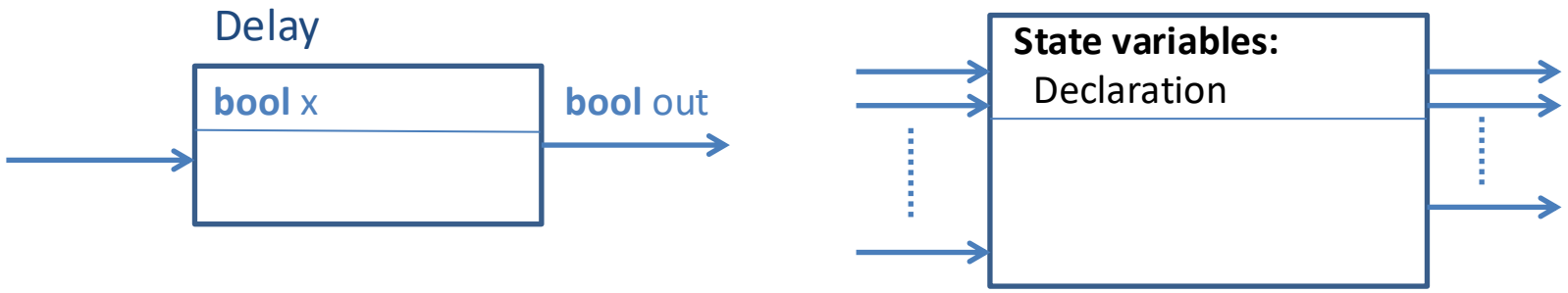
Output: Assignment of all the output variables

- The set of outputs (output assignments) is Q_O

For Delay

- O contains a single variable out of type **bool**
- The set of outputs is { { out := 0 }, { out := 1 } }

SRC Definition — States



Each component has a set S of typed state variables

State: Assignment of all the state variables

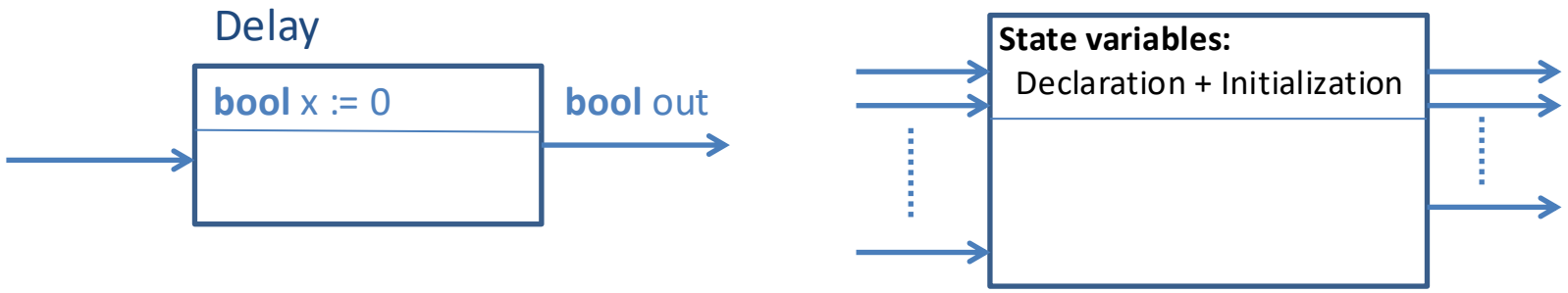
- The set of states is Q_S

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 — Initialization



Initialization of state variables specified by Init

- A sequence of assignments to state variables

Semantics of initialization:

- The set ⟦Init⟧ of initial states, which is a subset of Q

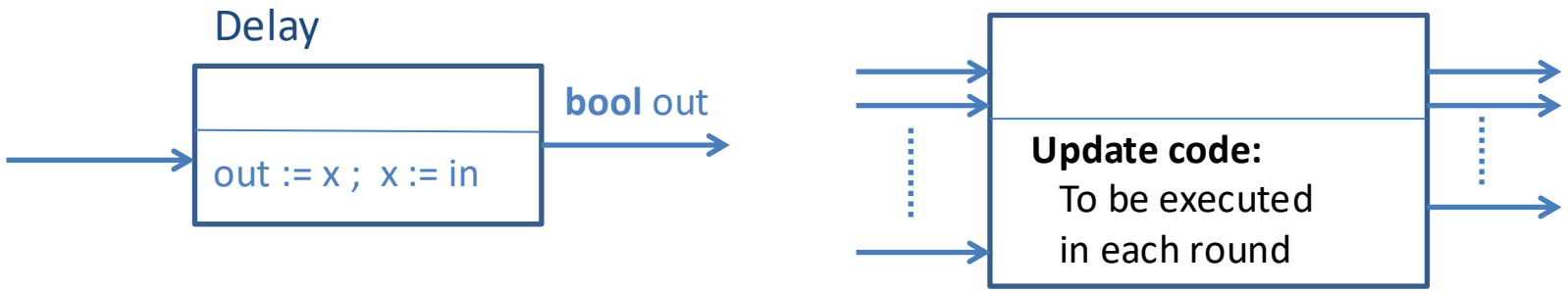
For Delay

- Init is given by the code fragment x := 0
- The set ⟦Init⟧ of initial states is { { x := 0 } }

Components can have multiple (alternative) initial states

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

SRC Definition — Reactions



Execution in each round given by code fragment React

- Sequence of assignments and conditionals that assign output variables and update state variables

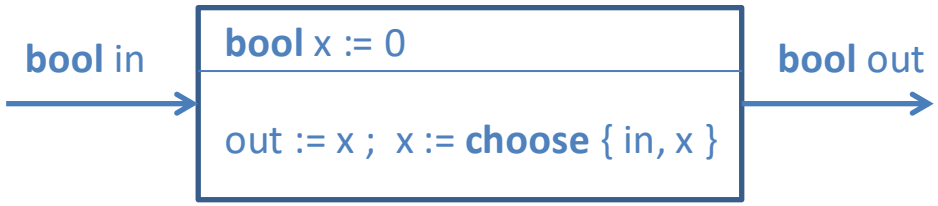
Semantics of update:

- The set ⟦React⟧ of reactions, where each reaction is of the form:
  (old) state – input / output –> (new) state
- **Note:** ⟦React⟧ is a subset of $Q_S \times Q_I \times Q_O \times Q_S$

For Delay:

- React is defined by the code fragment out := x ; x := in
- 4 possible reactions: 0 – 0/0 –> 0, 0 – 1/0 –> 1, 1 – 0/1 –> 0, 1 – 1/1 –> 1

Multiple Reactions



During update, x is either updated to input in or is left unchanged

- Models the possibility that an input may be lost or ignored

Nondeterministic reactions:

- Given (old) state and input, output/new state need not be unique
- The set $[\![\mathrm{React}]\!]$ of reactions now consists of

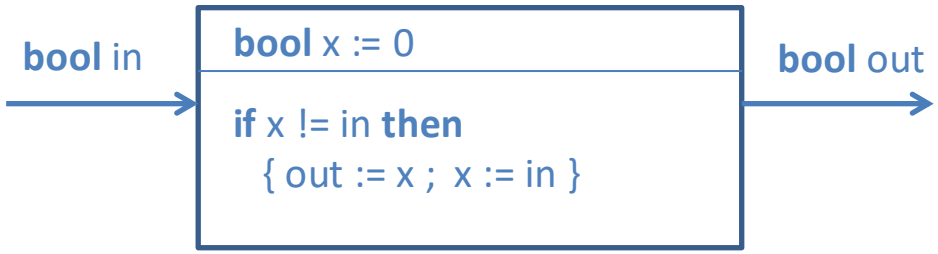
  0 – 0/0 –> 0

  0 – 1/0 –> 1, 0 – 1/0 –> 0

  1 – 0/1 –> 0, 1 – 0/1 –> 1

  1 – 1/1 –> 1

Selective Reactions



A component may not accept all inputs in all states

- **Motivation:** *blocking* communication

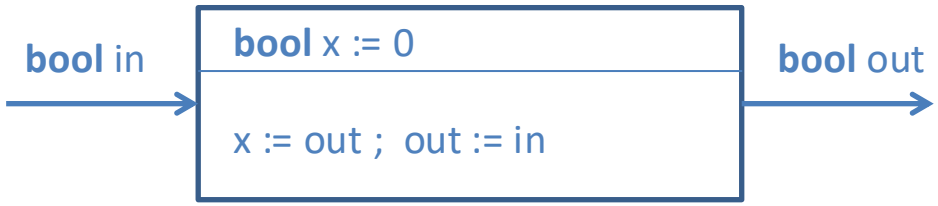
Possible set of reactions in certain state/input combinations may be empty

- The set $[\![\mathrm{React}]\!]$ of reactions now consists of

  0 – 1/0 –> 1

  1 – 0/1 –> 0

Syntax Errors



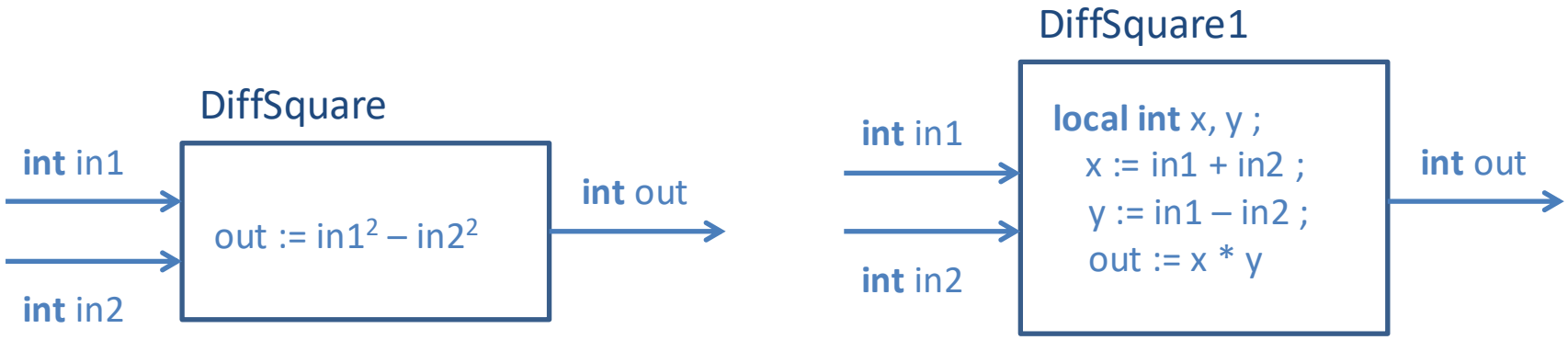
Update code is expected to satisfy a few requirements:

- Types of variables and expressions should match
- Output variables must first be written before being read
- Output variables must be explicitly assigned a value

Otherwise, no reaction possible

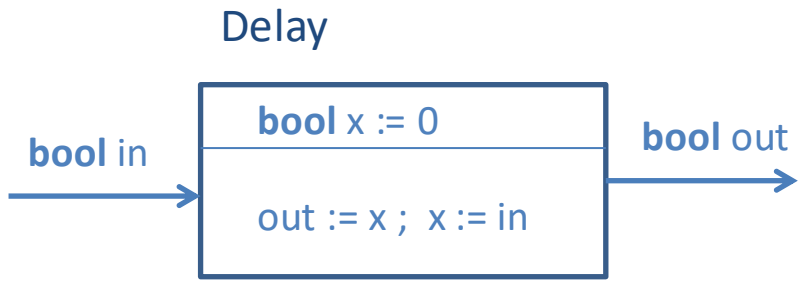
- In above, the set $[\![\mathrm{React}]\!]$ of reactions is empty

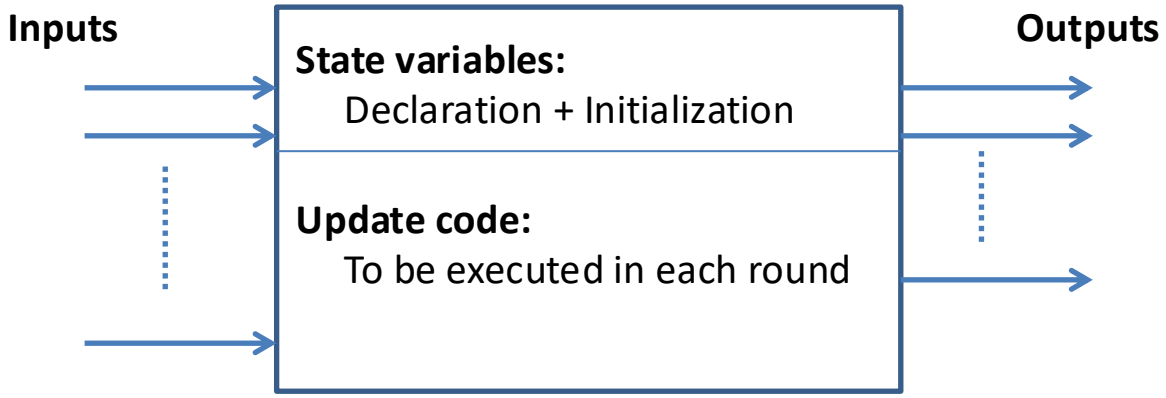
Semantic Equivalence



- These components have identical sets of reactions
- They are syntactically different but semantically equivalent
- Compiler can optimize code as long as semantics is preserved!

Synchronous Reactive Component





Update code contains no function calls or loops

Synchronous Reactive Component Definition

- Set I of typed *input variables* gives set $\mathrm{Q_I}$ of input assignments, or *inputs*
- Set O of typed *output variables*: gives set $\mathrm{Q_O}$ of output assignments, or *outputs*
- Set S of typed *state variables*: gives set $\mathrm{Q_S}$ of state assignments, or *states*
- Initialization code Init: defines set $[\![\mathrm{Init}]\!]$ of *initial states*
- Reaction description React: defines set $[\![\mathrm{React}]\!]$ of *reactions*
  of the form $\mathrm{s} - \mathrm{i} \,/\, \mathrm{o} -\!> \mathrm{t}$, 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!

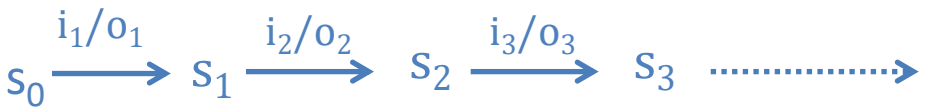
Definition of Executions

1. Given component $C = (I, O, S, Init, React)$, what are its executions?

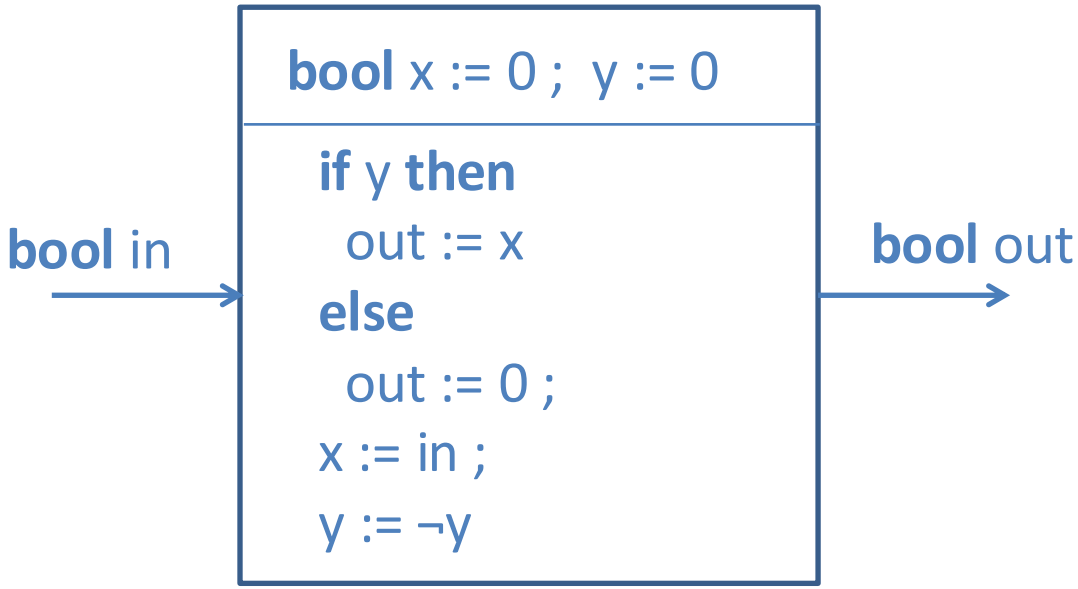
2. Initialize state to some state s_0 in $[\![Init]\!]$

3. Repeatedly execute rounds. In each round $n = 1, 2, 3, \ldots$
   a. Choose an input value i_n in Q_I
   b. Execute $React$ to produce output o_n and change state to s_n
      - that is, $s_{n-1} - i_n / o_n -> s_n$ must be in $[\![React]\!]$

Example execution:



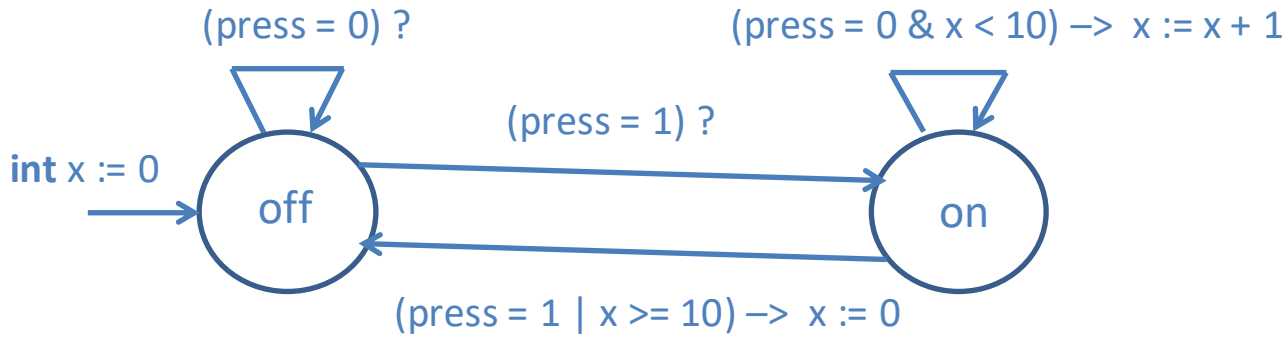
Exercise: What does this component do?



Explain behavior just in terms of the component's input and output

Extended State Machines

Input: **bool** press



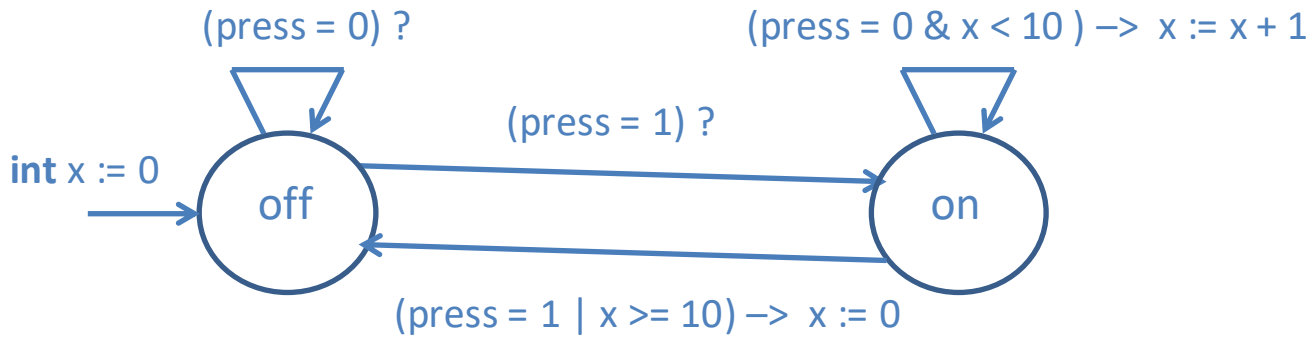
- There is an implicit state variable mode ranging over {on, off}
- Reaction corresponds to executing a *mode-switch*

Example mode-switch: from on to off with

guard (press = 0 | x >= 10) and *update* x := 0

Executing ESMs: Switch

Input: **bool** press



State of Switch assigns values to mode and x

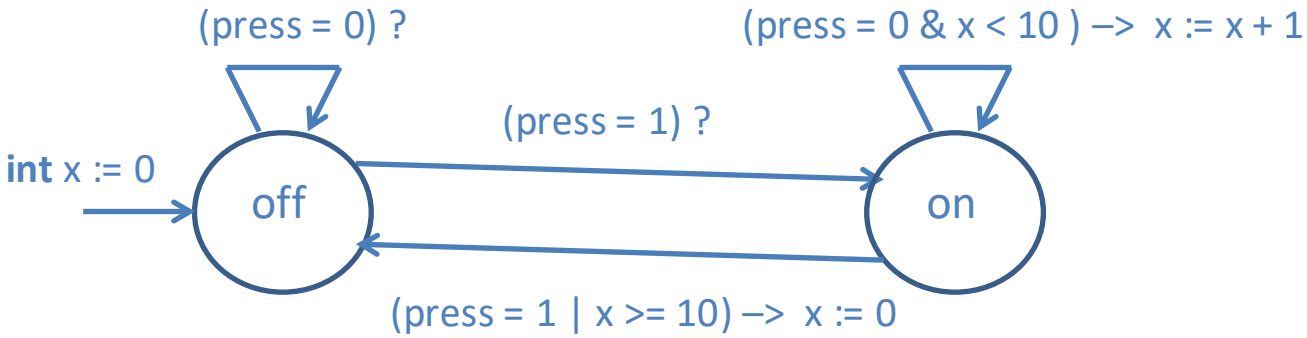
Initial state: (off, 0) (i.e., { mode := off, x := 0 }) **Output:** none

Example execution:

(off, 0) – 0 –> (off, 0) – 1 –> (on, 0) – 0 –> (on,1) – 0 –> (on,2) … – 0 –> (on, 10) – 0 –> (off, 0)

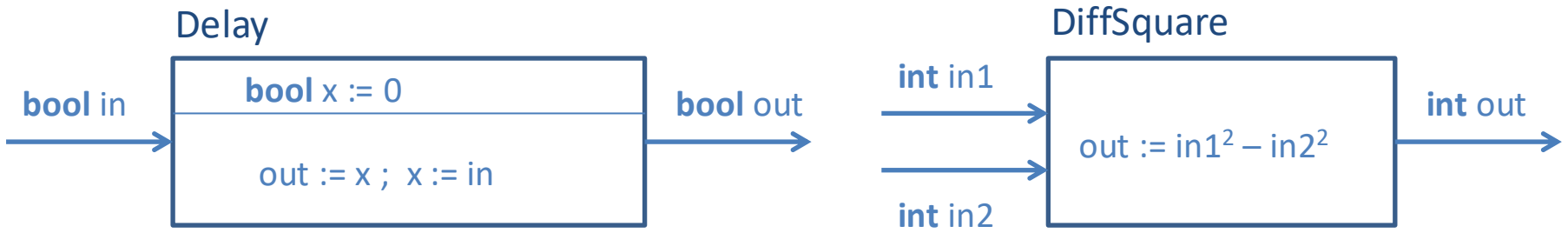
Exercise: ESM to SRC

Input: **bool** press



Rewrite this ESM as a synchronous reactive component

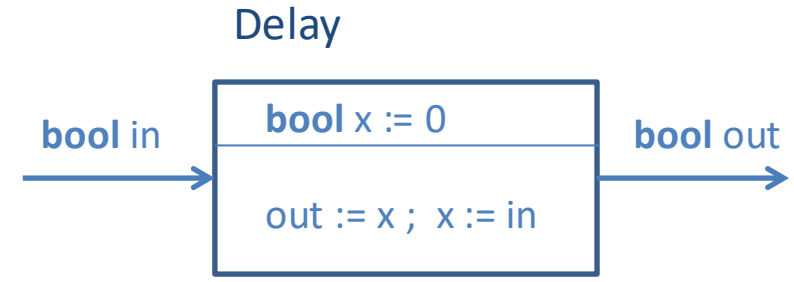
Finite-State Components

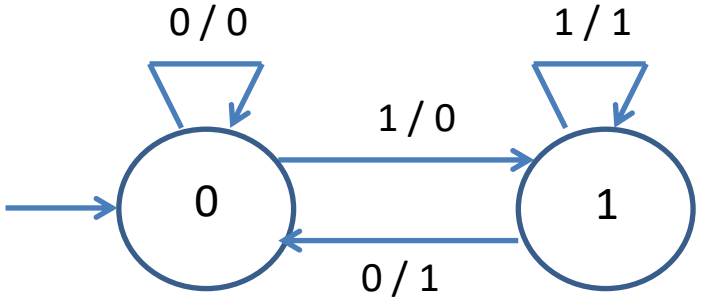


A component is *finite-state* if all its variables range over finite types

- *Finite types:* bool, enumerated and finite range types (e.g. {on, off}, **int**[-5..5])
- Delay is finite-state, but DiffSquare is not

Mealy Machines (for finite-state components)

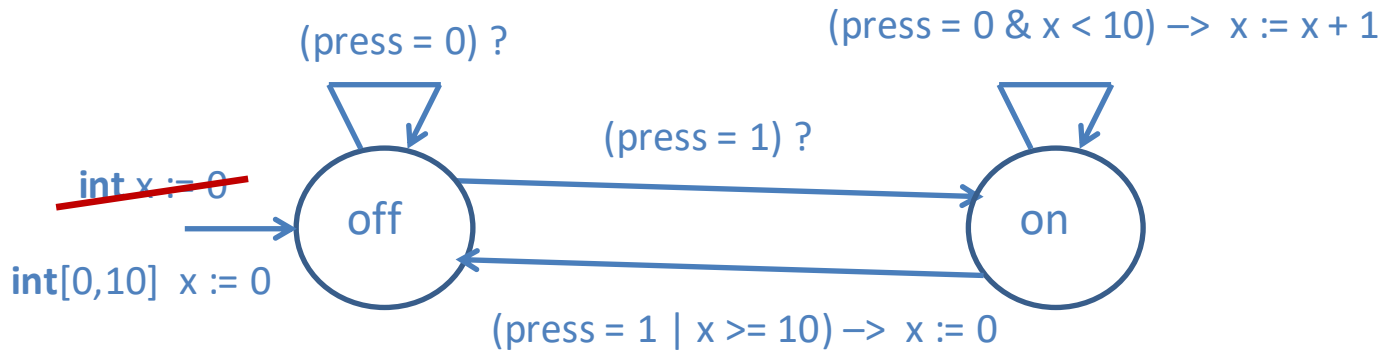




Finite-state components are amenable to exact, algorithmic analysis

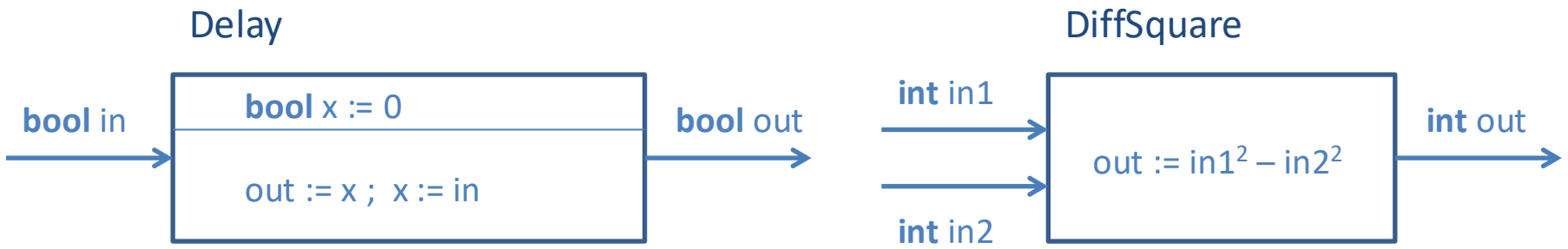
Switch: Is it finite-state?

Input: **bool** press



The system is effectively finite-state

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 it has a value

- **event** x means x ranges over {present, ⊥}
- **event(bool)** x means x ranges over {0, 1, ⊥}
- **event(nat)** x means x ranges over {⊥, 0, 1, 2, …}

Motivation: notion of clock can be different for different components

Syntax:

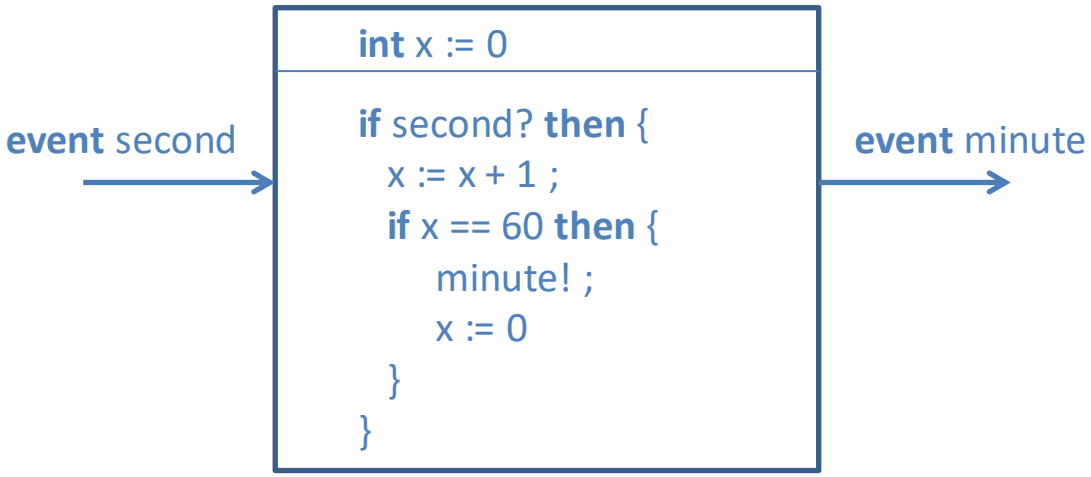
- x? is a short form for the test x != ⊥
- **Syntax:** x!v is a short form for the assignment x := v
- **Syntax:** x! is a short form for the assignment x := present (for event x)

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

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

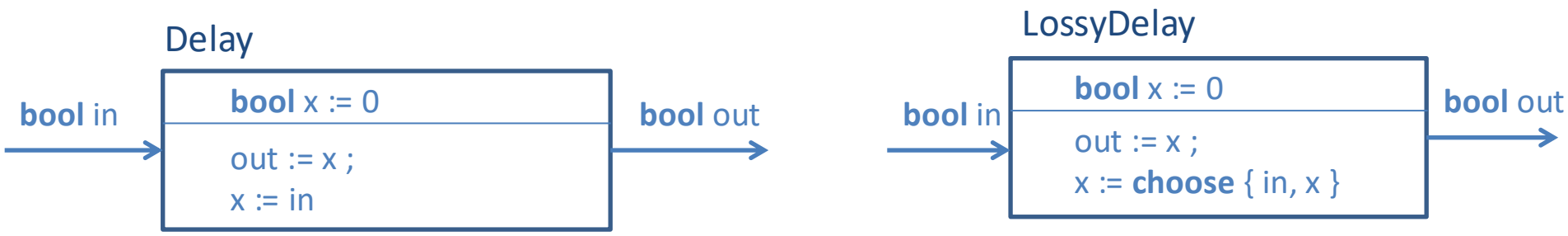


Event-Triggered Components

- Do not execute in a round where triggering input events are absent
- Technically, they do not output any event when inputs are absent

See textbook for formal definition

Deterministic Components

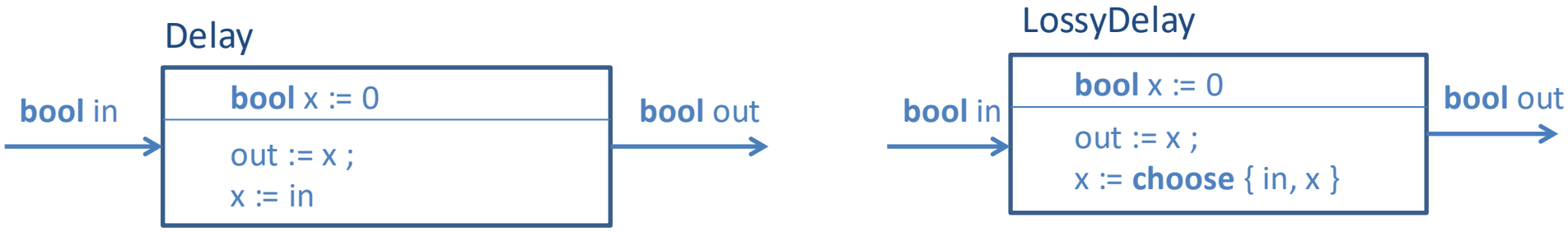


A component is *deterministic* if

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 $s - i/o -> t$ is a reaction

Delay is deterministic, but LossyDelay is not

Deterministic Components



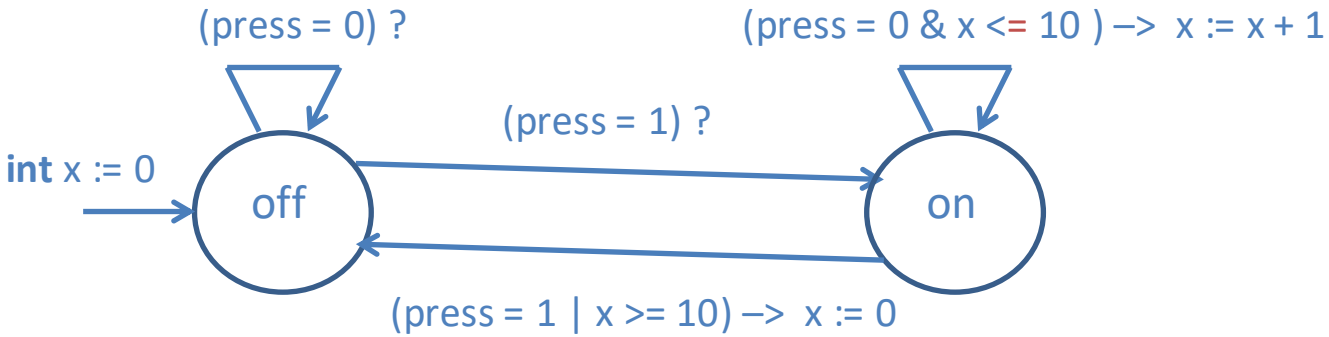
Determinism: same sequence of inputs supplied,
same sequence of outputs observed (predictable, repeatable behavior)

Nondeterminism is

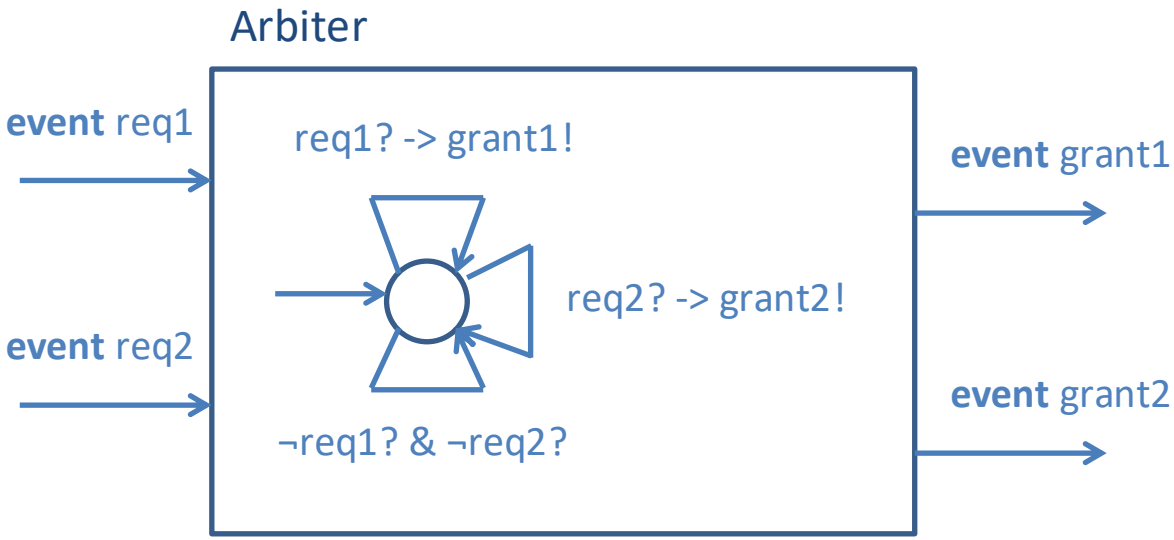
- useful in modeling uncertainty, partial knowledge or abstraction
- different from probabilistic (or random) choice

Modified Switch: What executions are possible?

Input: **bool** press

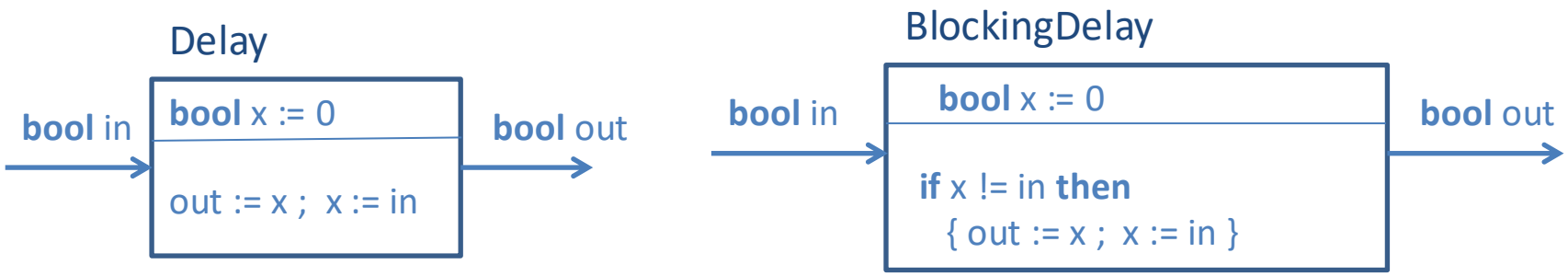


Mixed Notation



What does this component do?

Input-Enabled Components



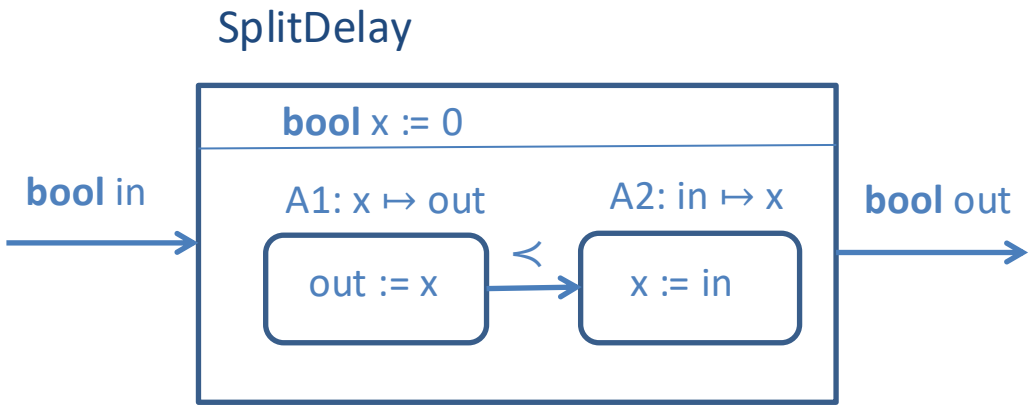
A component is *input-enabled* if for every state s and input i,
there is a state t and an output o such that $s - i/o -> t$ is a reaction

- Delay is input-enabled, but BlockingDelay is not

Non-input-enabled means component is making assumptions about the context in which it is going to be used

- When rest of system is designed, we must check that it indeed satisfies those assumptions

Splitting Reaction Code into Tasks



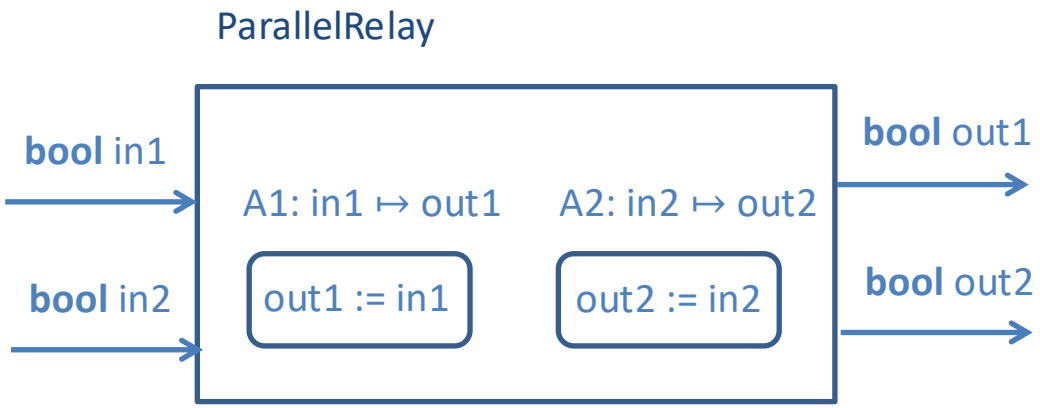
A1 and A2 are *tasks*: atomic blocks of code

- Each task specifies the variables it reads and writes
- **Example:** A1 reads x and writes out

***Task Graph*:** vertices are tasks and edges denote precedence ($\prec$)

- A1 $\prec$ A2 means that A1 should be executed before A2
- Graph must be acyclic

Example Task Graph

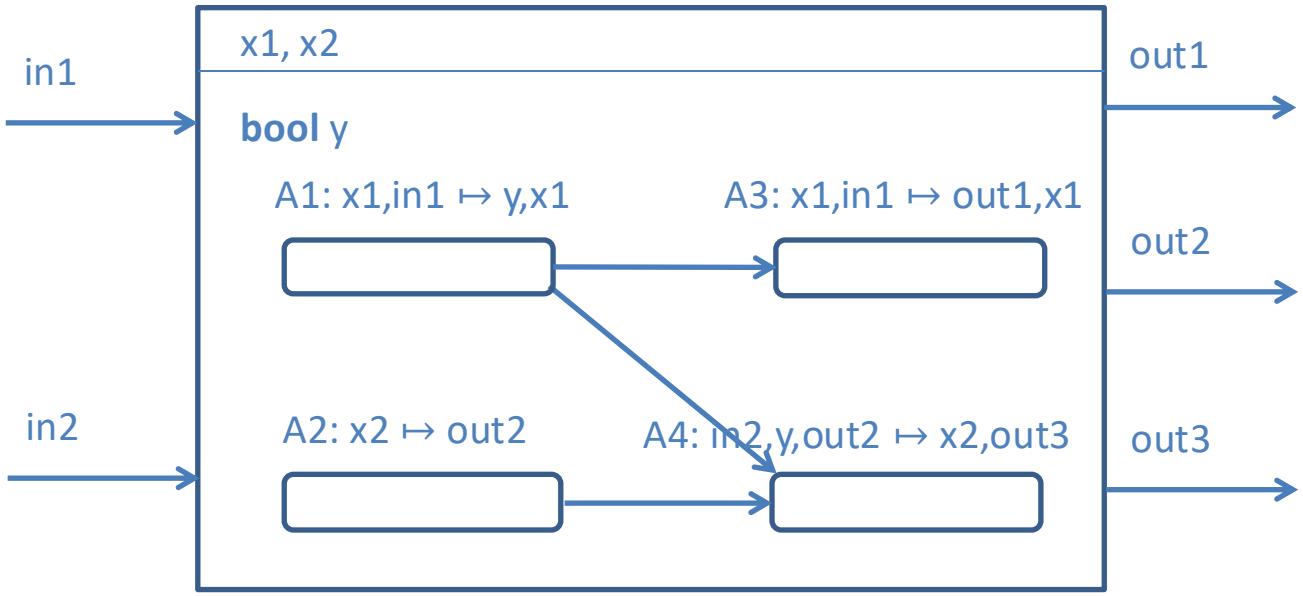


Tasks A1 and A2 are unordered

- Possible *schedules* (linear ordering of tasks): A1, A2 and A2, A1
- All consistent schedules must give the same result

I/O *await dependencies*: out1 awaits in1, out2 awaits in2

Example Task Graph



1. What are possible schedules consistent with precedence constraints?
2. 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

1. a set of tasks, and
2. precedence edges $\prec$ over these tasks

Each task A is specified by:

1. *Read-set* R: a subset of $I \cup S \cup O \cup L$
2. *Write-set* W: a subset of $S \cup O \cup L$
3. *Update*: code to write vars in W based on values of vars in R
   - The set $[\![\text{Update}]\!]$ is a subset of $Q_R \times Q_W$

Requirements on Task Graph (1)

The precedence relation $\prec$ must be acyclic

Notation: $A \prec^+ A'$ means that there is a path from task A to task A' in the task graph using precedence edges

- Relation $\prec^+$ is the *transitive closure* of the relation $\prec$

Task schedule: Total ordering $A_1, A_2, \ldots, A_n$ of all the tasks that is consistent with the precedence edges

- If $A \prec A'$, then A must appear before A' in the ordering
- Multiple schedules are possible
- If $A \prec^+ A'$ then A must appear before A' in every schedule

Note: 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, y is available to the rest of the system

If task A writes output y, then y *awaits* an input variable x, written $y \succ x$, if

either the task A reads x

or some another task A' such that $A' \prec^{+} A$ 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 $A' \prec^{+} A$

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/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 $A \prec^+ A'$ or $A' \prec^+ A$

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

Task Properties

1. Task $A = (R, W, Update)$ is *deterministic* if for every value $u \in Q_R$ there is a unique value $v \in Q_W$ such that $(u, v) \in [\![Update]\!]$

   If all tasks of a component are deterministic, what can we conclude about the component itself?

2. Task $A = (R, W, Update)$ is *input-enabled* if for every value $u \in Q_R$ there exists at least one value $v \in Q_W$ such that $(u, v) \in [\![Update]\!]$

   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