Cyber-Physical Systems (CSC.T431)

Synchronous Model (2)

Instructor: Takuo Watanabe (Department of Computer Science)

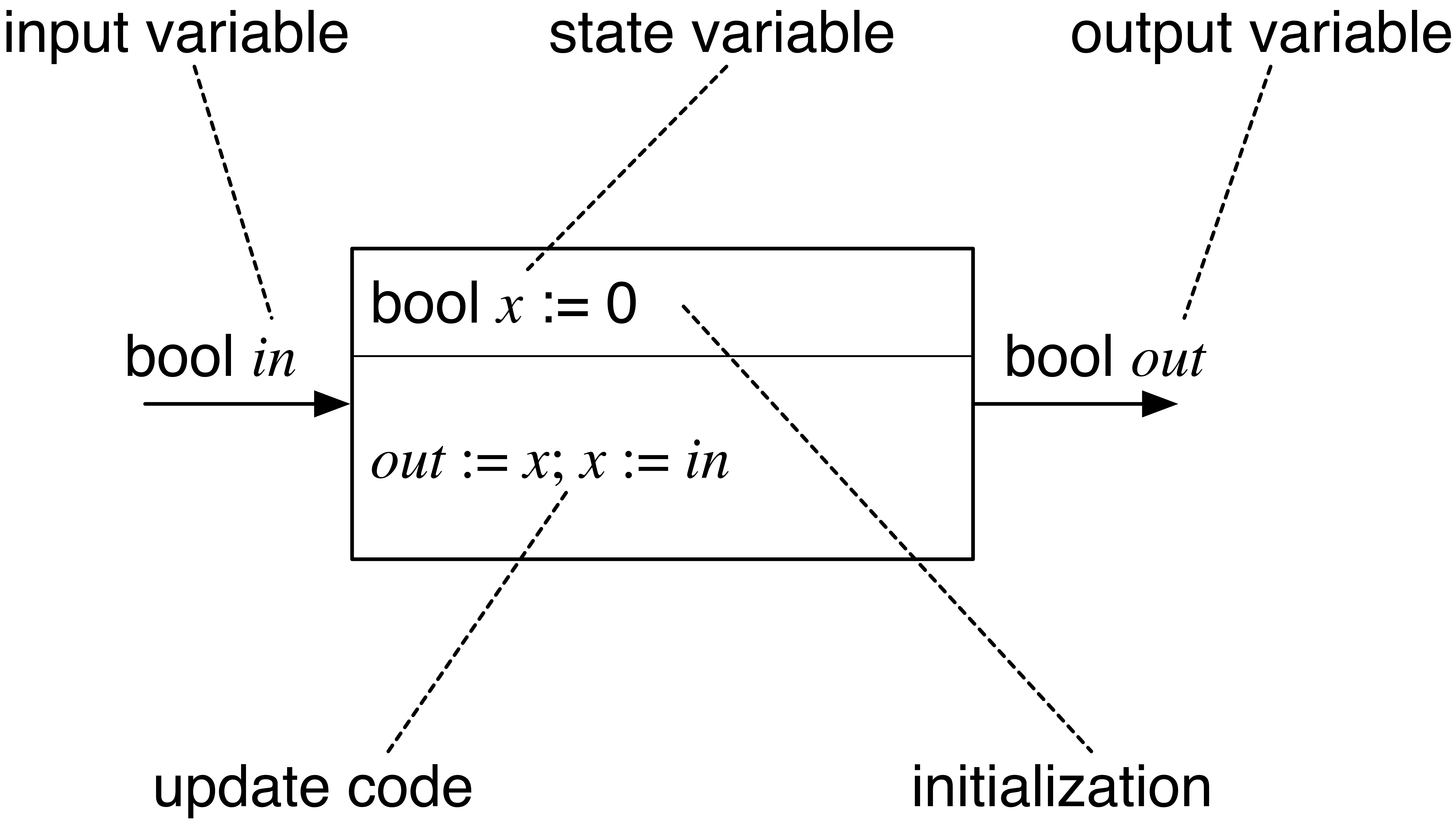
Agenda

1. Synchronous Model (2)

Course Support & Material

- Slides: OCW-i
- Course Web: https://titech-cps.github.io
- Course Slack: titech-cps.slack.com

Ex. Delay

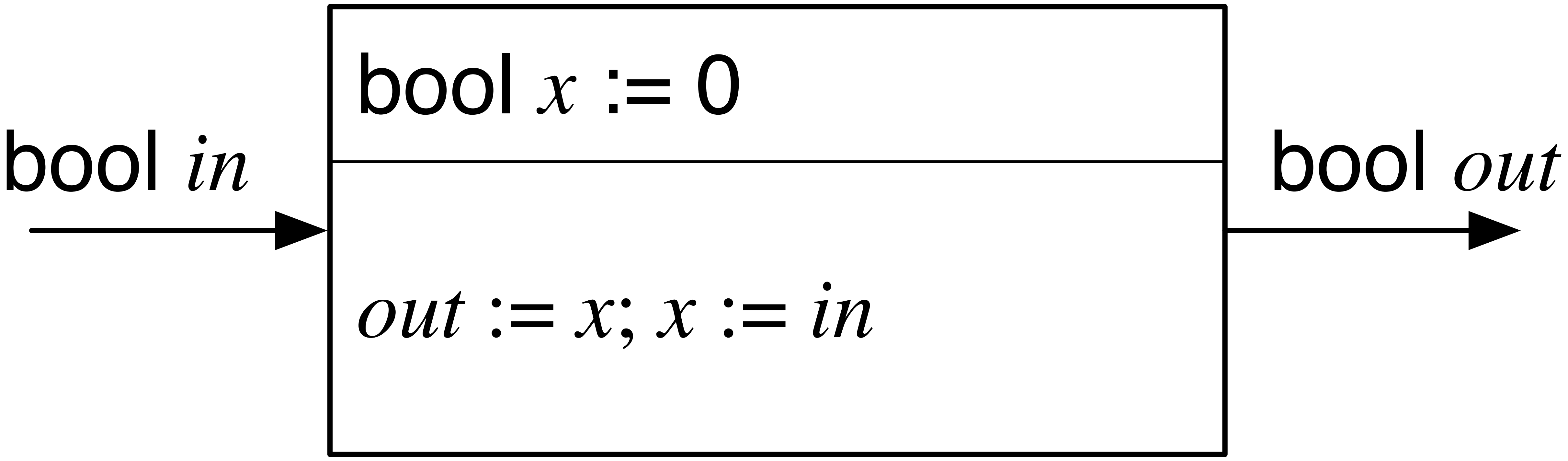


input variable
state variable
output variable
bool x := 0
bool in
bool out
out := x; x := in
update code
initialization

Ex. Delay

$$x \xrightarrow{in/out} x'$$

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



bool in

bool x := 0

$out := x; x := in$

bool out

round	**init**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
in		0	1	0	1	1	0	0	1
x		0	0	1	0	1	1	0	0
x'	0	0	1	0	1	1	0	0	1
out		0	0	1	0	1	1	0	0

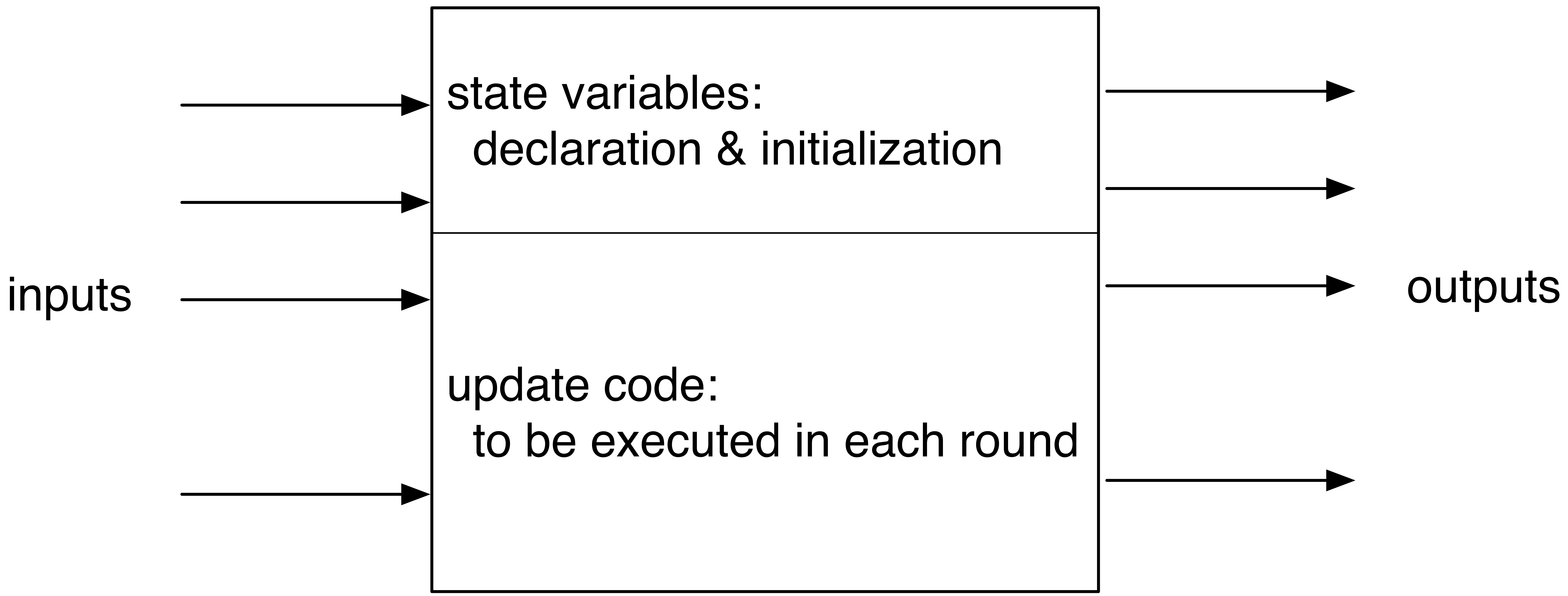
Synchrony Hypothesis

- We assume that the time needed to execute the update code is negligible compared to delay between successive input arrivals (or time between rounds).
- Logically, we can say that the execution of update code takes zero time, and the production of output and reception of inputs occurs at the same time.

- Cf. Synchronous Languages
  - Programming languages based on the synchrony hypothesis
  - e.g., Esterel, Lustre, Signal, Ceu

Synchronous Reactive Components

$$C = (I, O, S, [[Init]], [[React]])$$



Synchronous Reactive Components

$$C = (I, O, S, [[Init]], [[React]])$$

- A synchronous reactive component C is described by
    - a finite set I of typed input variables defining the set Q_I of inputs,
    - a finite set O of typed output variables defining the set Q_O of outputs,
    - a finite set S of typed state variables defining the set Q_S of states,
    - an initialization $Init$ defining the set $[[Init]] \subseteq Q_S$ of initial states, and
    - a reaction description $React$ defining the set $[[React]]$ of reactions of the form $s \overset{i/o}{\rightarrow} t$ where s, t are states, i is an input, and o is an output.

- Ex. (Delay) $I = \{in\}, O = \{out\}, S = \{x\}$

Valuations

disjoint sum

values of type τ_v

$$D = \sum_{v \in V} D_{\tau_v}$$

- V : finite set of typed variables
- $q : V \to D$: a valuation over V
- For all $v \in V, q(v)$ is a value of type τ_v (τ_v : type of v)
- Q_V : set of all valuations over V

finite function
as a set of tuples

$V = \{x, y\}$

$q : V \to N$

$q(x) = 1, q(y) = 2$

$\Downarrow$

$q = \{(x,1), (y,2)\}$

- Ex. (Delay) $I = \{in\}, O = \{out\}, S = \{x\}$
  - $O_I = \{\{(in,0)\}, \{(in,1)\}\}$, $Q_O = \{\{(out,0)\}, \{(out,1)\}\}$, $Q_O = \{\{(x,0)\}, \{(x,1)\}\}$
  - $[[Init]] = \{\{(x,0)\}\}$

Reaction

$$s_0 \xrightarrow{i_1/o_1} s_1 \xrightarrow{i_2/o_2} s_2 \xrightarrow{i_3/o_3} s_3 \cdots s_{k-1} \xrightarrow{i_k/o_k} s_k$$

- State : $s_j \in Q_S \quad (0 \leq j \leq k)$
- Input : $i_j \in Q_I \quad (1 \leq j \leq k)$
- Output : $o_j \in Q_O \quad (1 \leq j \leq k)$
- Initial state : $s_0 \in [[Init]]$
- Reaction : $s_{j-1} \xrightarrow{i_j/o_j} s_j \in [[React]] \quad (1 \leq j \leq k)$

$$C = (I, O, S, [[Init]], [[React]])$$

Ex. Delay

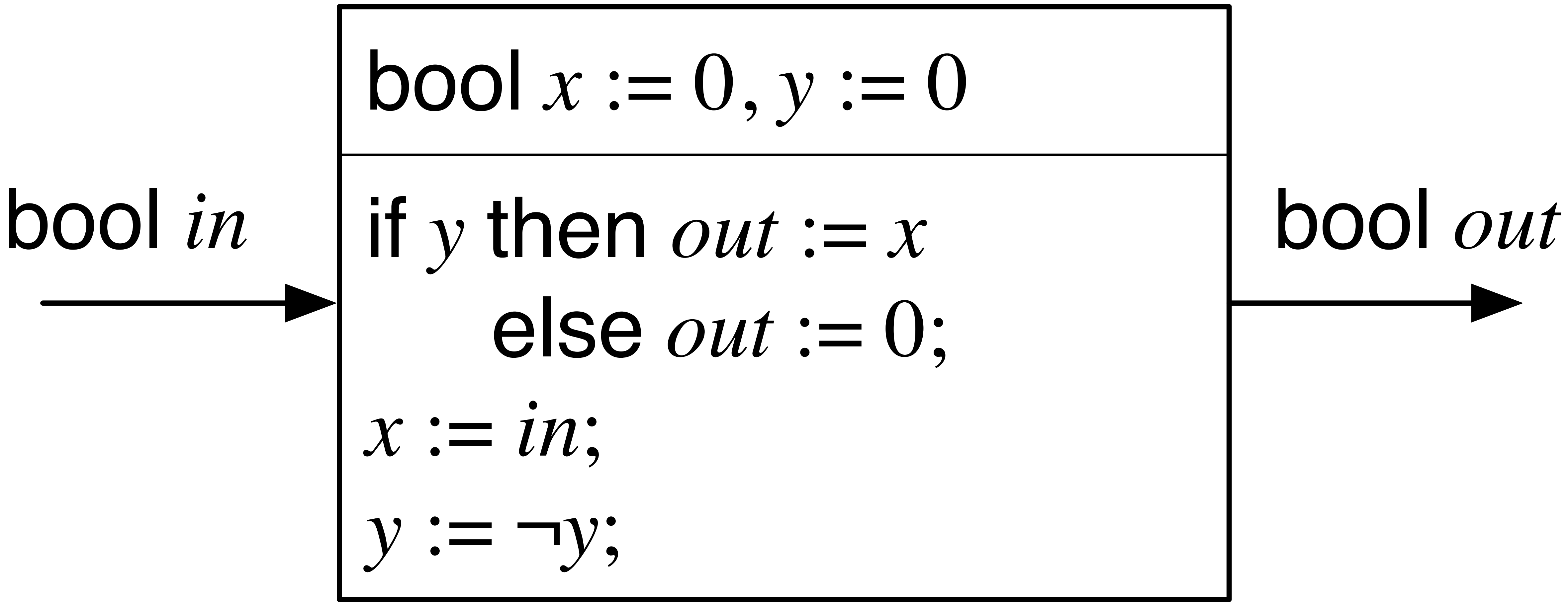
Full description

$$\begin{aligned}
I &= \{in\},\ O = \{out\},\ S = \{x\} \\
Q_I &= \{\{(in,0)\},\{(in,1)\}\},\ Q_O = \{\{(out,0)\},\ \{(out,1)\}\},\ Q_S = \{\{(x,0)\},\{(x,1)\}\} \\
[[Init]] &= \{\{(x,0)\}\} \\
[[React]] &= \left\{ \begin{array}{l} \{(x,0)\} \xrightarrow{\{(in,0)\}/\{(out,0)\}} \{(x,0)\},\ \{(x,0)\} \xrightarrow{\{(in,1)\}/\{(out,0)\}} \{(x,1)\}, \\ \{(x,1)\} \xrightarrow{\{(in,0)\}/\{(out,1)\}} \{(x,0)\},\ \{(x,1)\} \xrightarrow{\{(in,1)\}/\{(out,1)\}} \{(x,1)\} \end{array} \right\}
\end{aligned}$$

Abridged description

$$\begin{aligned}
I &= \{in\},\ O = \{out\},\ S = \{x\} \\
Q_I &= \{0,1\},\ Q_O = \{0,1\}, Q_S = \{0,1\} \\
[[Init]] &= \{0\} \\
[[React]] &= \left\{0 \xrightarrow{0/0} 0,\ 0 \xrightarrow{1/0} 1,\ 1 \xrightarrow{0/1} 0,\ 1 \xrightarrow{1/1} 1\right\}
\end{aligned}$$

Ex. OddDelay



bool in

bool $x := 0, y := 0$

```
if $y$ then $out := x$
    else $out := 0$;
$x := in$;
$y := \neg y$;
```

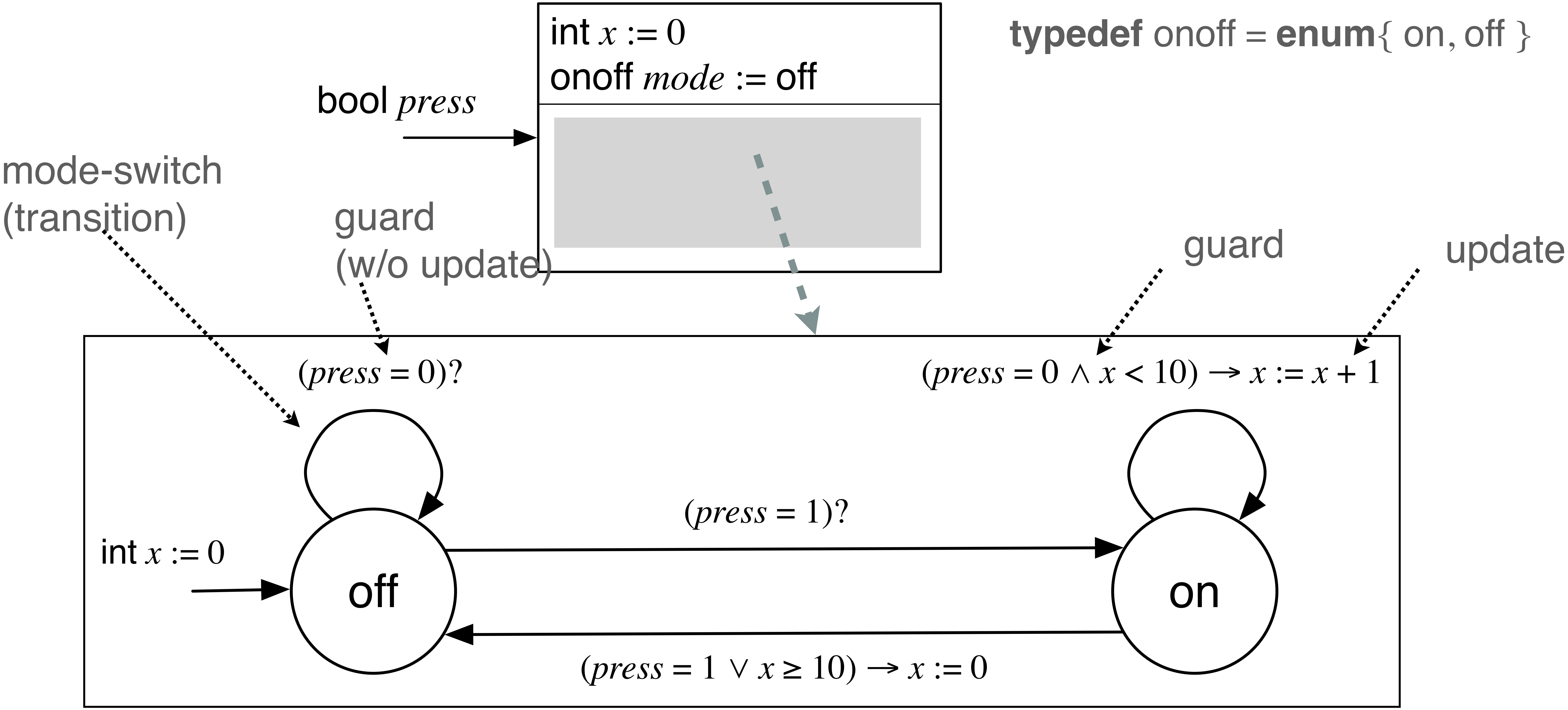
bool out

round	**init**	**1**	**2**	**3**	**4**	**5**	**6**
in		0	1	1	0	1	1
x		0	0	1	1	0	1
x'	0	0	1	1	0	1	1
y		0	1	0	1	0	1
y'	0	1	0	1	0	1	0
out		0	0	0	1	0	1

Ex. Switch

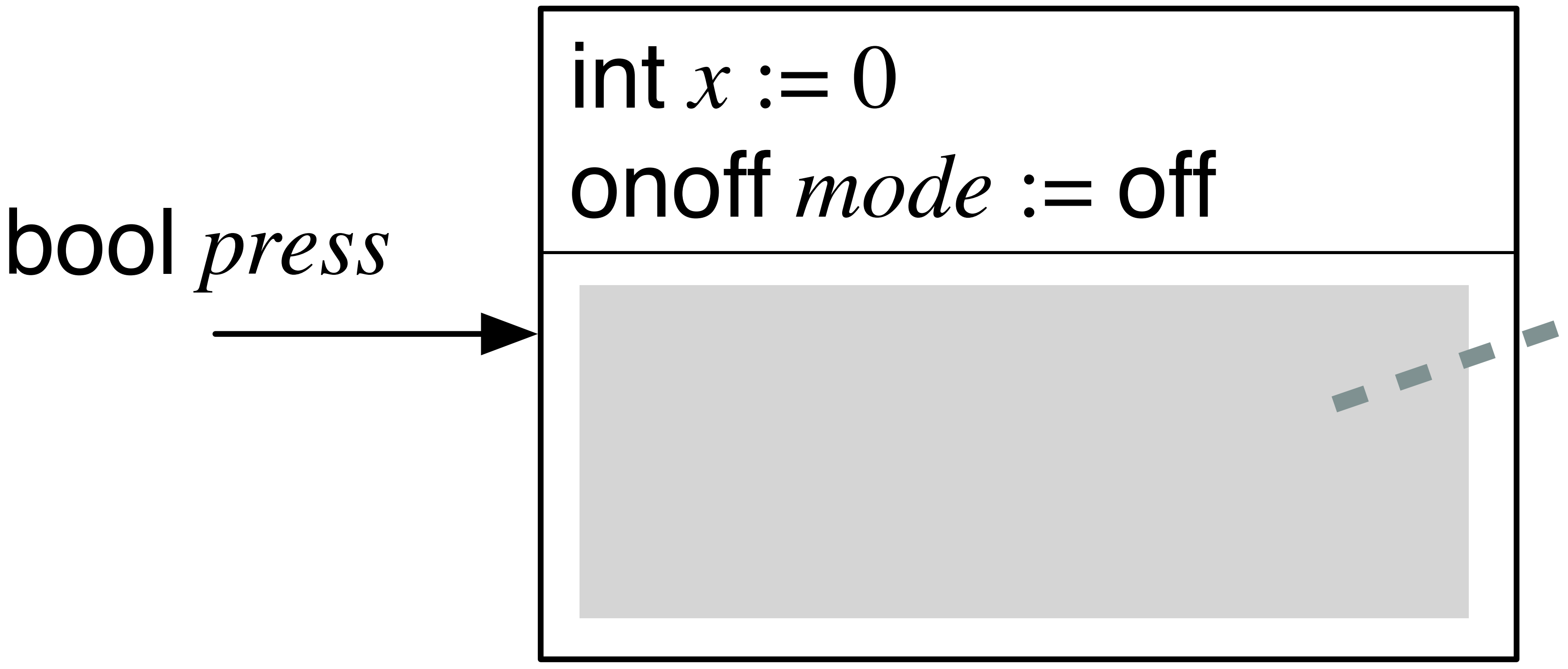
Behavior as an Extended-State Machine

typedef onoff = enum{ on, off }

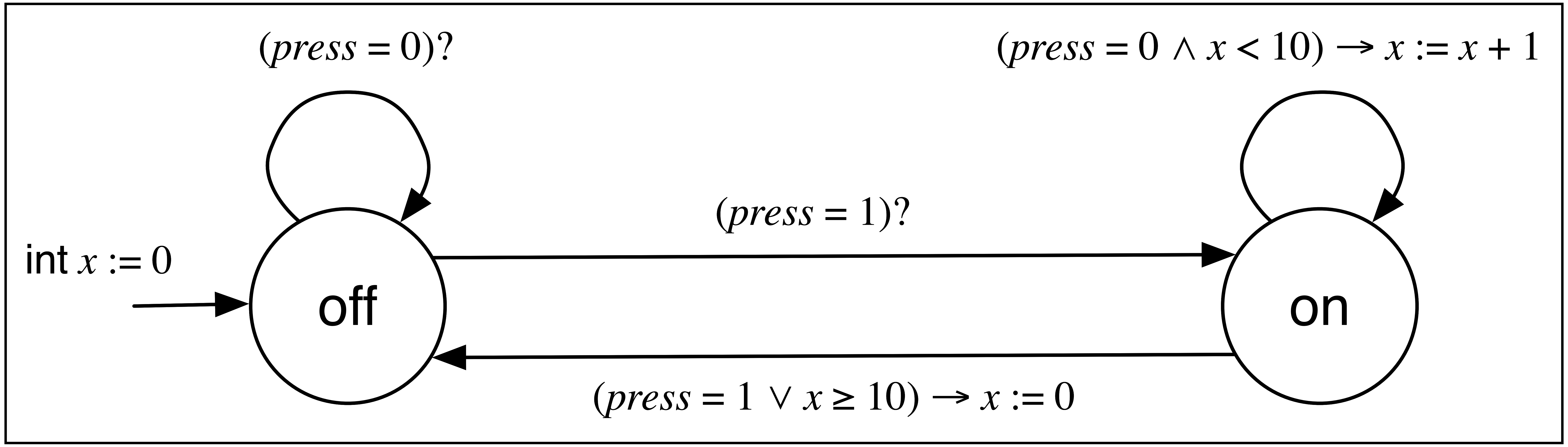


Ex. Switch

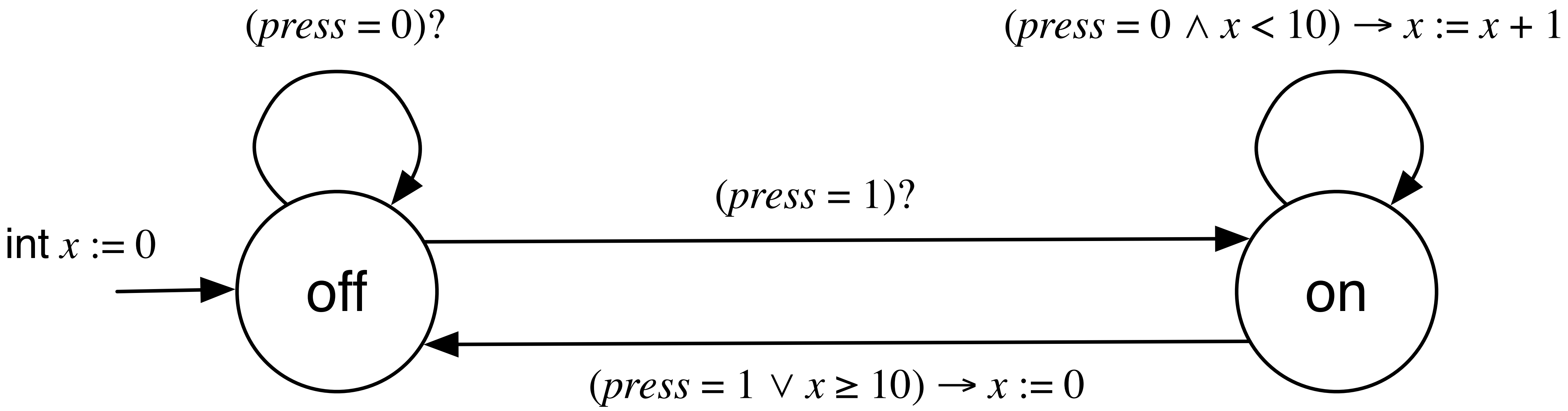
Behavior as a program



```
if $mode$ = off then
    if $press = 0$ then
        skip    // do nothing
    else
        $mode$ = on
else  // $mode$ = on
    if $press = 0 \wedge x < 10$ then
        $x := x + 1$
    else  // $press = 1 \vee x \geq 10$
      { $x := 0$; $mode :=$ off };
```


$(press = 0)?$
$(press = 0 \wedge x < 10) \rightarrow x := x + 1$
$(press = 1)?$
int $x := 0$
off
on
$(press = 1 \vee x \geq 10) \rightarrow x := 0$

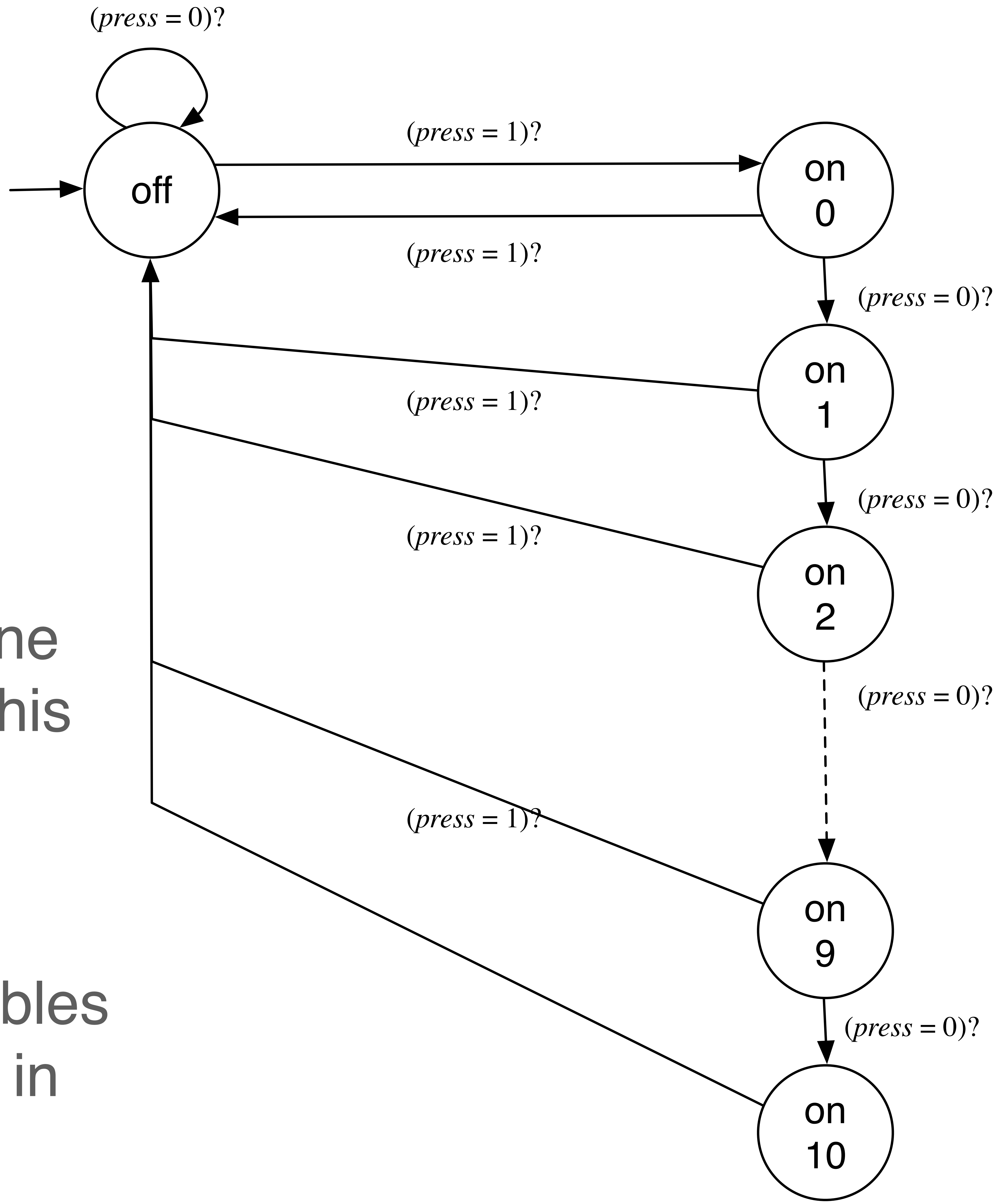
Extended-State Machine



(press = 0)?
(press = 0 ∧ x < 10) → x := x + 1
(press = 1)?
int x := 0
off
on
(press = 1 ∨ x ≥ 10) → x := 0

An *extended-state machine* is a state machine augmented with extra state variables (x for this example).

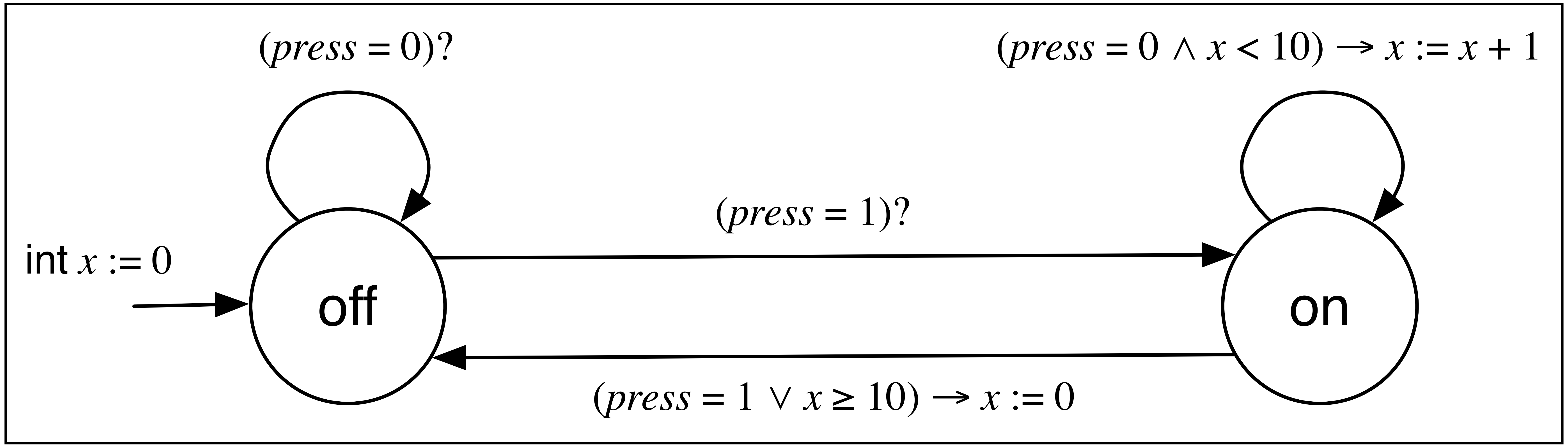
If we don't use such variables, we need to represent all the possible values of the variables as the states of the state machine as shown in the figure on the right.



Ex. Switch

possible scenario

round	**init**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
$press$		0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
x		0	0	0	1	2	0	0	0	0	1	2	3	4	5	6	7	8	9	10	0
x'	0	0	0	1	2	0	0	0	0	1	2	3	4	5	6	7	8	9	10	0	0
$mode$		off	off	on	on	on	off	off	off	on	on	on	on	on	on	on	on	on	on	on	off
$mode'$	off	off	on	on	on	off	off	off	on	on	on	on	on	on	on	on	on	on	on	off	off



$(press = 0)?$
$(press = 0 \wedge x < 10) \rightarrow x := x + 1$
$(press = 1)?$
int $x := 0$
off
on
$(press = 1 \vee x \geq 10) \rightarrow x := 0$

Ex. Switch

Formal Description

$$Q_I = \{\{(press{,}0)\}, \{press{,}1\}\}$$

$$Q_S = \{\{(mode, s), (x, n)\} \mid s \in \{\mathsf{off}, \mathsf{on}\} \wedge n \in \mathbb{Z}\}$$

$$\begin{aligned}
C &= (I, O, S, [[Init]], [[React]]) \\
I &= \{press\},\ O = \emptyset,\ S = \{mode, x\} \\
Q_I &= \{0, 1\},\ Q_O = \{\emptyset\},\ Q_S = \{(s, n) \mid s \in \{\mathsf{off}, \mathsf{on}\} \wedge n \in \mathbb{Z}\} \\
[[Init]] &= \{(\mathsf{off}, 0)\} \\
[[React]] &= \{(\mathsf{off}, n) \xrightarrow{0/} (\mathsf{off}, n) \mid n \in \mathbb{Z}\} \\
&\cup \{(\mathsf{off}, n) \xrightarrow{1/} (\mathsf{on}, n) \mid n \in \mathbb{Z}\} \\
&\cup \{(\mathsf{on}, n) \xrightarrow{0/} (\mathsf{on}, n+1) \mid n \in \mathbb{Z} \wedge n < 10\} \\
&\cup \{(\mathsf{on}, n) \xrightarrow{0/} (\mathsf{off}, 0) \mid n \in \mathbb{Z} \wedge n \geq 10\} \\
&\cup \{(\mathsf{on}, n) \xrightarrow{1/} (\mathsf{off}, 0) \mid n \in \mathbb{Z}\}
\end{aligned}$$

$\mathbb{Z}$: set of integers

Note: Q_S is defined as an infinite set. But the number of states reachable from the initial state is finite.

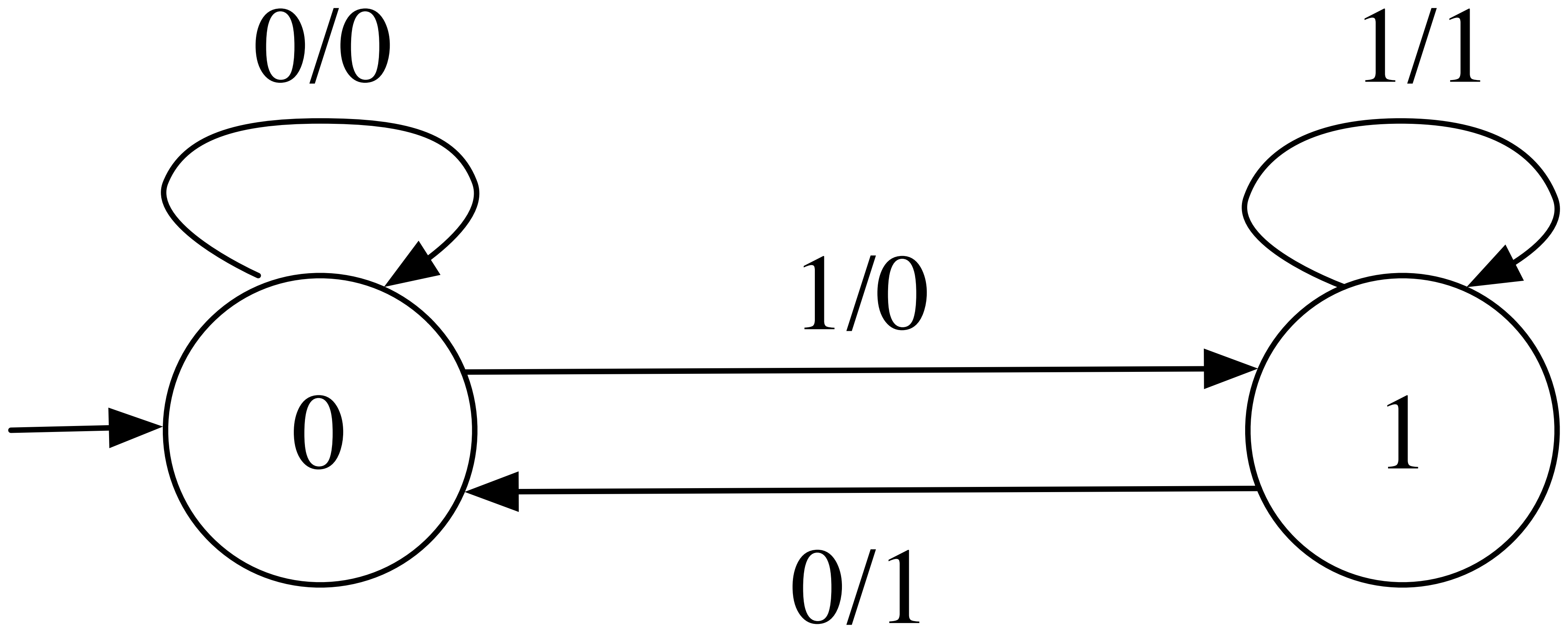
Finite-State Component

- A synchronous reactive component C is finite-state if the types of its input, output, and state variables are finite.
    - Finite types: Boolean, enumerated types
    - Ex. Delay is finite-state

- How about Switch?
    - The state variable x has type int which is not finite.
    - However, the value of x never exceed 10 (and never goes below 0).
    - Thus, we can replace the type of x with a finite range-type int[0, 10] and make Switch finite.

Mealy Machine Representation

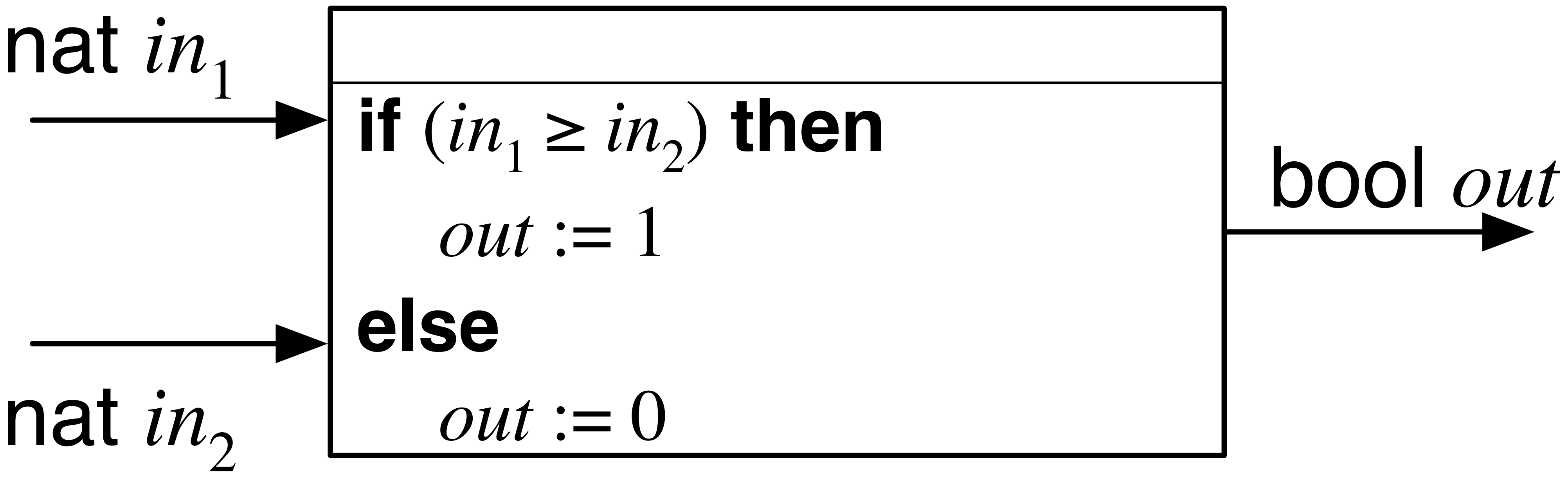
- The behavior of a finite component C can be represented as a labeled finite graph (Mealy machine).
  - The nodes of the graph are the states of C.
  - If $s \in [[Init]]$, there is a sourceless edge to s.
  - If $s \overset{i/o}{\to} t \in [[React]]$, there is an edge from s to t labeled with i/o.
- Ex. Delay



Combinatorial Components

Stateless Components

- A synchronous reactive component $C = (I, O, S, [[Init]], [[React]])$ is combinatorial if Q_S is a singleton set (typically $S = \emptyset$).
- Ex. Comparator



nat in_1

nat in_2

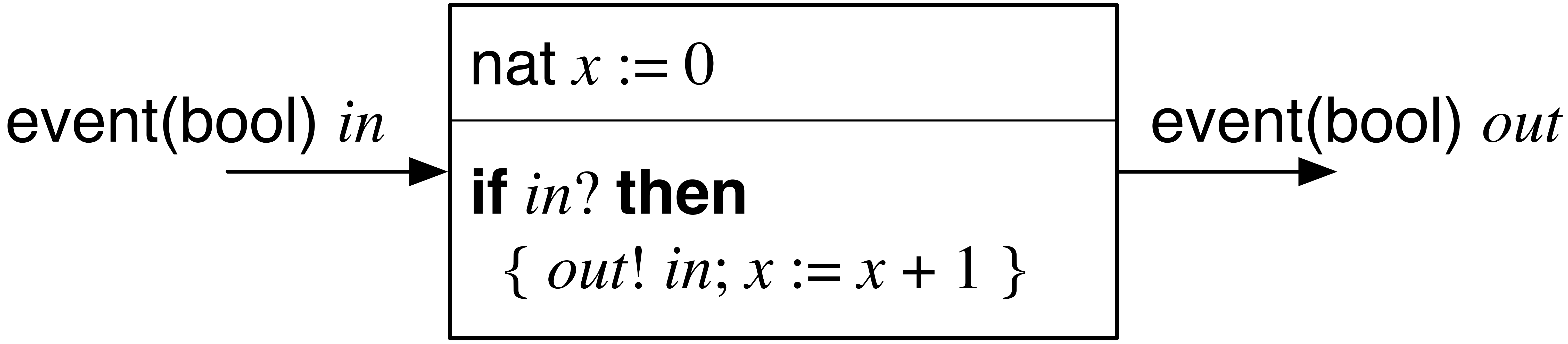
```
if ($in_1 \geq in_2$) then
    $out := 1$
else
    $out := 0$
```

bool out

- Because S is empty, Q_S should be $\{\emptyset\}$. The only element $\emptyset$ of this set corresponds to the unique state $s_\emptyset$ of the component.
- A possible execution of Comparator: $s_\emptyset \xrightarrow{(2,3)/0} s_\emptyset \xrightarrow{(5,1)/1} s_\emptyset \xrightarrow{(40,40)/1} s_\emptyset$

Event-Triggered Components

Ex. TriggeredCopy



event(bool) in

nat $x := 0$

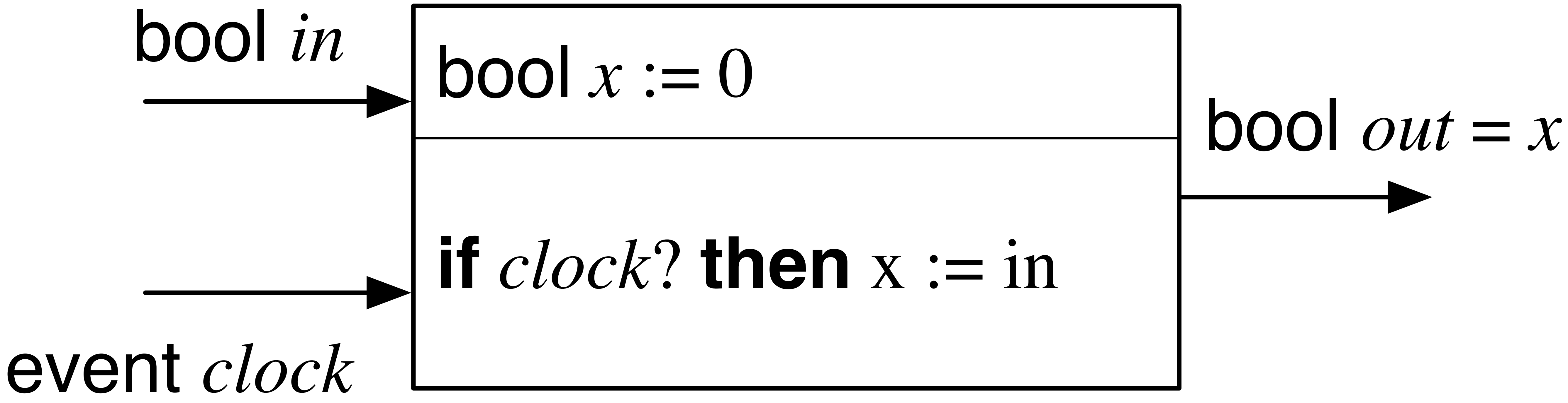
if $in?$ **then**
$\{\ out!\ in;\ x := x + 1\ \}$

event(bool) out

- possible execution: $0 \xrightarrow{\bot/\bot} 0 \xrightarrow{0/0} 1 \xrightarrow{1/1} 2 \xrightarrow{1/1} 2 \xrightarrow{\bot/\bot} 2 \xrightarrow{\bot/\bot} 2 \xrightarrow{1/1} 3$
  - $[[React]] = \{n \xrightarrow{\bot/\bot} n \mid n \in \mathbb{N}\} \cup \{n \xrightarrow{b/b} n + 1 \mid b \in \{0{,}1\} \wedge n \in \mathbb{N}\}$
- Event types
  - event : enumerated type of events $\{\ \top\ ,\ \bot\ \}$ where $\top$ and $\bot$ denotes denotes the presence and absence of an event respectively.
  - event(τ) : event with type τ ($= \tau + \{\ \bot\ \}$). e.g., event(bool) has values 0, 1, and $\bot$.

Event-Triggered Components

Ex. ClockedCopy



- possible execution: $0 \xrightarrow{(1,\bot)/0} 0 \xrightarrow{(1,\top)/1} 1 \xrightarrow{(0,\bot)/1} 1 \xrightarrow{(0,\bot)/1} 1 \xrightarrow{(0,\top)/0} 0$
- The output is latched until the next clock event ($\top$) comes.
- The output variable out is implemented as an alias of the state variable x.

Event-Triggered Components

Definition

- For a synchronous reactive component $C = (I, O, S, [[Init]], [[React]])$, a set $J \subseteq I$ of input variables is said to be a *trigger* if:
  1. every input variable in J is of type event,
  2. every output variable either is latched or is of type event, and
  3. if i is an input with all events in J absent (i.e., $\forall x \in J \,.\, i(x) = \perp$), then for all states s, if $s \xrightarrow{i/o} t$ is a reaction, then $s = t$ and $o(y) = \perp$ for every output variable y of event type.
- A component C is said to be *event-triggered* if there exists a subset $J \subseteq I$ of its input variables such that J is a trigger for C.

Deterministic Components

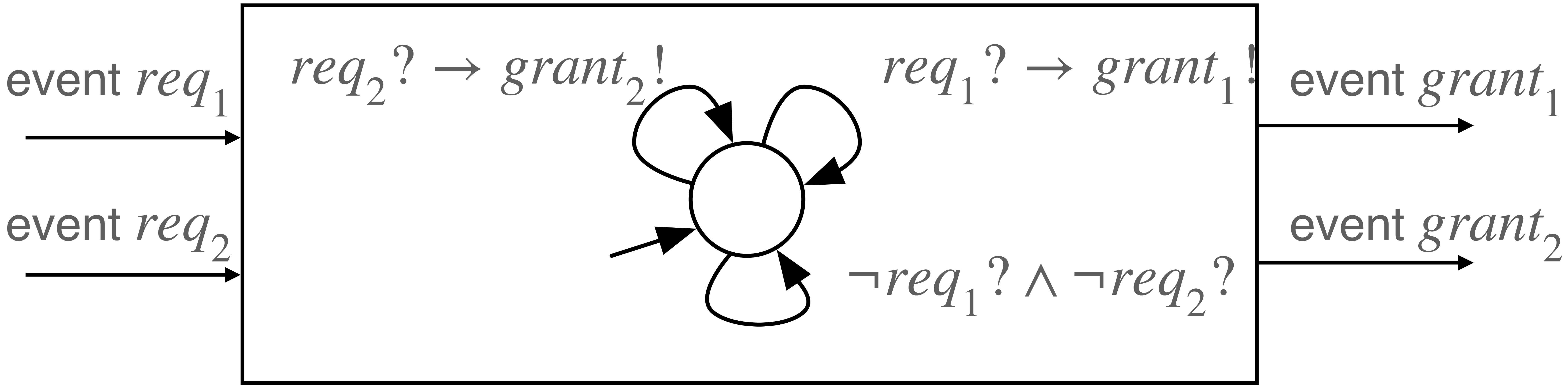
- A synchronous reactive component C is *deterministic* if:

1. C has a single initial state, and
2. for every state s and every input i, there is precisely one output o and one state t such that $s \xrightarrow{i/o} t$ is a reaction of C.

- Delay, Switch, Comparator, TriggeredCopy, and ClockedCopy are deterministic.

Nondeterministic Components

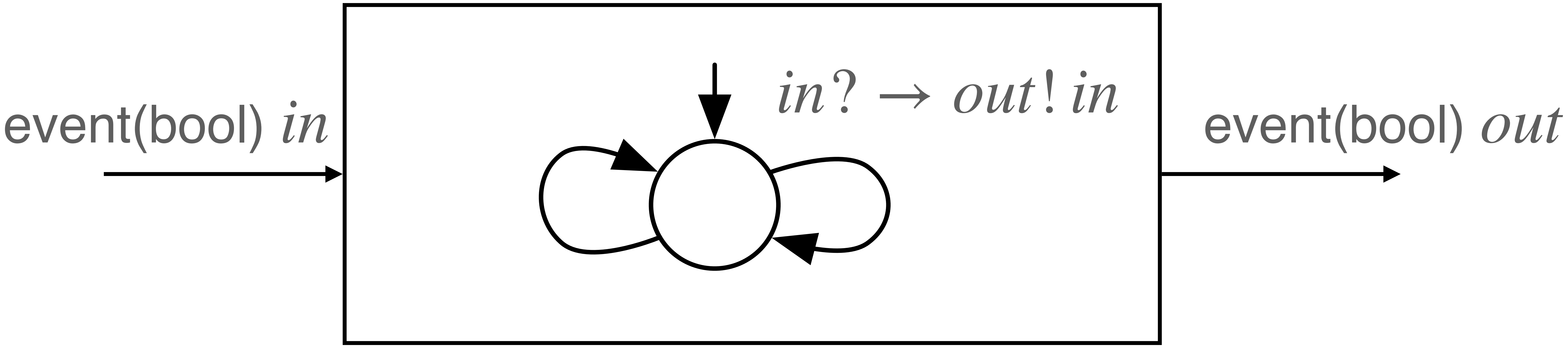
- C is nondeterministic if it is not deterministic. In other words, it may respond with different output sequences for the same input sequence.
- Ex. Arbiter



event req_1
event req_2
$req_2? \rightarrow grant_2!$
$req_1? \rightarrow grant_1!$
$\neg req_1? \wedge \neg req_2?$
event $grant_1$
event $grant_2$

Nondeterministic Components

- Ex. LossyCopy (combinatorial, event-trigger, nondeterministic)



- Possible execution: $S_\varnothing \xrightarrow{0/0} S_\varnothing \xrightarrow{1/\bot} S_\varnothing \xrightarrow{\bot/\bot} S_\varnothing \xrightarrow{1/1} S_\varnothing \xrightarrow{\bot/\bot} S_\varnothing \xrightarrow{0/\bot} S_\varnothing$
- The update code can be written as: if $in? \wedge$ choose$(0,1)$ then $out!\,in$
  - choose(x, y) : nondeterministic choice of x and y

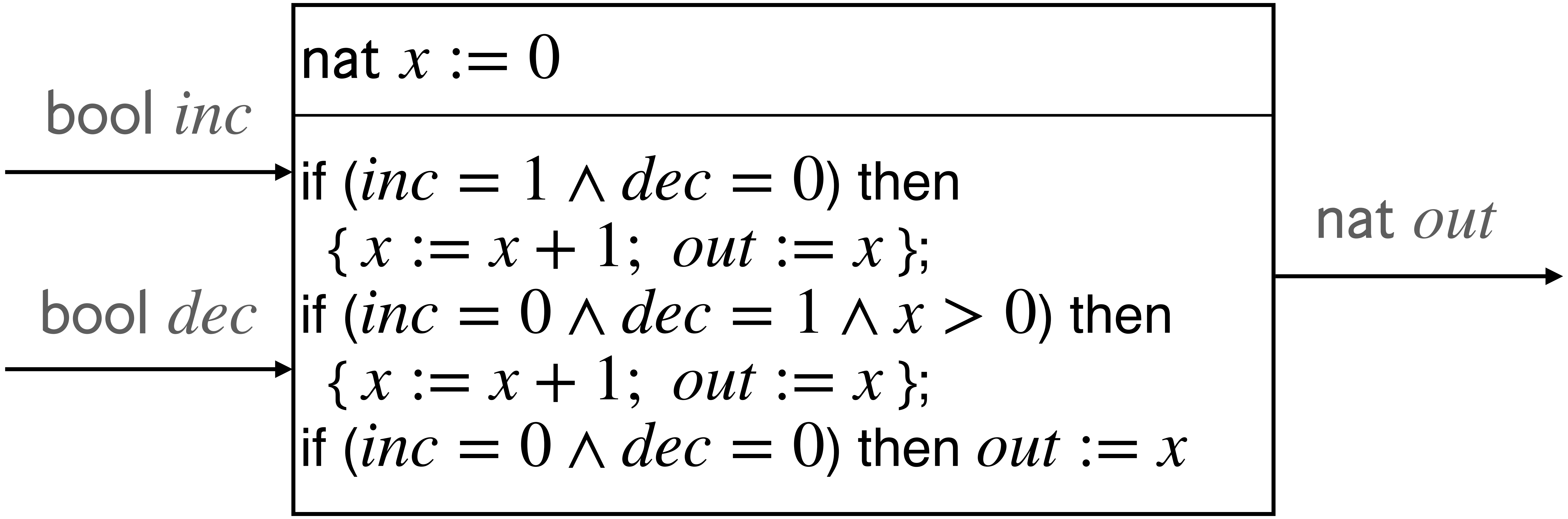
Input-Enabled Components

- For a synchronous reactive component C, an input i is *enabled* in a state s if there exists an output o and a state t such that $s \xrightarrow{i/o} t$ is a reaction of C.
- The component is *input-enabled* if every input is enabled in every state. In other words, in every state and for every input, C has at least one reaction.

- Delay, OddDelay, Switch, Comparator, TriggeredCopy, ClockedCopy, Arbiter, and LossyCopy are input-enabled.

Ex. Counter

An example with input assumptions

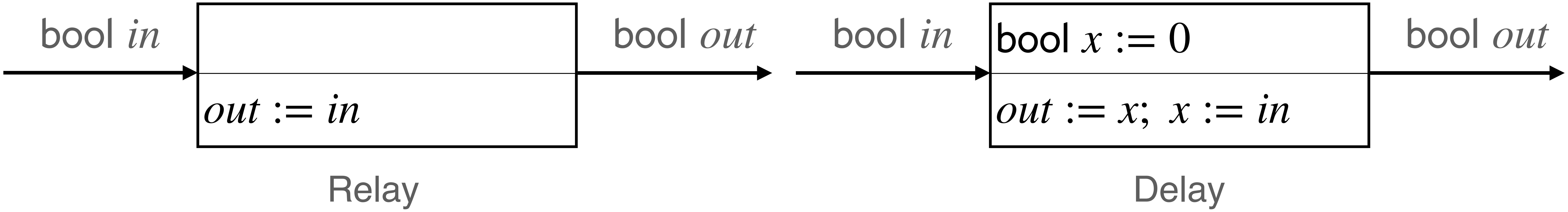


```
nat $x := 0$

if ($inc = 1 \wedge dec = 0$) then
  {$x := x + 1;\ out := x$};
if ($inc = 0 \wedge dec = 1 \wedge x > 0$) then
  {$x := x + 1;\ out := x$};
if ($inc = 0 \wedge dec = 0$) then $out := x$
```

- The component behaves as a non-negative counter.
  - There are no reactions for input $inc = 1 \wedge dec = 1$.
  - When $x = 0$, there are no reactions for input $inc = 0 \wedge dec = 1$.

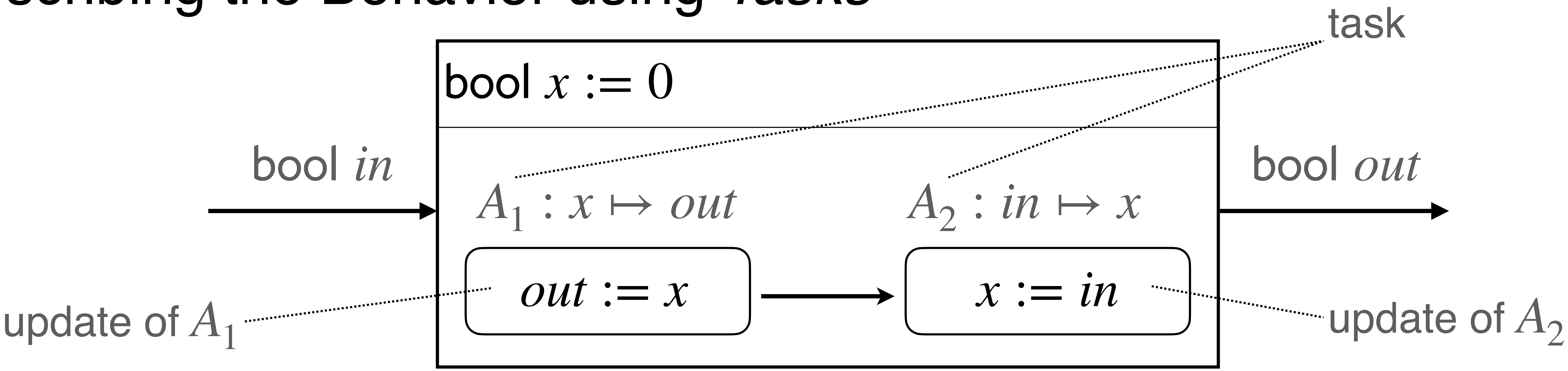
Relay and Delay



- Both components have the same input/output variables.
- *Intra-round dependency* of output variables on input variables
  - In a given round, the output of Delay does not depend on its input in that round, whereas Relay can produce its output only after reading the input for the current round.
  - I.e., for each round, the output of Relay must await its input.
- How can we make such dependencies clear?

Ex. SplitDelay

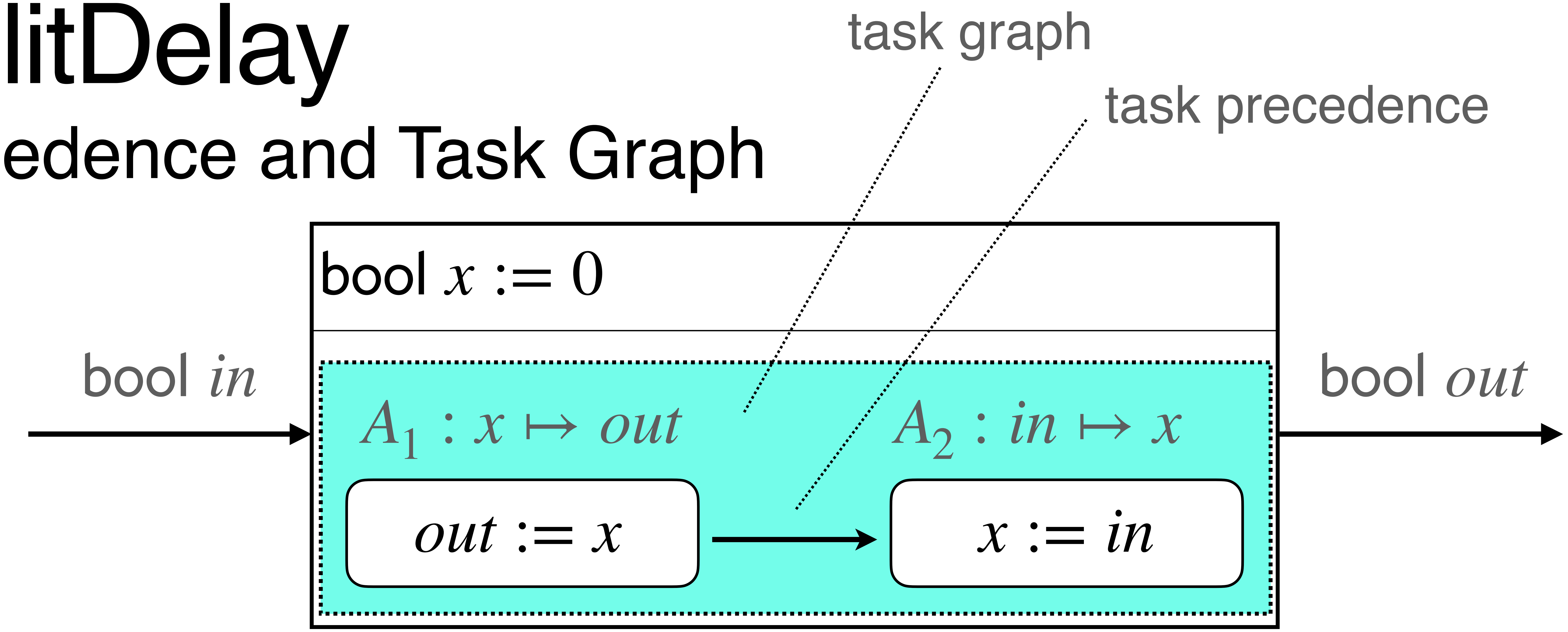
Describing the Behavior using *Tasks*



- A task $A : x_1, x_2, \ldots, x_m \mapsto y_1, y_2, \ldots, y_n$ is a triple $(R, W, [[Update]])$.
    - $R = \{x_1, x_2, \ldots, x_m\}$ (read-set), $W = \{y_1, y_2, \ldots, y_n\}$ (write-set)
    - $[[Update]] \subseteq Q_R \times Q_W$ (update)
- SplitDelay
    - $A_1 : x \mapsto out, R_1 = \{x\}, W_1 = \{out\}, A_2 : in \mapsto x, R_2 = \{in\}, W_2 = \{x\}$

Ex. SplitDelay

Task Precedence and Task Graph



- Task precedence: $A_1 \prec A_2$ (A_1 should be executed before A_2)
  - There is no task A such that $A \prec^+ A$ (whre $\prec^+$ is the transitive closure of $\prec$).
- Task graph: directed graph whose nodes are tasks and edges are task prec.
  - A task graph should be a DAG.

Await Dependencies

between output and input variables

- Output variables written by a task must await the input variables this task reads, and if $A_1 \prec^+ A_2$, then the output variables written by A_2 must await the input variables read by A_1.
- Notation: $y \succ x$ iff the output variable y awaits the input variable x.
- Examples:
  - In SplitDelay, out does not await in because A_1 does not read in and there are not tasks A such that $A \prec A_1$ and in is read by A.
  - In Relay, clearly out awaits in.
  - We can say that Delay consists of a single task $A : in, x \mapsto x, out$. Thus, using this construction, out awaits in.

Task Graph and Await Dependencies (1/2)

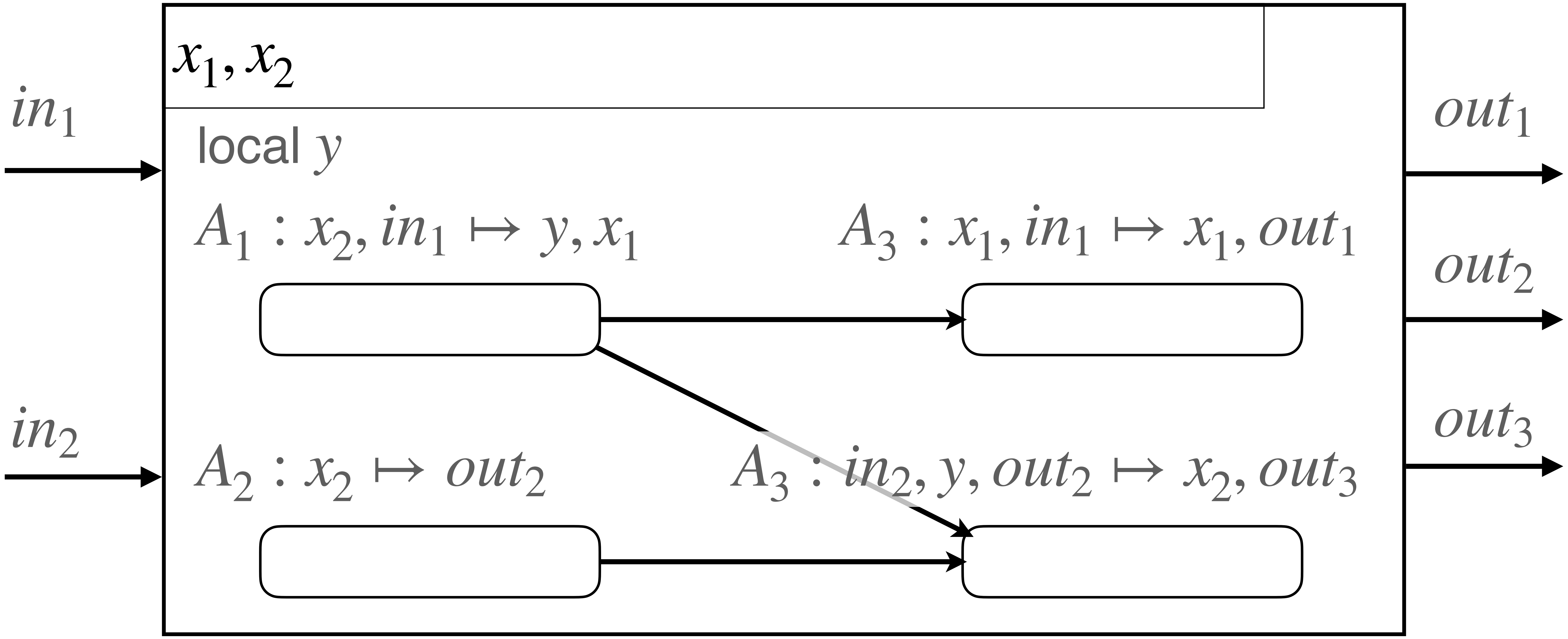
- For a synchronous reactive component $C = (I, O, S, [[Init]], [[React]])$, a task-graph description of the reactions using a set L of local variables consists of a set of tasks and a binary precedence relation $\prec$ over these tasks. Each task A has a read-set $R \subseteq I \cup S \cup O \cup L$, a write-set $W \subseteq O \cup S \cup L$, and an update description $Update$ with $[[Update]] \subseteq Q_R \times Q_W$ such that:
  - The precedence relation $\prec$ is acyclic.
  - Each output variable belongs to the write-set fo exactly one task.
  - If an output or a local variable y belongs to the read-set of a task A, then there exists a task A' such that y is in the write-set of A' and $A' \prec^+ A$.
  - If a state or a local variable x belongs to the write-set of a task A and also to either the read-set or write-set of a different task A', then either $A \prec^+ A'$ or $A' \prec^+ A$.

Task Graph and Await Dependencies

(2/2)

- For an output variable y and an input variable x, y *awaits* x (also written $y \succ x$), precisely when for the unique task A, such that y belongs to the write-set of A, either x belongs to the read-set of A, or there exists a task A' such that $A' \prec^{+} A$ and x belongs to the read-set of A'.

Task Schedule


x_1, x_2
in_1
local y
out_1
$A_1 : x_2, in_1 \mapsto y, x_1$
$A_3 : x_1, in_1 \mapsto x_1, out_1$
out_2
in_2
out_3
$A_2 : x_2 \mapsto out_2$
$A_3 : in_2, y, out_2 \mapsto x_2, out_3$

- A task schedule is a linear ordering of all the tasks that is consistent with $\prec$.
- Ex. There are five possible schedules for the above example.
    - $A_1, A_2, A_3, A_4;\ A_1, A_2, A_4, A_3;\ A_1, A_3, A_2, A_4;\ A_2, A_1, A_3, A_4;\ A_2, A_1, A_4, A_3$

Deterministic / Input-Enabled Tasks

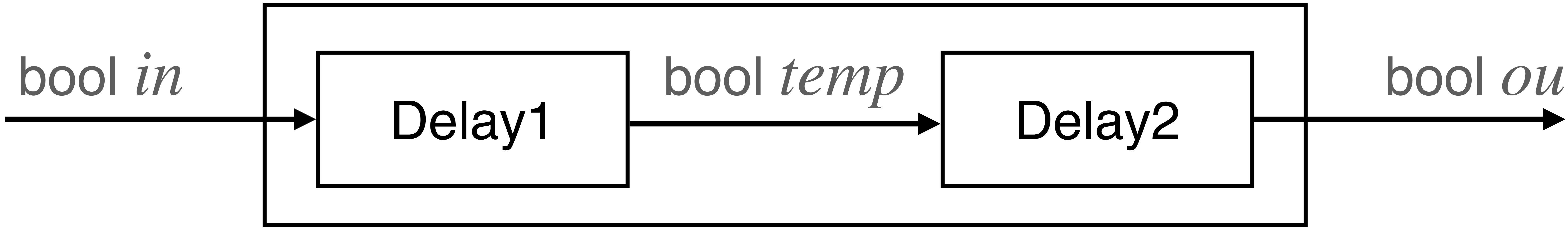
- Deterministic Tasks
    - A task $A = (R, W, [[Update]])$ is *deterministic* if $\forall s \in Q_R \,.\, \exists ! t \in Q_W \,.\, (s, t) \in [[Update]]$.
    - Given values for the variables in the read-set, a deterministic task assigns unique values to the variables in the write-set.
- Input-Enabled Tasks
    - A task $A = (R, W, [[Update]])$ is *input-enabled* if $\forall s \in Q_R \,.\, \exists t \in Q_W \,.\, (s, t) \in [[Update]]$.
    - Given values for the variables in the read-set, an input-enabled task produces at least one result.

Thm.

- If a component has a single initial state and all the tasks in the task-graph are deterministic, the the component is deterministic.
- If all the tasks in the task-graph of a component are input-enabled, then the component is input-enabled.

Composition of Components

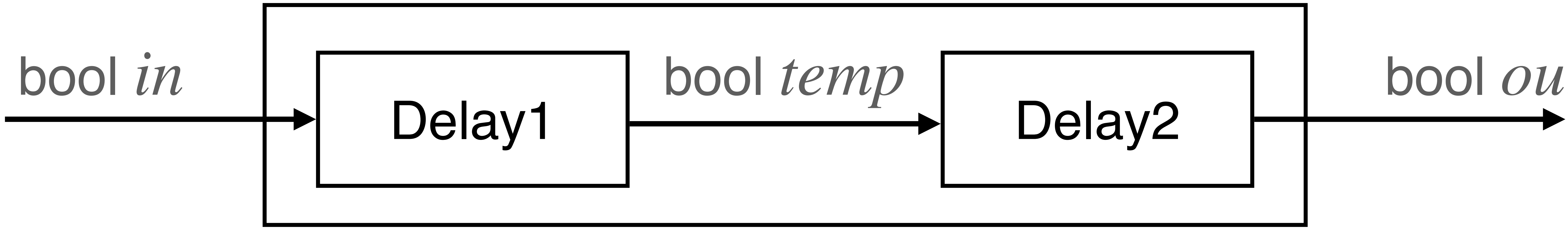
Ex. DoubleDelay from two Delays



- The DoubleDelay is a component with a Boolean input in and a Boolean output out, such that in the first two rounds the output is 0 and in every subsequent round n, the output equals the input in round $n-2$.
- We can construct DoubleDelay by composing two Delay components as the block diagram above shows.

Composition of Components

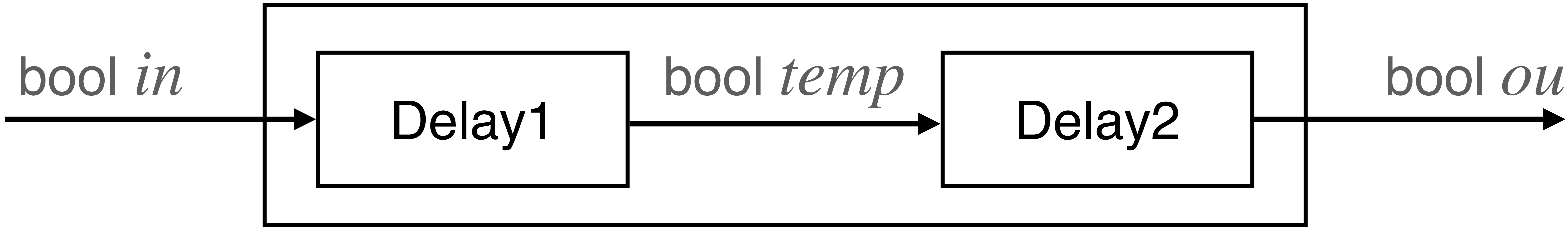
Three Operations on Composition



- Instantiation
  - Delay1 (Delay2) is a copy of Delay except its output (input) variable is renamed to *temp*.
- Parallel Composition
  - Delay1 and Delay2 run in parallel and communicate synchronously via *temp*.
- Output Hiding
  - The variable *temp* is not exported to the outside of DoubleDelay.

Composition of Components

Three Operations on Composition



- DoubleDelay is textually defined as

$$(\mathsf{Delay}[out \mapsto temp] \;||\; \mathsf{Delay}[in \mapsto temp]) \setminus temp$$

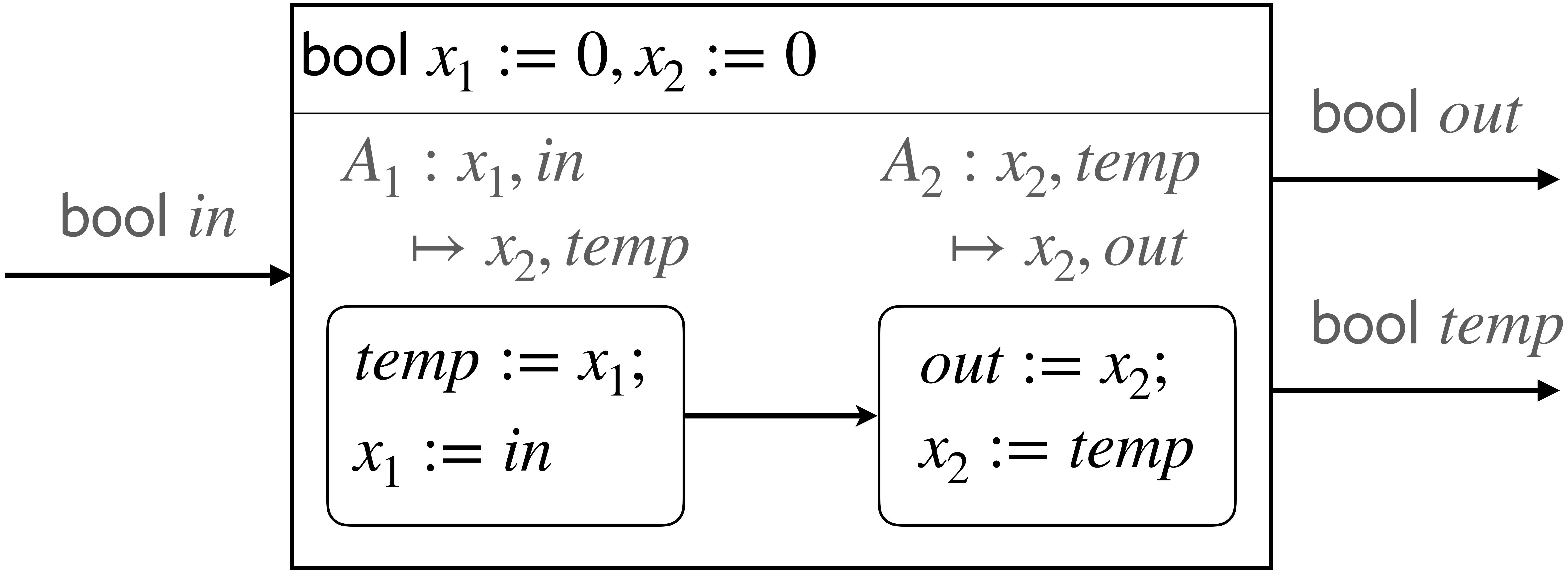
renaming

parallel composition

hiding

Parallel Composition

Delay1$[out \mapsto temp]$ || Delay2$[in \mapsto temp]$



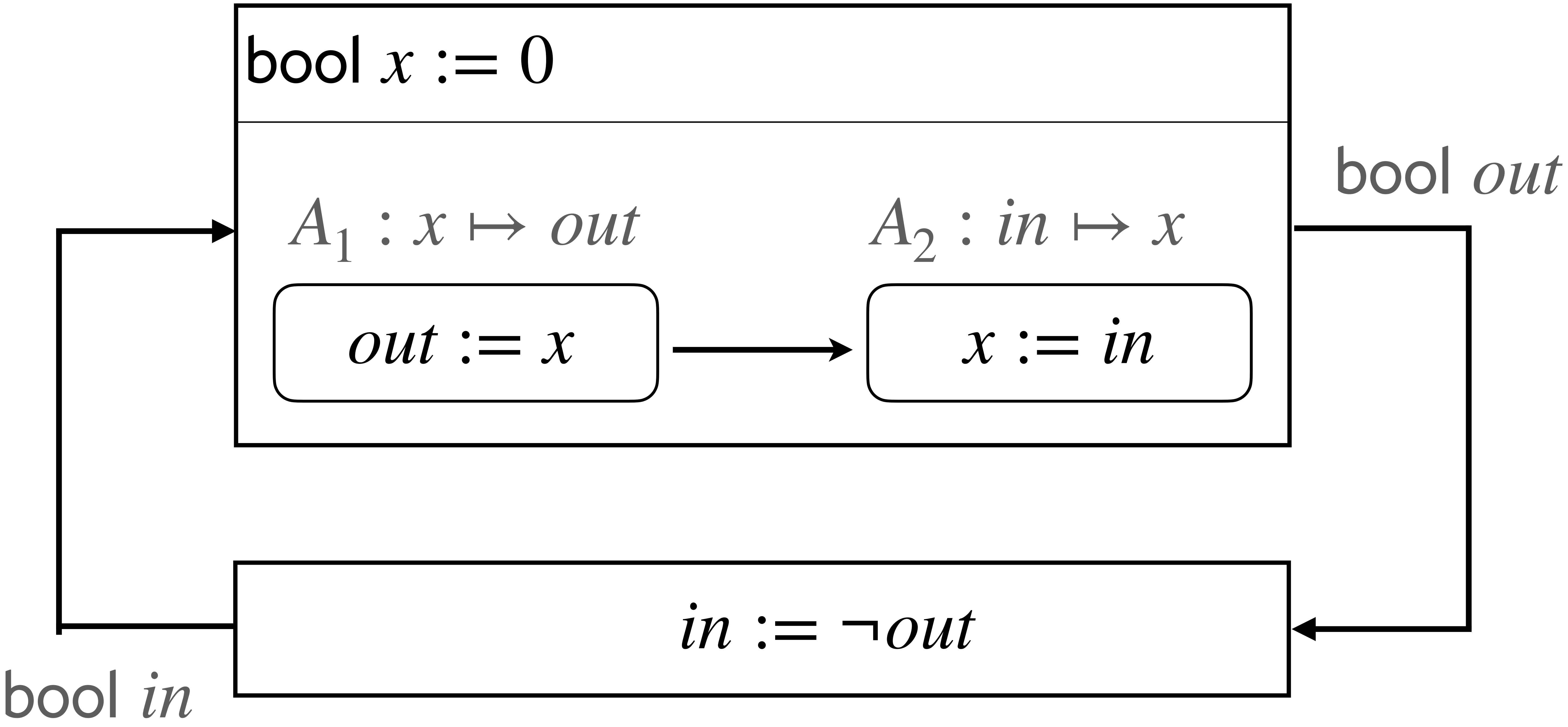
Reactions:

$$(0,0) \xrightarrow{0/(0,0)} (0,0);\ (0,0) \xrightarrow{1/(0,0)} (1,0);\ (0,1) \xrightarrow{0/(0,1)} (0,0);\ (0,1) \xrightarrow{1/(0,1)} (1,0);$$

$$(1,0) \xrightarrow{0/(1,0)} (0,1);\ (1,0) \xrightarrow{1/(1,0)} (1,1);\ (1,1) \xrightarrow{0/(1,1)} (0,1);\ (1,1) \xrightarrow{1/(1,1)} (1,1)\ .$$

Feedback Composition

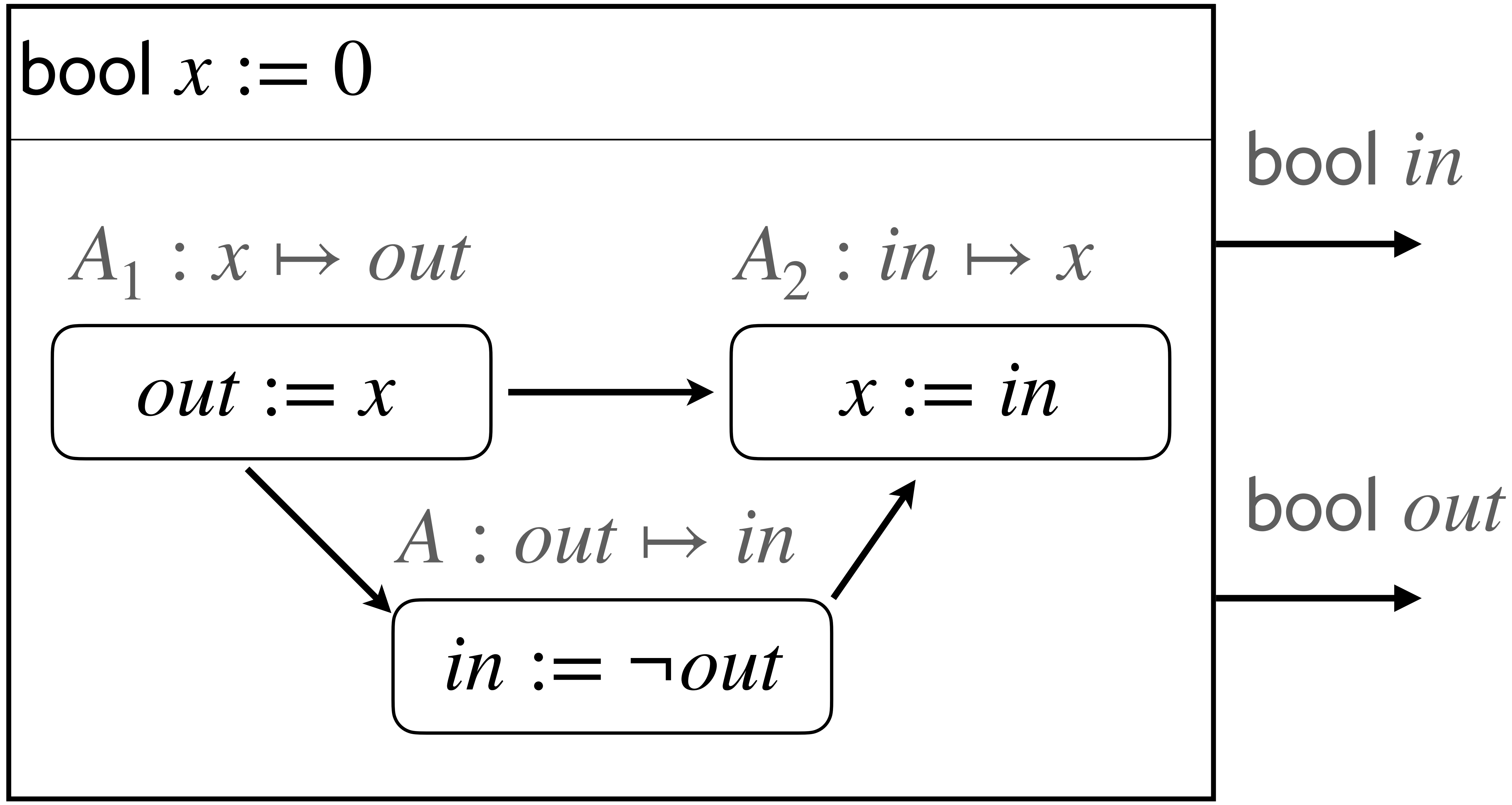
SplitDelay and Inverter : Block Diagram



$$0 \xrightarrow{/(1,0)} 1 \xrightarrow{/(0,1)} 0 \xrightarrow{/(1,0)} 1 \xrightarrow{/(0,1)} 0 \xrightarrow{/(1,0)} \cdots$$

Feedback Composition

SplitDelay and Inverter : Parallel Composition

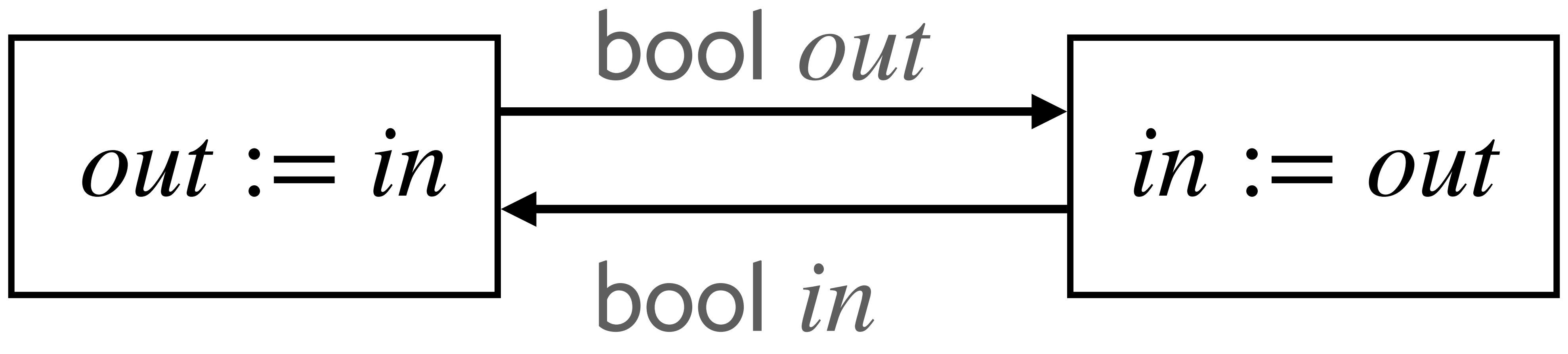


Component Compatibility

- C_1, C_2 : synchronous reactive components
  - $C_1 = (I_1, O_1, S_1, [[Init_1]], [[React_1]])$ w/await-dependency $\succ_1 \subseteq O_1 \times I_1$
  - $C_2 = (I_2, O_2, S_2, [[Init_2]], [[React_2]])$ w/await-dependency $\succ_2 \subseteq O_2 \times I_2$

- C_1 and C_2 are compatible if
  1. the set O_1 and O_2 are disjoint, and
  2. the relation $\succ_1 \cup \succ_2$ is acyclic.

Ex. Two Relays : incompatible



Summary

- Synchronous Model (2)
  - Definition of Synchronous Reactive Components
  - Synchrony Hypothesis
  - Extended-State Machines
  - Finite-State Components / Mealy Machine Representation
  - Combinatorial Components, Event-Triggered Components
  - Deterministic / Nondeterministic Components, Input-Enabled Components
  - Task-Graph / Await Dependency / Schedule of Tasks
  - Composition of Components / Compatibility