TDDI11: Embedded Software

State Machines

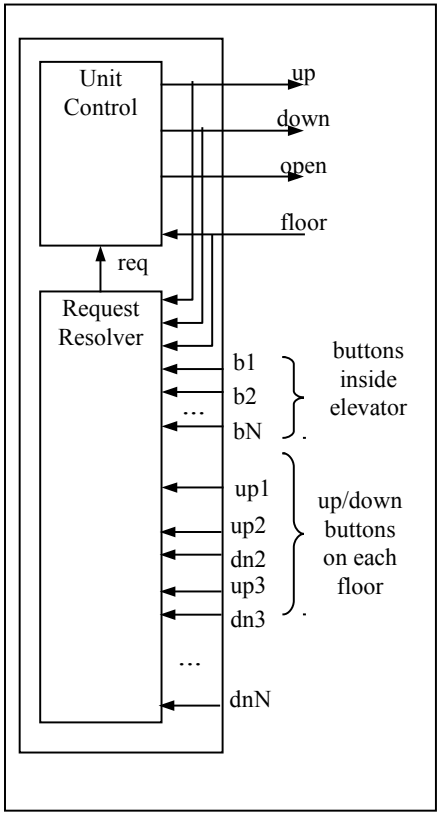
Introductory example: An elevator controller

- Simple elevator controller
  - *Request Resolver* resolves various floor requests into single requested floor
  - *Unit Control* moves elevator to this requested floor

Partial English description

"Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don't change directions unless there are no higher requests when moving up or no lower requests when moving down…"

System interface



Elevator controller using a sequential program model

Sequential program model

Inputs: int floor; bit b1..bN; up1..upN-1; dn2..dnN;
Outputs: bit up, down, open;
Global variables: int req;

```
void UnitControl()
{
  up = down = 0; open = 1;
  while (1) {
    while (req == floor);
    open = 0;
    if (req > floor) { up = 1;}
    else {down = 1;}
    while (req != floor);
    up = down = 0;
    open = 1;
    delay(10);
  }
}
```

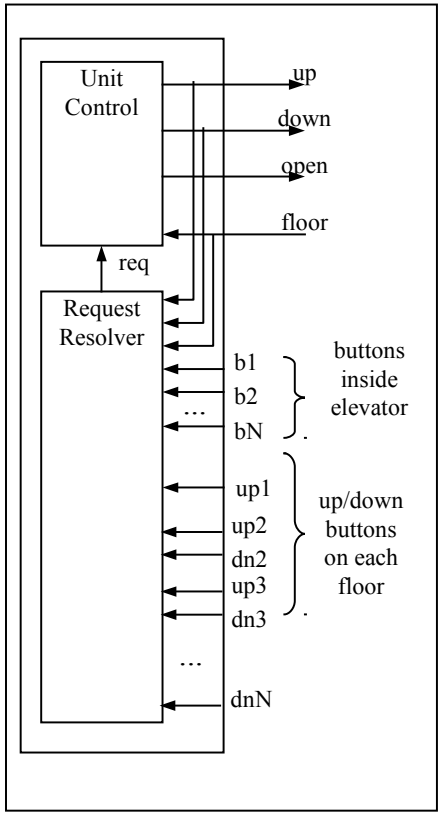
```
void RequestResolver()
{
  while (1)
  ...
    req = ...
  ...
}

void main()
{
  Call concurrently:
    UnitControl() and
    RequestResolver()
}
```

Partial English description

"Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don't change directions unless there are no higher requests when moving up or no lower requests when moving down…"

System interface



Finite-state machine (FSM) model

- Trying to capture this behavior as sequential program is a bit awkward

- Instead, we might consider an FSM model, describing the system as:
  - Possible states
    - E.g., *Idle*, *GoingUp*, *GoingDn*, *DoorOpen*
  - Possible transitions from one state to another based on input
    - E.g., req > floor
  - Actions that occur in each state
    - E.g., In the *GoingUp* state, u,d,o,t = 1,0,0,0 (up = 1, down, open, and timer_start = 0)

State machine

- The system is described as a set of states
- Each state is a representation of what the system looks like now and how it got there
- Each state reacts in a specific way to *every possible* input event, leading to a new state via a transition

Mealy and Moore

- There are two kinds of state machines
  - Mealy and Moore
  - Equally powerful

- Note: Finite state machines or finite automata are two names of the same thing

Moore machines

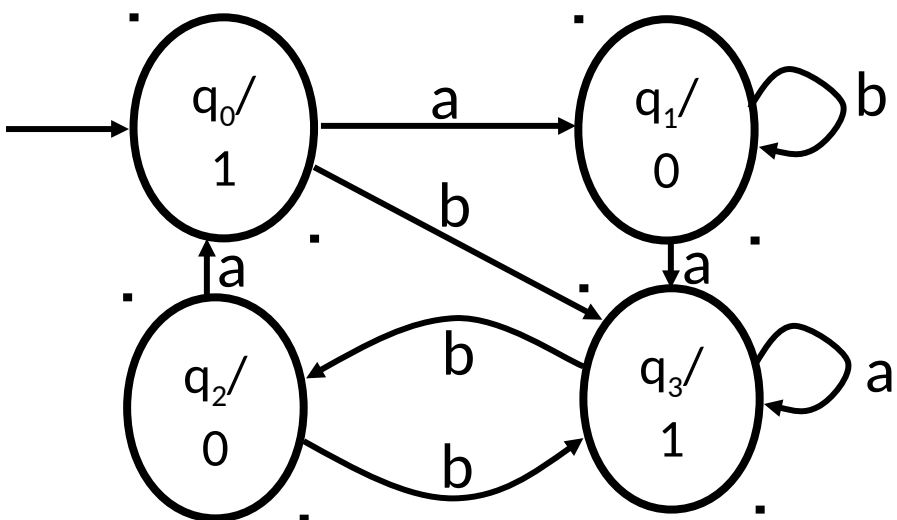
- A Moore machine is a collection of 5 things:
  1. a finite set of states $q_0, q_1, q_2, \ldots$, where q_0 is designated the start state
  2. an alphabet Σ of input letters
  3. an alphabet Γ of output characters
  4. a transition table that shows for each state and each input letter what state to go to next.
  5. an output table that shows what character is output (or printed) when entering a state.

(state, letter from Σ) $\xrightarrow{\text{transition}}$ state

Example of a Moore machine

Example: states = $\{q_0,q_1,q_2,q_3\}$ $\Sigma=\{a,b\}$ $\Gamma=\{0,1\}$

Old state	Output by the old state	New state	
		After input a	After input b
$-q_0$	1	q_1	q_3
q_1	0	q_3	q_1
q_2	0	q_0	q_3
q_3	1	q_3	q_2

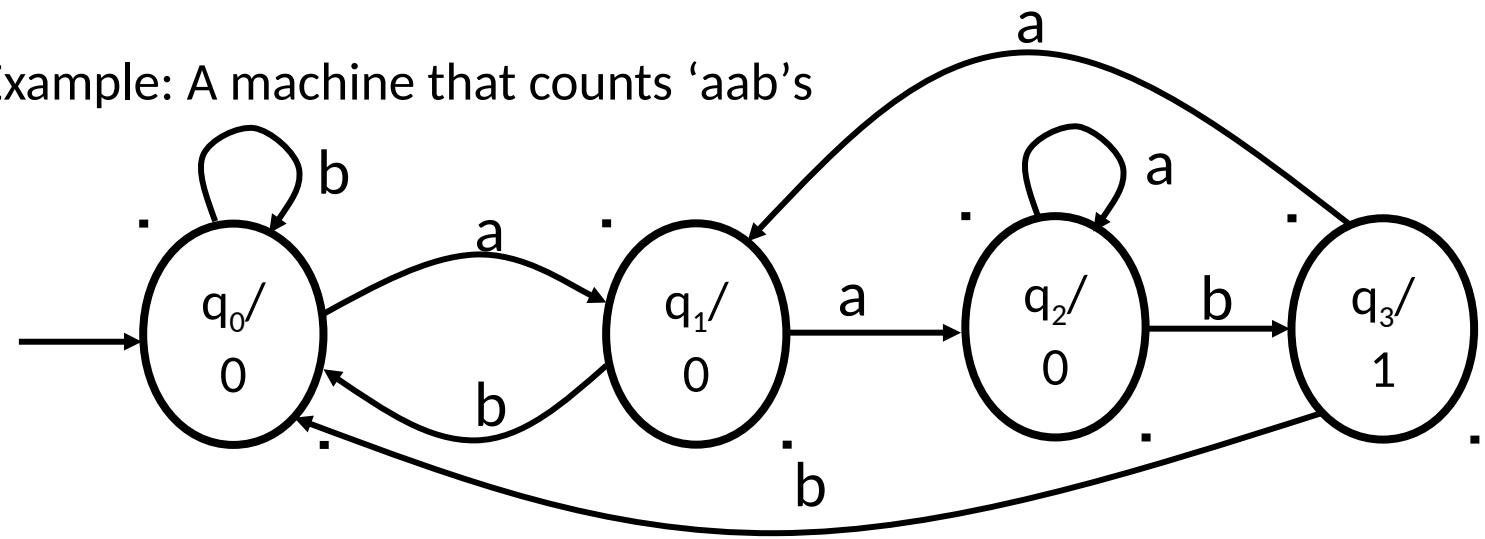


abab

10010

Another Moore machine

Example: A machine that counts ‘aab’s



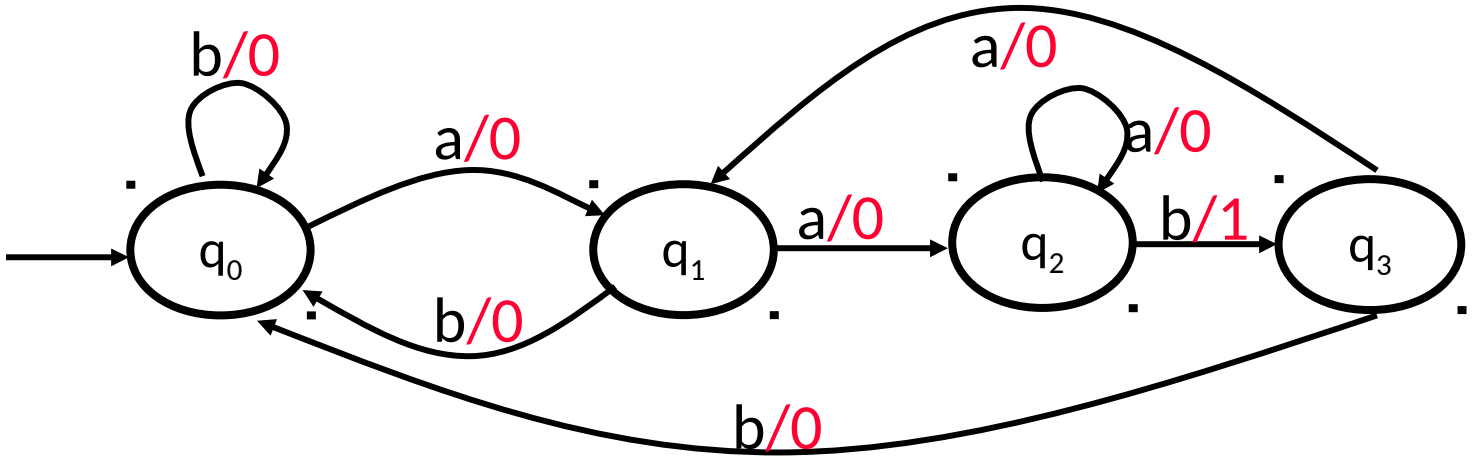
Input		a	a	a	b	a	b	b	a	a	b	b
State	q_0	q_1	q_2	q_2	q_3	q_1	q_0	q_0	q_1	q_2	q_3	q_0
Output	0	0	0	0	1	0	0	0	0	0	1	0

Printing 1 ≈ found a word that end in aab

Mealy machine

- A Mealy machine is:
  1. a finite set of states $q_0,q_1,q_2,\ldots$, where q_0 is designated the start state
  2. an alphabet Σ of input letters
  3. an alphabet Γ of output characters
  4. a finite set of transitions that indicate, for each state and letter of the input alphabet, the state to go to next and the character that is output.

Mealy machine example

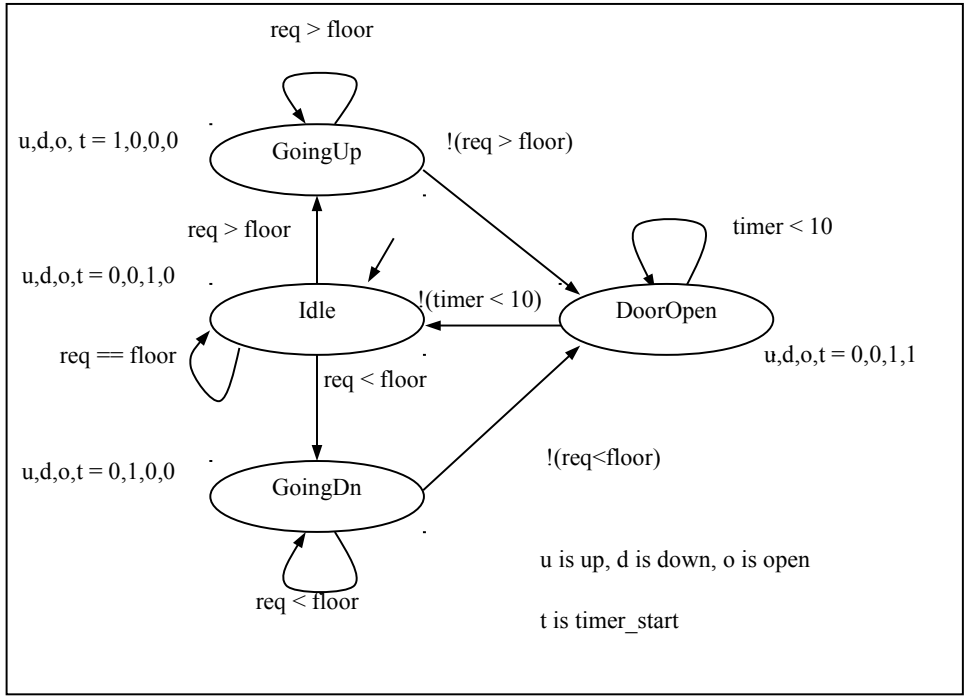


Mealy vs. Moore

- Mealy and Moore machines are equivalent
  - If Mealy machine capture a model, so does a Moore machine
- Moore models usually means:
  - more states
  - simpler to understand

Describing a system as a state machine

1. List all possible states
2. Declare all variables
3. For each state, list possible transitions, with conditions, to other states
4. For each state and/or transition, list associated actions
5. For each state, ensure exclusive and complete exiting transition conditions
   - No two exiting conditions can be true at same time
     - Otherwise nondeterministic state machine
   - One condition must be true at any given time
     - Reducing explicit transitions should be avoided when first learning



States

- States are circles when drawn graphically
  - A descriptive name is written within
  - An action, what to do as long as staying in this state (which outputs to set, or code to execute) is written next to the circle
  - The action should set every output to a known value
  - For each possible input event a transition arrow leads out from the state in to some other state

Transitions

- Transitions are arrows out from one state in to another
  - A condition is associated with each arrow
  - When the condition is true the transition is performed, the machine moves to the state indicated by the transition
  - If none of the transitions from a state is true, the default transition should be to stay in that state

State machine vs. sequential program model

- Different thought process used with each model
- State machine:
  - Encourages designer to think of all possible states and transitions among states based on all possible input conditions
- Sequential program model:
  - Designed to transform data through series of instructions that may be iterated and conditionally executed
- State machine description excels in many cases
  - More natural means of computing in those cases
  - *Not* due to graphical representation (state diagram)

Capturing state machines in sequential programming language

- Despite benefits of state machine model, most popular development tools use sequential programming language
  - C, C++, Java, Ada, VHDL, Verilog, etc.
  - Development tools are complex and expensive, therefore not easy to adapt or replace

- Two approaches to capturing state machine model with sequential programming language
  - Front-end tool approach
  - Language subset approach

Language subset approach

```
#define IDLE 0
#define GOINGUP 1
#define GOINGDN 2
#define DOOROPEN 3
void UnitControl() {
   int state = IDLE;
   while (1) {
      switch (state) {
         IDLE: up=0; down=0; open=1; timer_start=0;
            if   (req==floor) {state = IDLE;}
            if   (req > floor) {state = GOINGUP;}
            if   (req < floor) {state = GOINGDN;}
            break;
         GOINGUP: up=1; down=0; open=0; timer_start=0;
            if   (req > floor) {state = GOINGUP;}
            if   (!(req>floor)) {state = DOOROPEN;}
            break;
         GOINGDN: up=1; down=0; open=0; timer_start=0;
            if   (req < floor) {state = GOINGDN;}
            if   (!(req<floor)) {state = DOOROPEN;}
            break;
         DOOROPEN: up=0; down=0; open=1; timer_start=1;
            if (timer < 10) {state = DOOROPEN;}
            if (!(timer<10)){state = IDLE;}
            break;
      }
   }
}
```

UnitControl **state machine in sequential programming language**

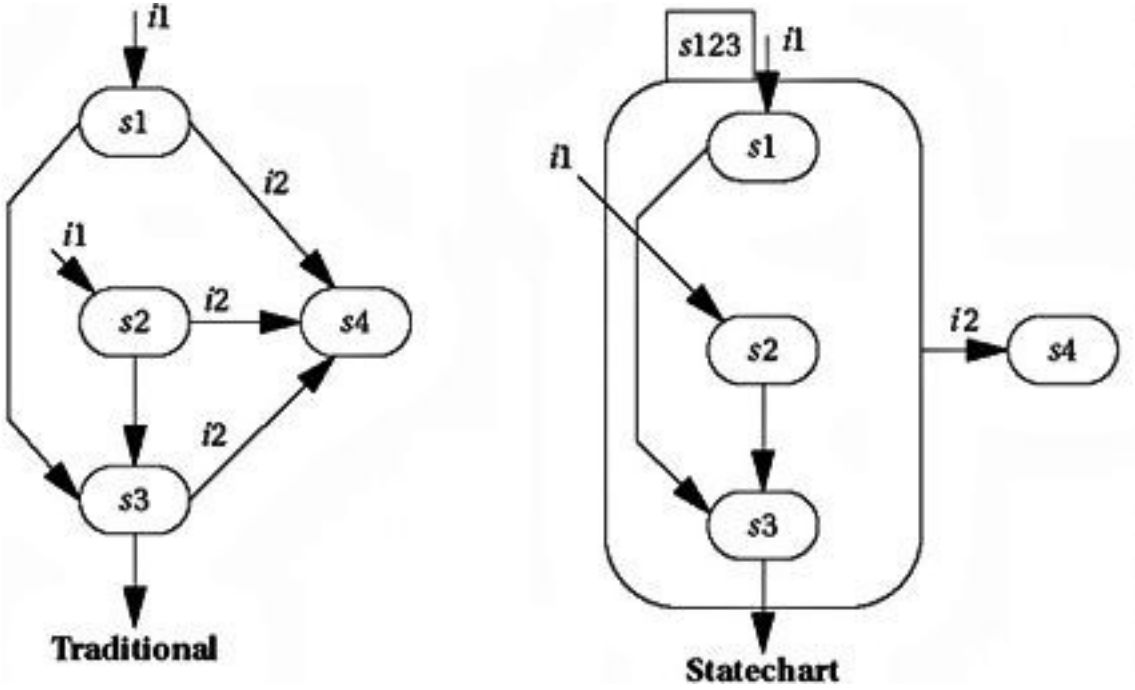
Statecharts

- Another well-known technique for state-based specification
  - Helps to eliminate clutter and clarify the structure
- Allows states to be grouped together

Statechart groups

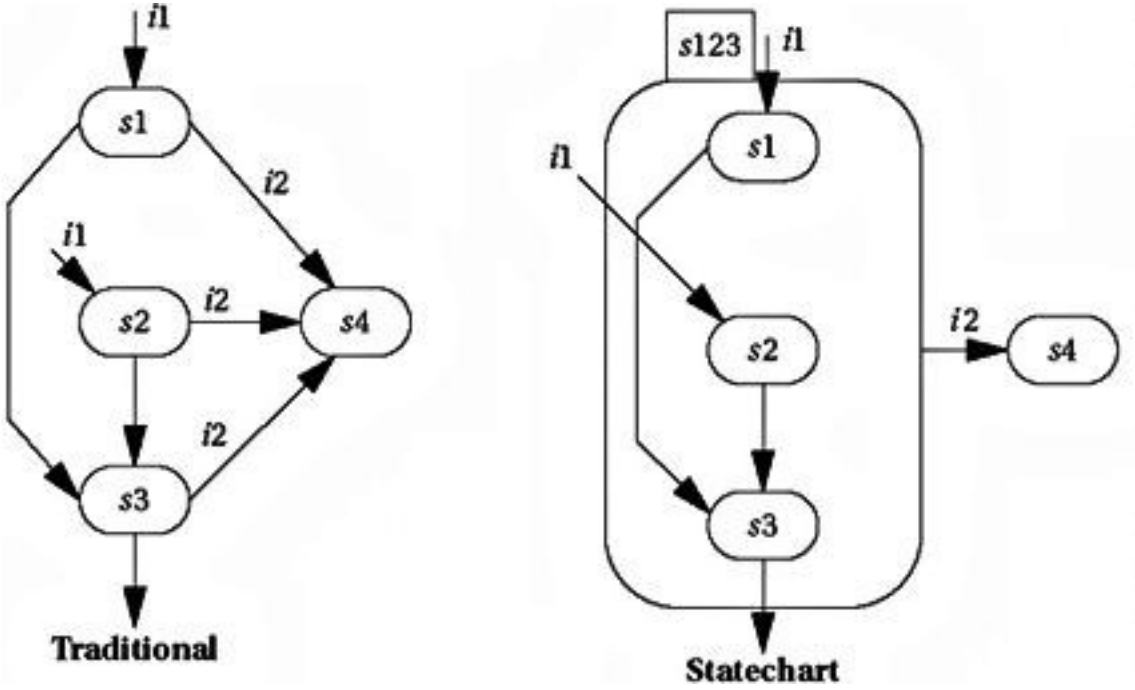
- Two groups are
  - OR
  - AND

Example of OR



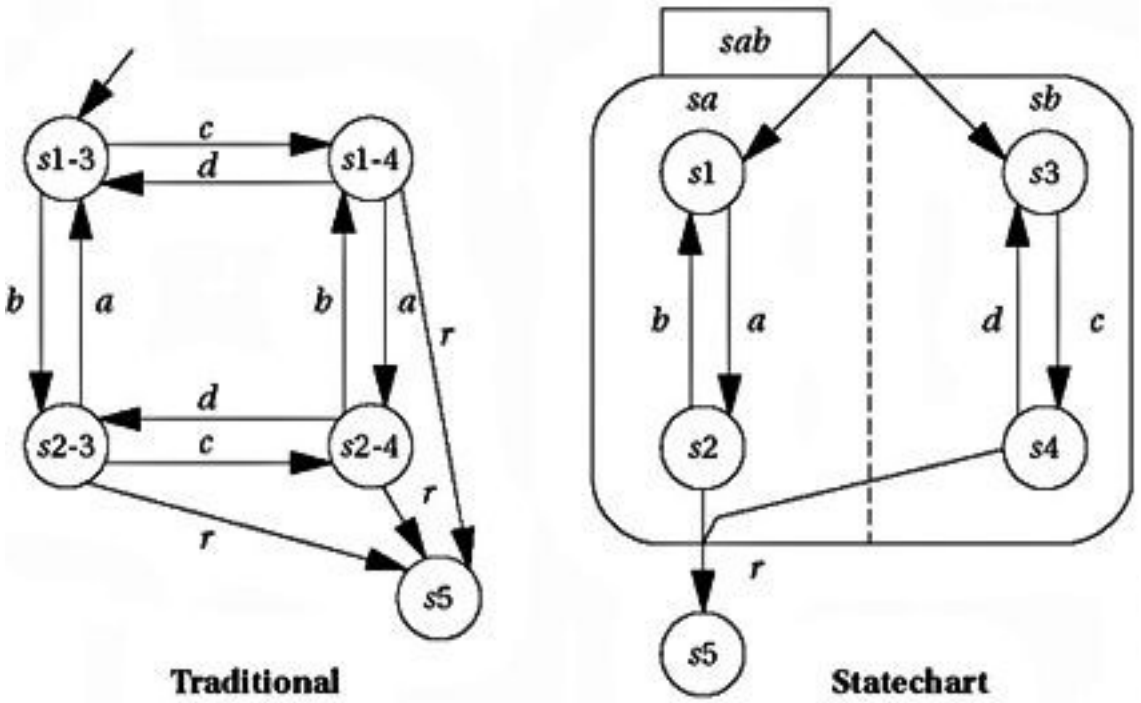
- The machine goes to s4 from s1, s2, s3
- Statechart captures this by having one state around s1, s2, and s3
- The name of the OR state is written on top of the state

Example of OR



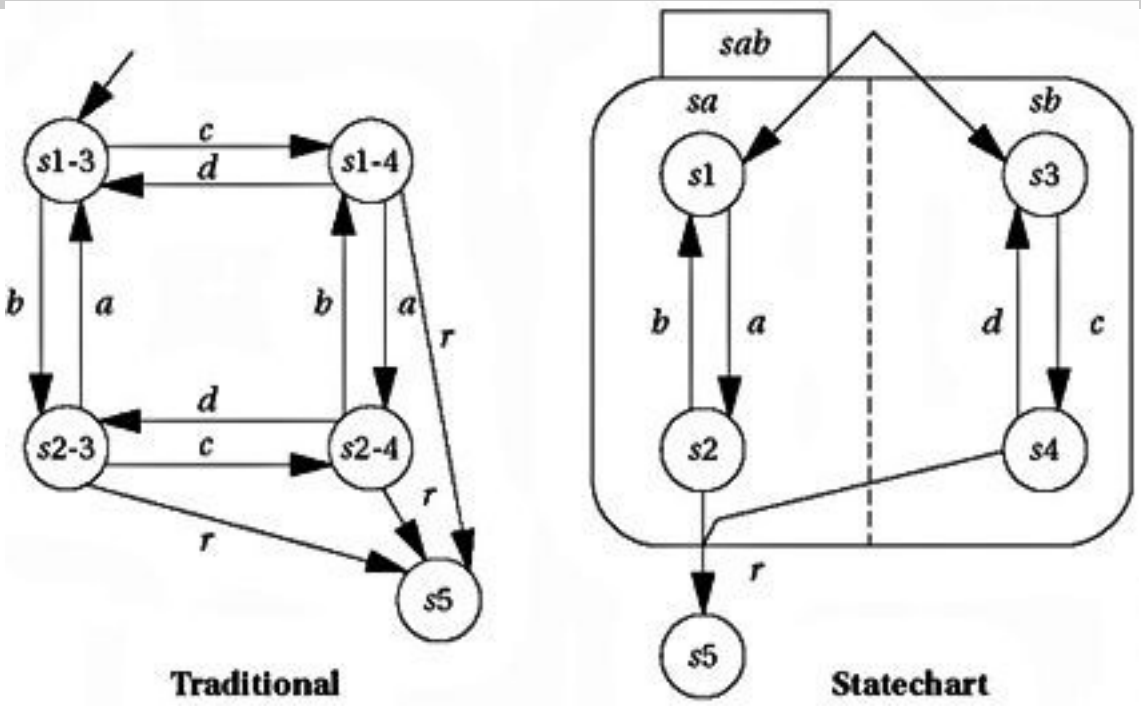
- A single transition out of s123 means that on event i2, all states go to s4.
- The OR state allows transitions between its own internal states

Example of AND



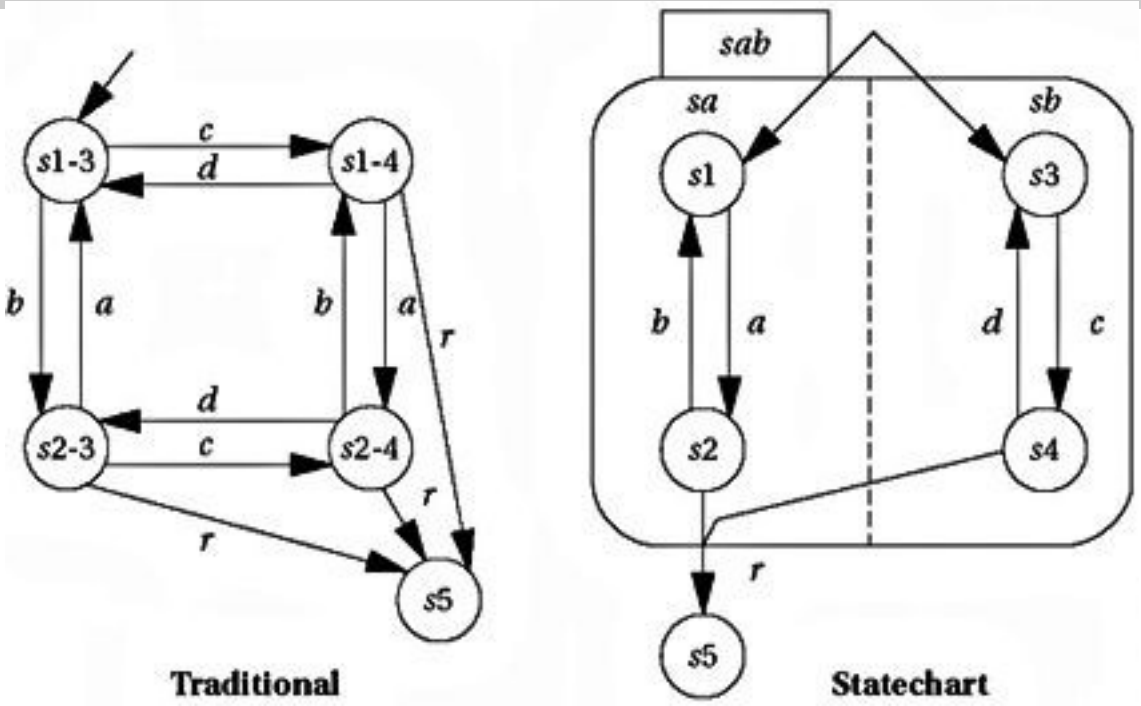
- The traditional model has many transitions
  - Between states
  - Going out of all states
  - One initial transition

Example of AND



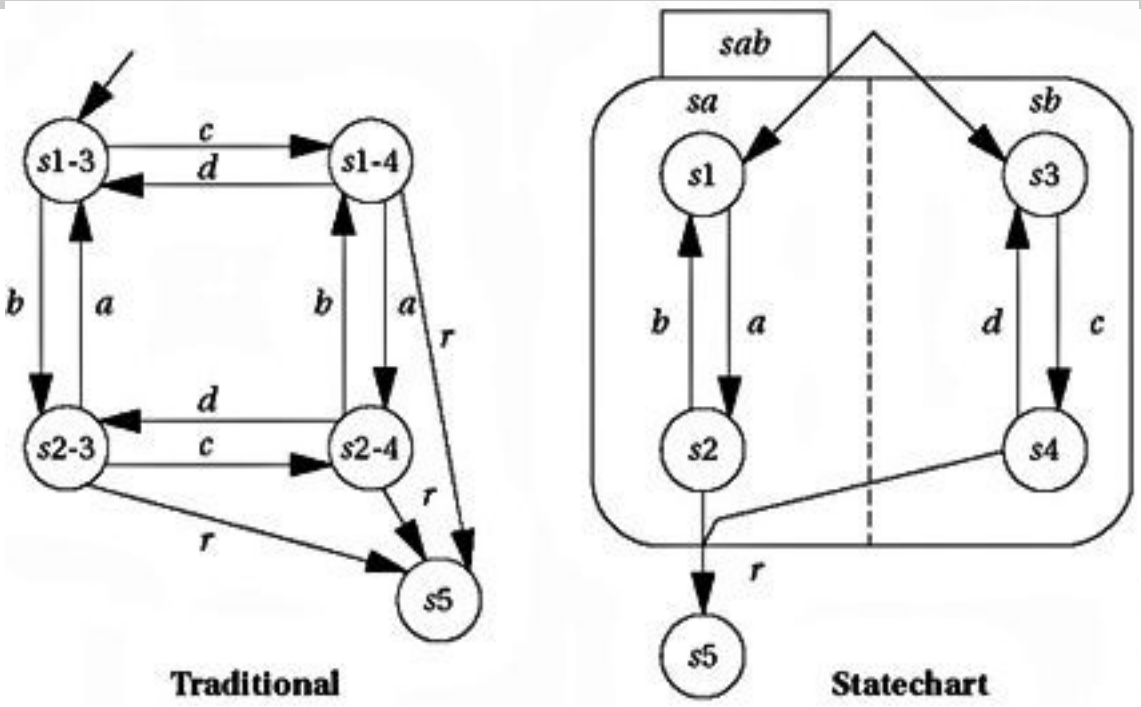
- The statechart has a AND state called sab
- sab has two components sa and sb
- When the machine enters the state sab, it is simultaneously in both sa and sb
  - We must know both sa and sb to know sab

Example of AND



- The name of states reveal their relations
- s1-3 corresponds to sab in s1 and in s3
- When the machine enters the state sab, it is simultaneously in both sa and sb

Example of AND

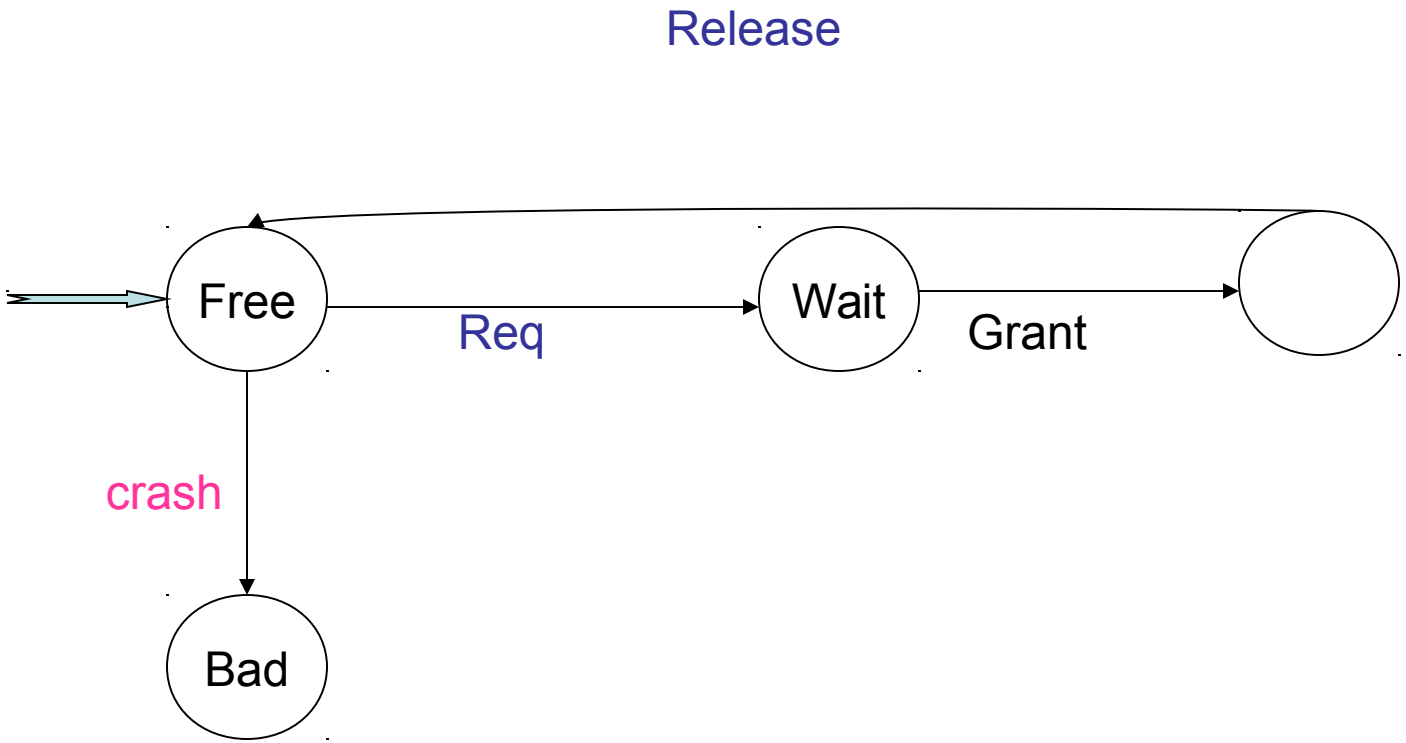


- Both models describe the same behavior but the statechart is much simpler, cleaner and easier to understand

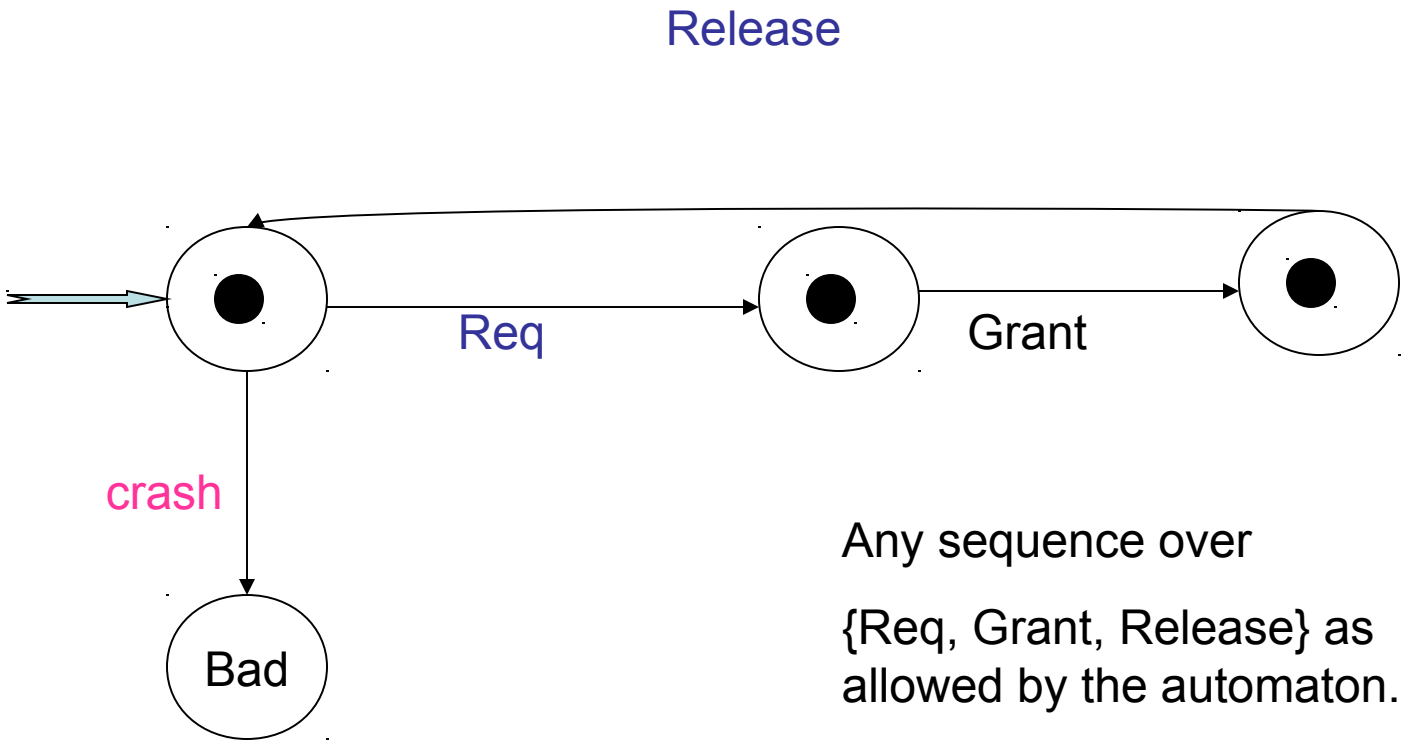
Verification

Example

Example



Example



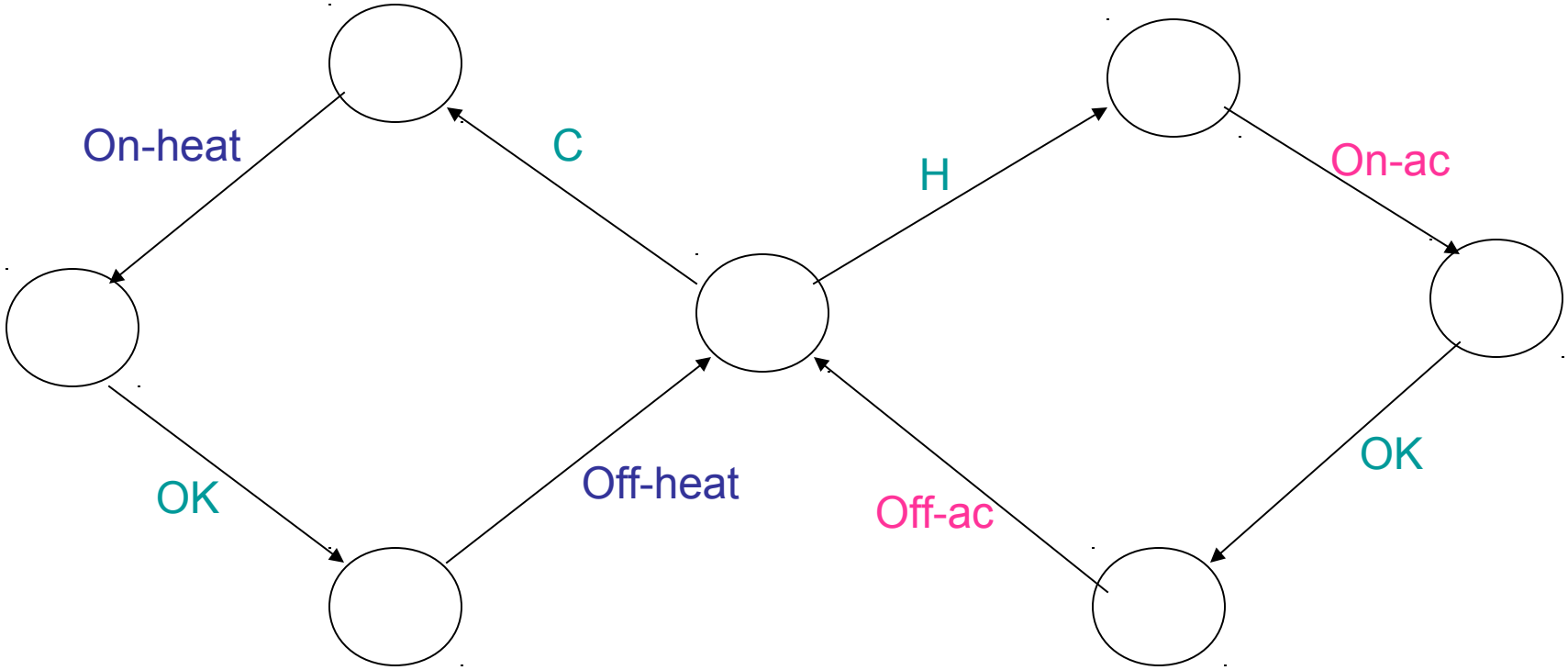
Any sequence over

{Req, Grant, Release} as allowed by the automaton.

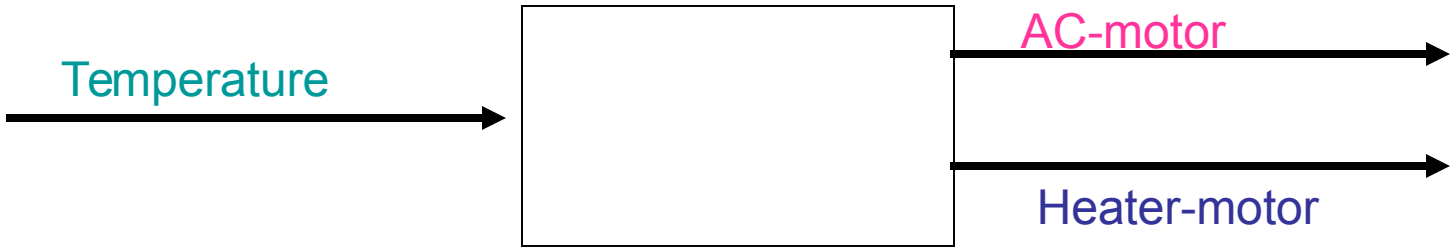
Rq G Rl Rq G allowed

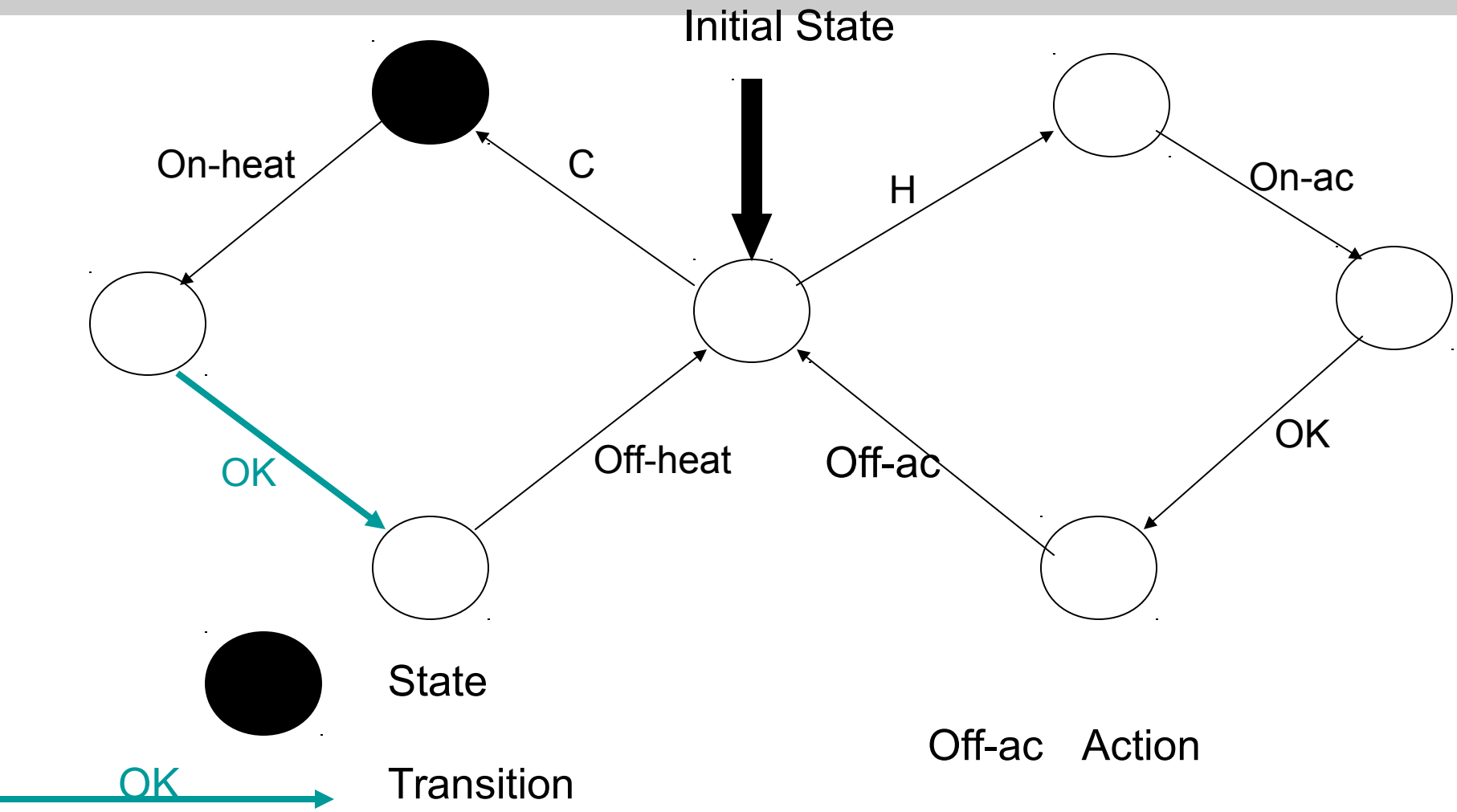
Rq G Rl Cr not wanted!

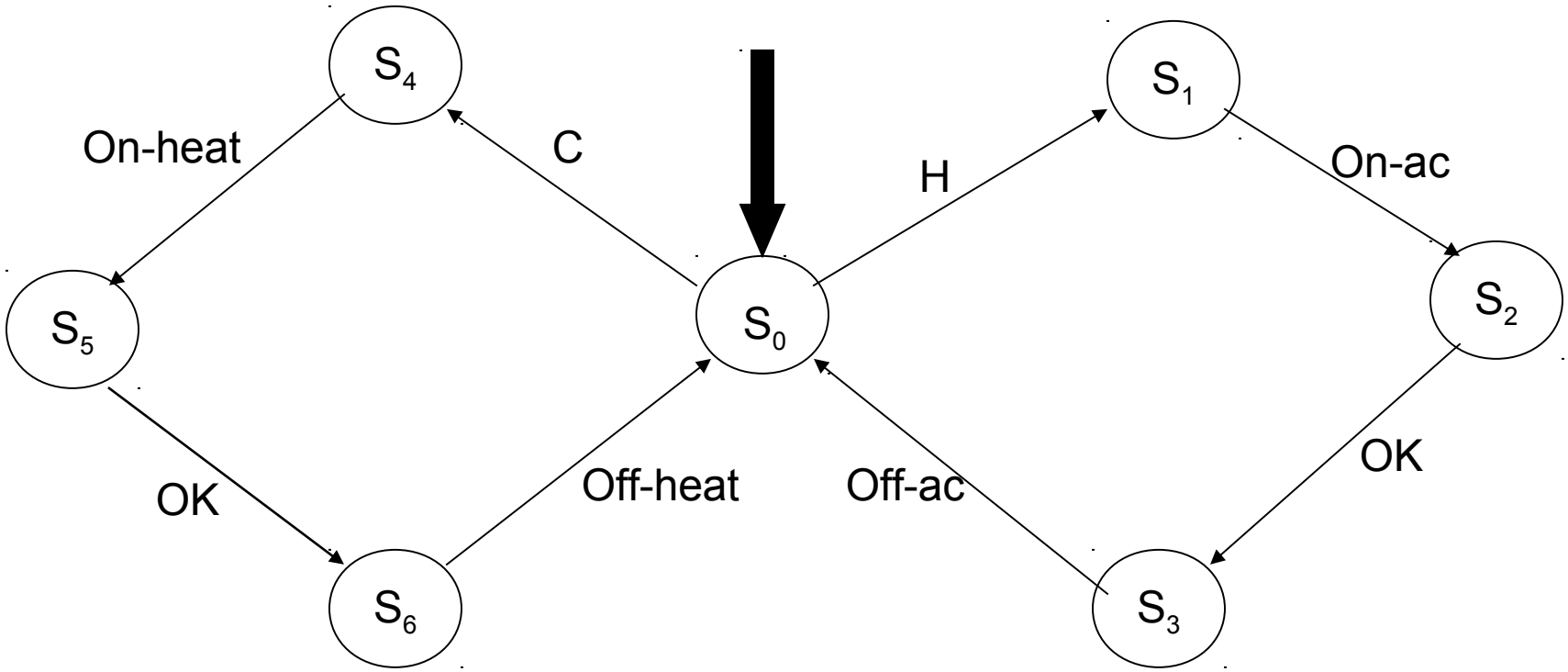
Temperature controller



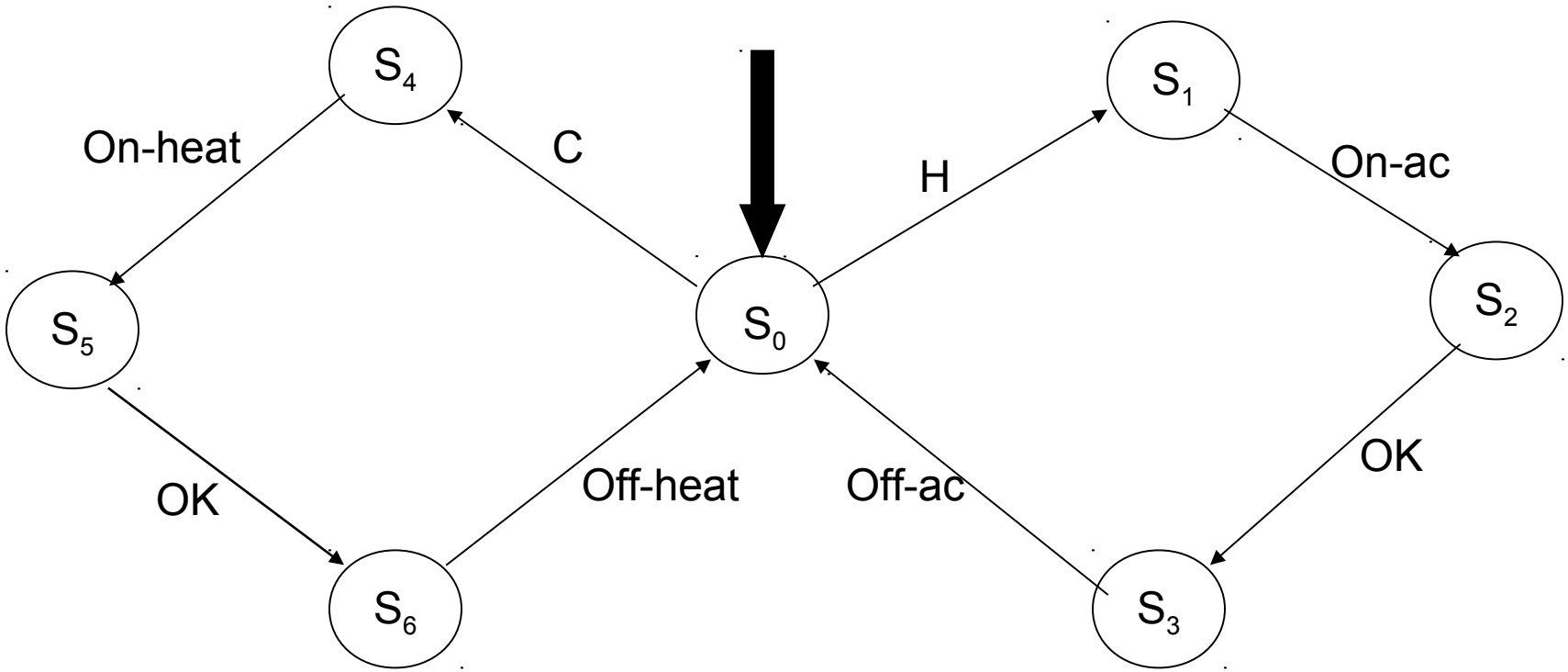
Signal Flow




Initial State
On-heat
C
H
On-ac
OK
Off-heat
Off-ac
OK
State
Transition
Off-ac Action
OK



PATH – S_4 on-heat S_5 OK S_6 off-heat S_0 ? S_1 ….



PATH – S_4 S_5 S_6 S_0 S_1 ….

Run ---- Path starting from an initial state

----- S_0 S_1 S_2 S_3 S_0 S_1 ….

Transition Systems

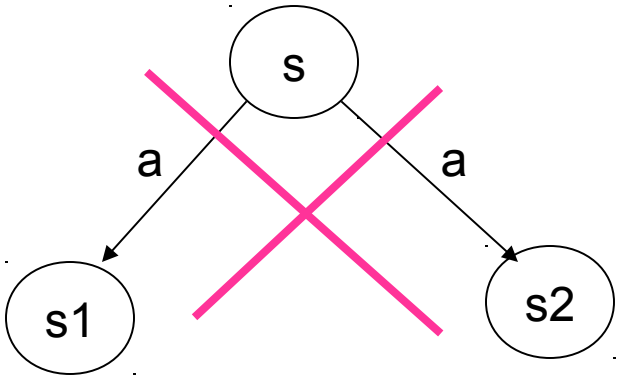
- TS = (S, Act, $\rightarrow$, S_{in}) --- Transition System
  - S --- States
  - Act --- A set of actions
  - $\rightarrow \subseteq S \times Act \times S$ ---- Transition Relation
  - $S_{in} \subseteq S$ ---- Initial states

- Often:
  - S and Act are finite sets.
  - S_{in} has only one element.
  - The transition relation is deterministic.

Deterministic Transition Systems

- TS = (S, Act, $\longrightarrow$, S_{in}) --- Transition System
- (s, a, s') $\in$ $\longrightarrow$
  - s $\xrightarrow{a}$ s'

Deterministic Transition Systems

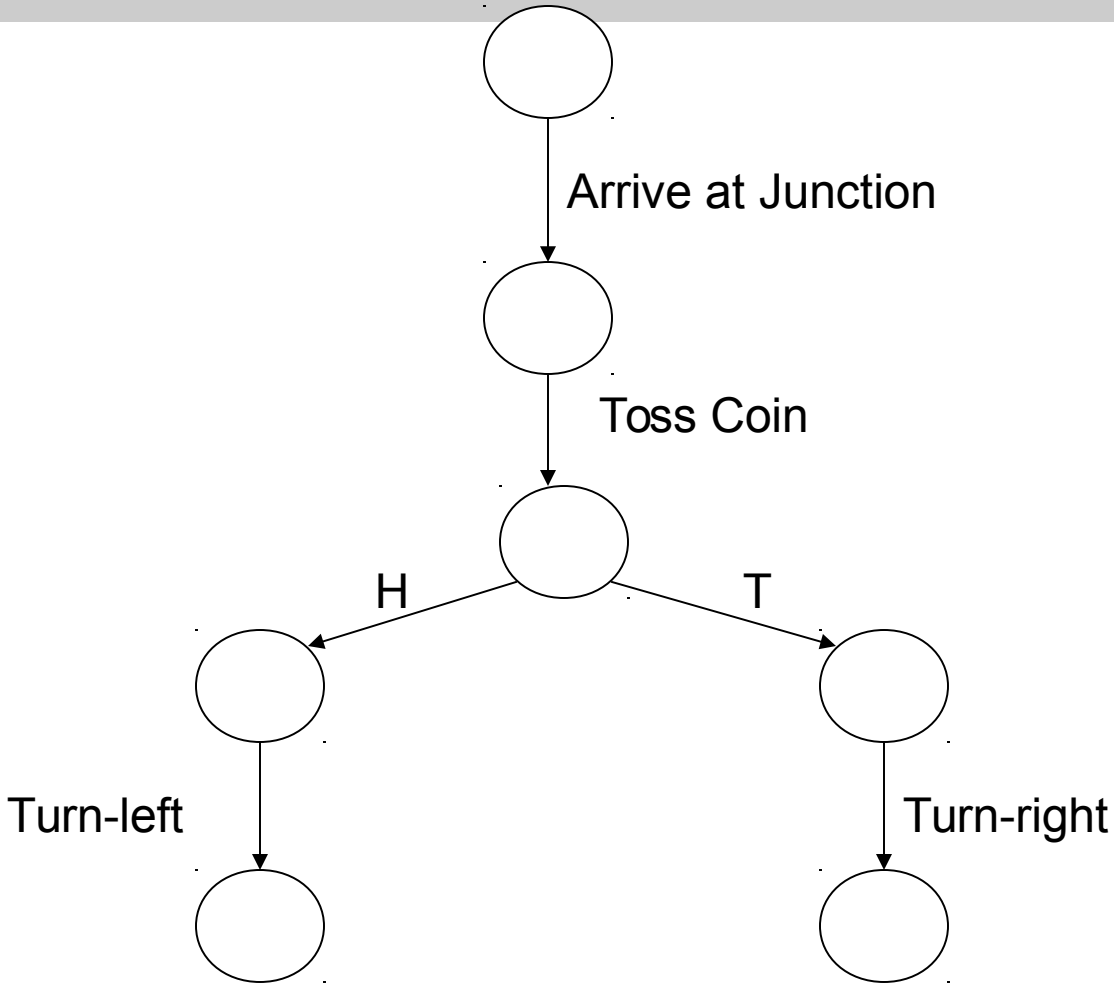


$s \xrightarrow{a} s1$ AND $s \xrightarrow{a} s2$ IMPLIES $s1 = s2$

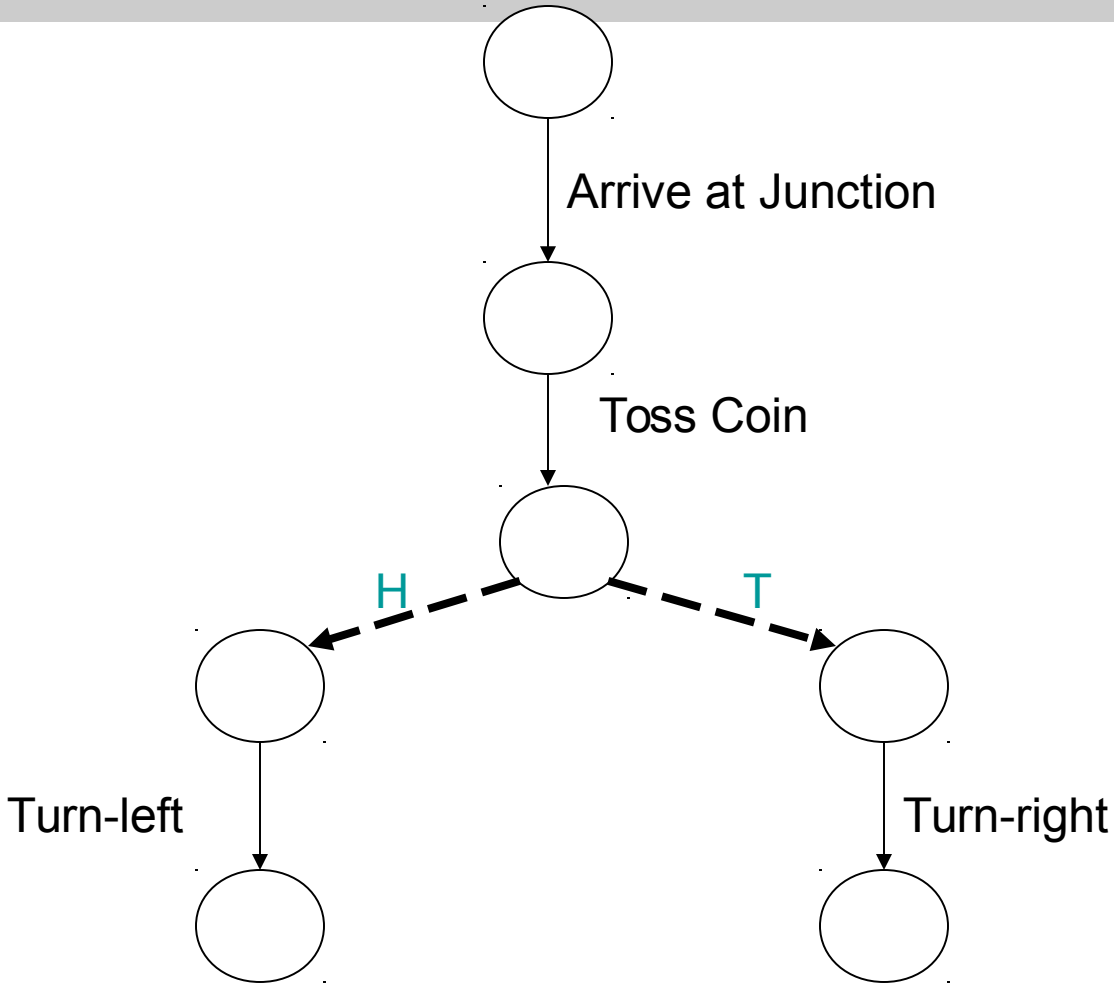
Non-determinism is useful for getting succinct specifications.

Abstractions (hiding details) give rise to non-determinism.

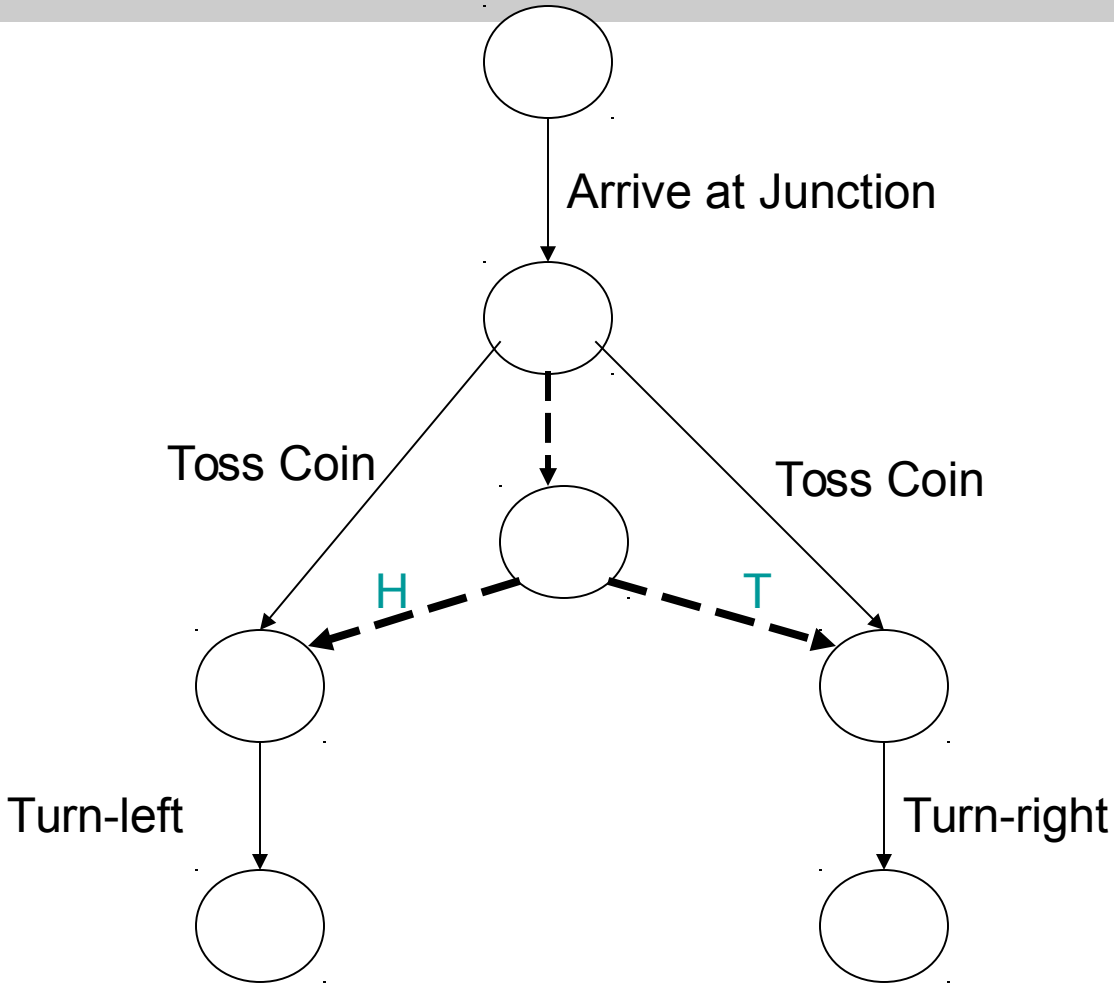
Non-Determinism



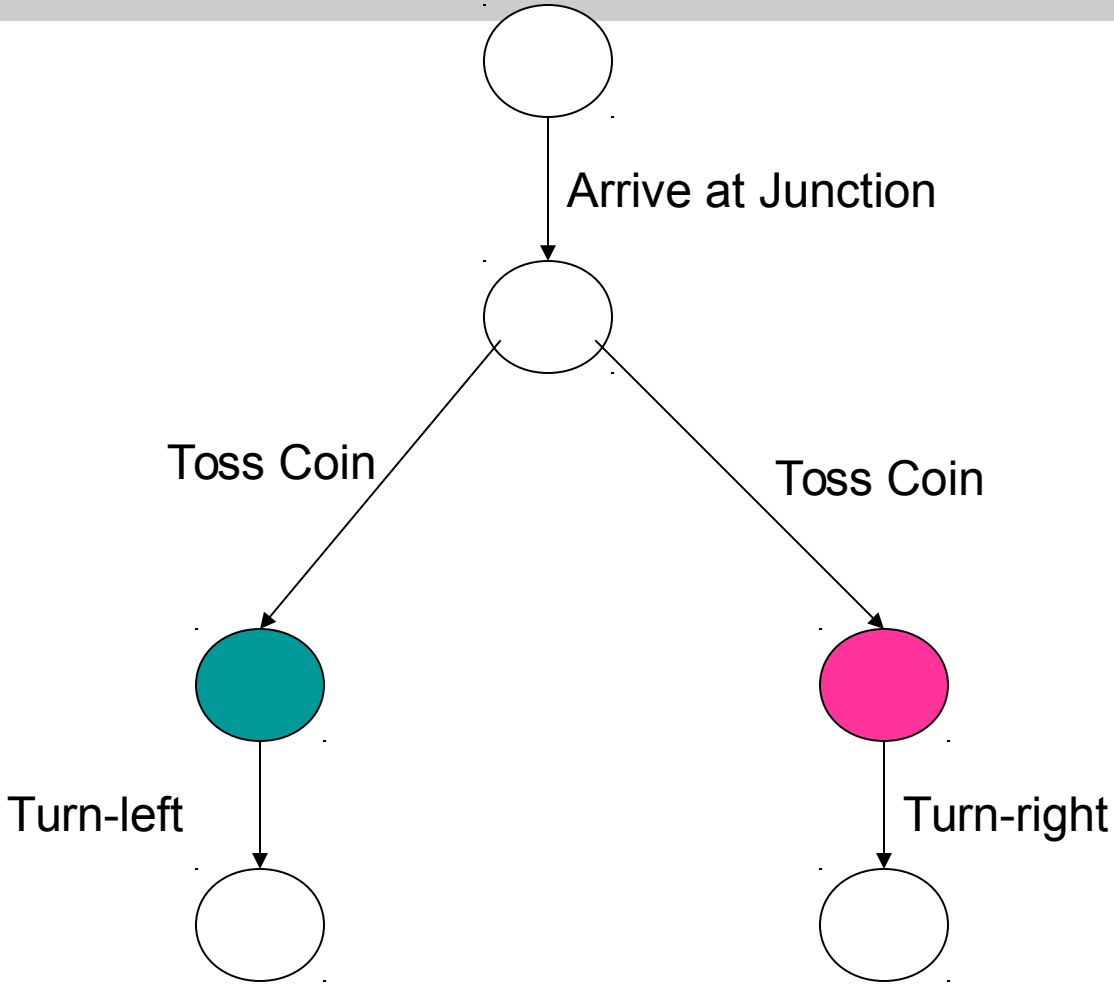
Non-Determinism



Non-Determinism



Non-Determinism

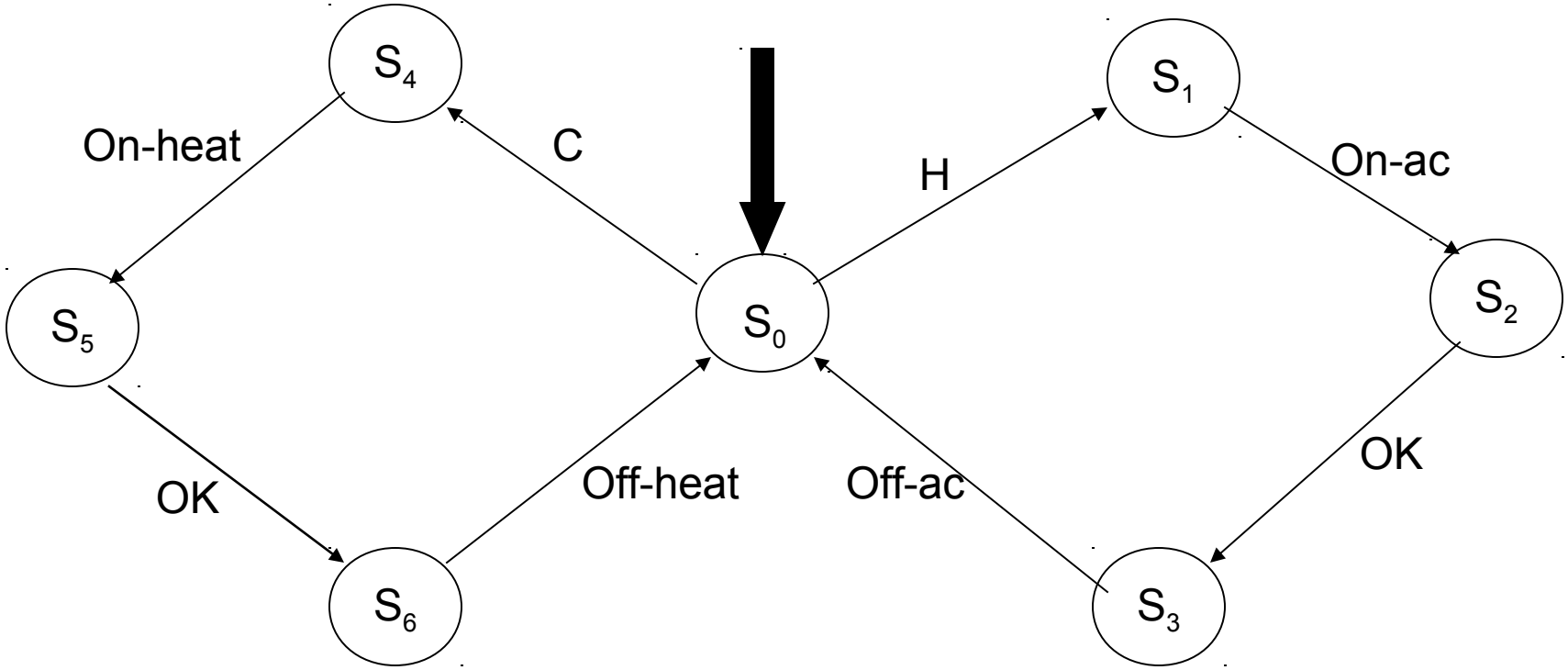


Behaviors

- TS = (S, Act, $\rightarrow$, S_{in})
- The behavior of TS.
  - The ***runs*** of TS.
- Properties:
  - Is there a run leading to ***deadlock***?
    - s_0 ----------> s $s_0 \in S_{in}$
    - No action is enabled at s
  - Is the state s ***reachable***?
  - Is there a dead state which is reachable?
- Often TS is presented implicitly!
  - For example, as a network of smaller transition systems.

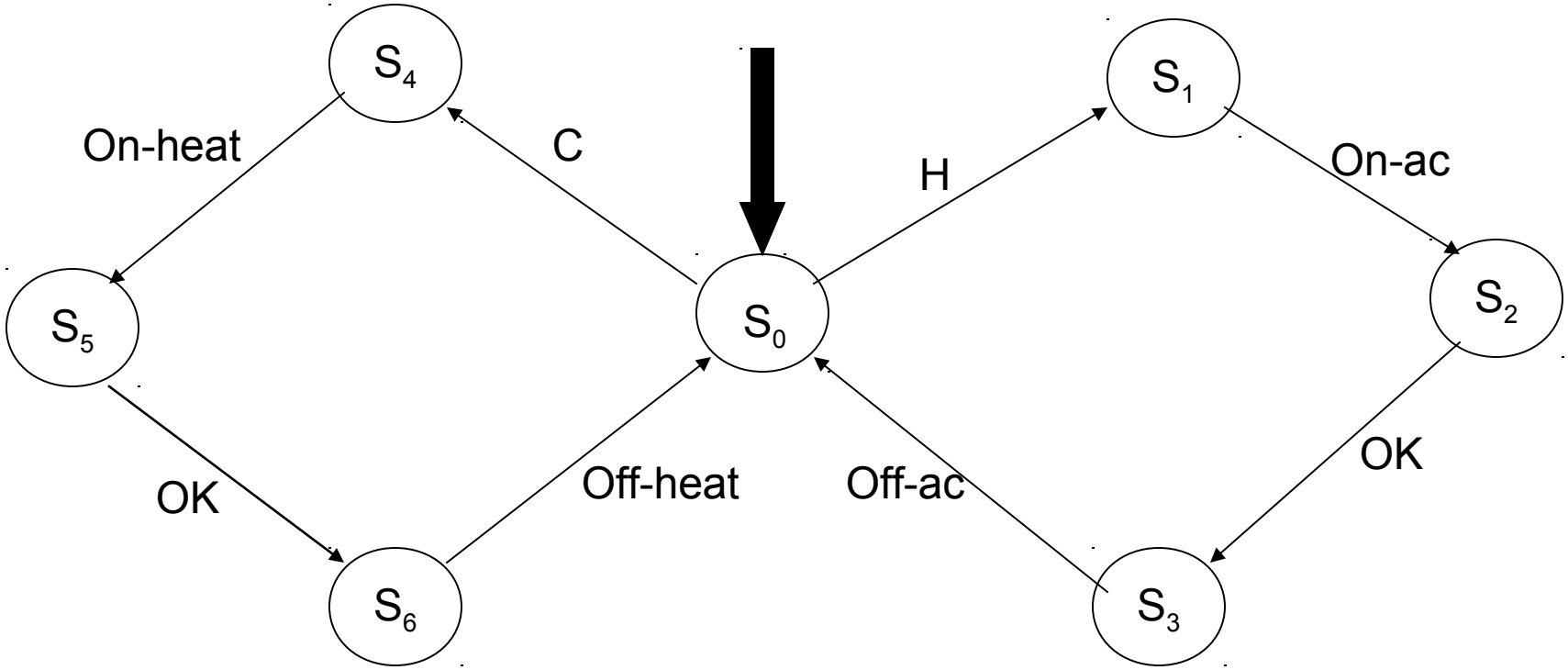
Computations

- TS = (S, Act, $\longrightarrow$, S_{in})
- Behaviors can also be defined as action sequences:
  - Computations, traces,…
- s_0 s_1 s_2 ……. s_n ---- run.
- s_0 a_1 s_1 a_2 s_2 ….s_{n-1} a_n s_n
- $s_i \xrightarrow[a_i]{} s_{i+1}$
- a_1 a_2 a_3 ….a_n is a computation.



Run ----- S_0 S_1 S_2 S_3

Computation ----- ?



Run ----- S_0 S_1 S_2 S_3 S_0

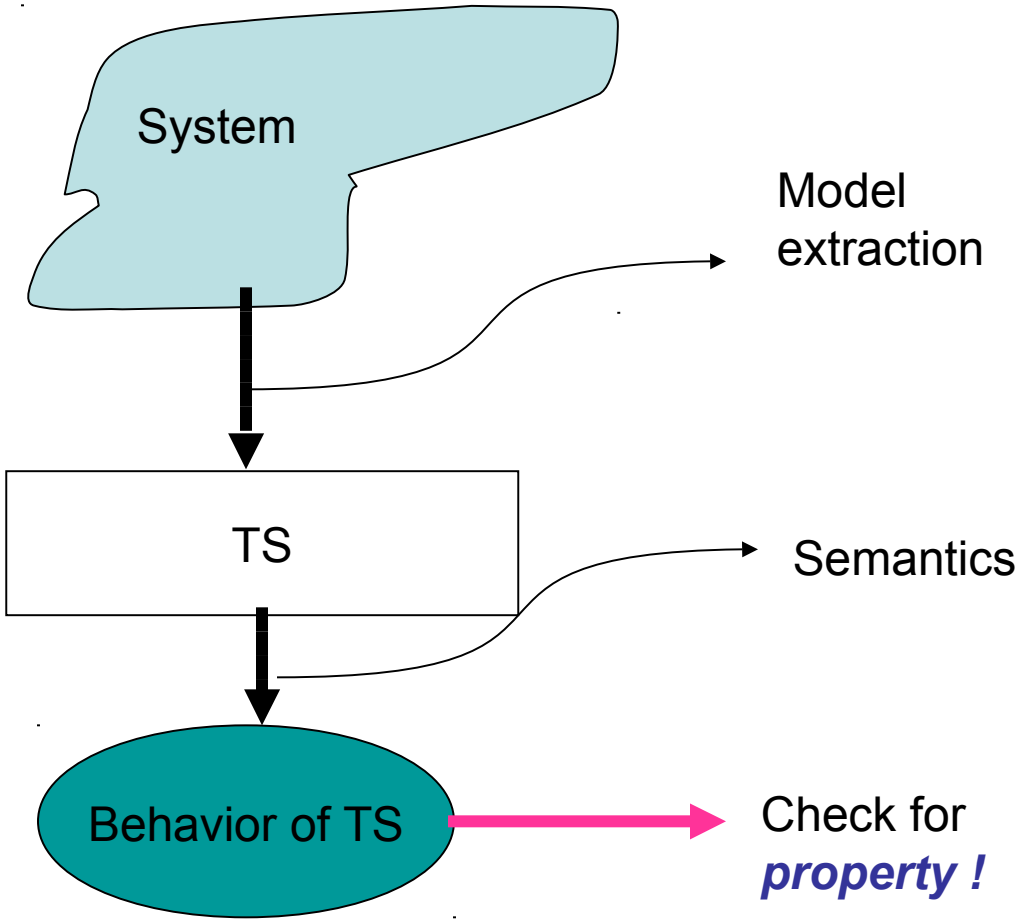
Computation ----- H On-ac OK off-ac

Behaviors

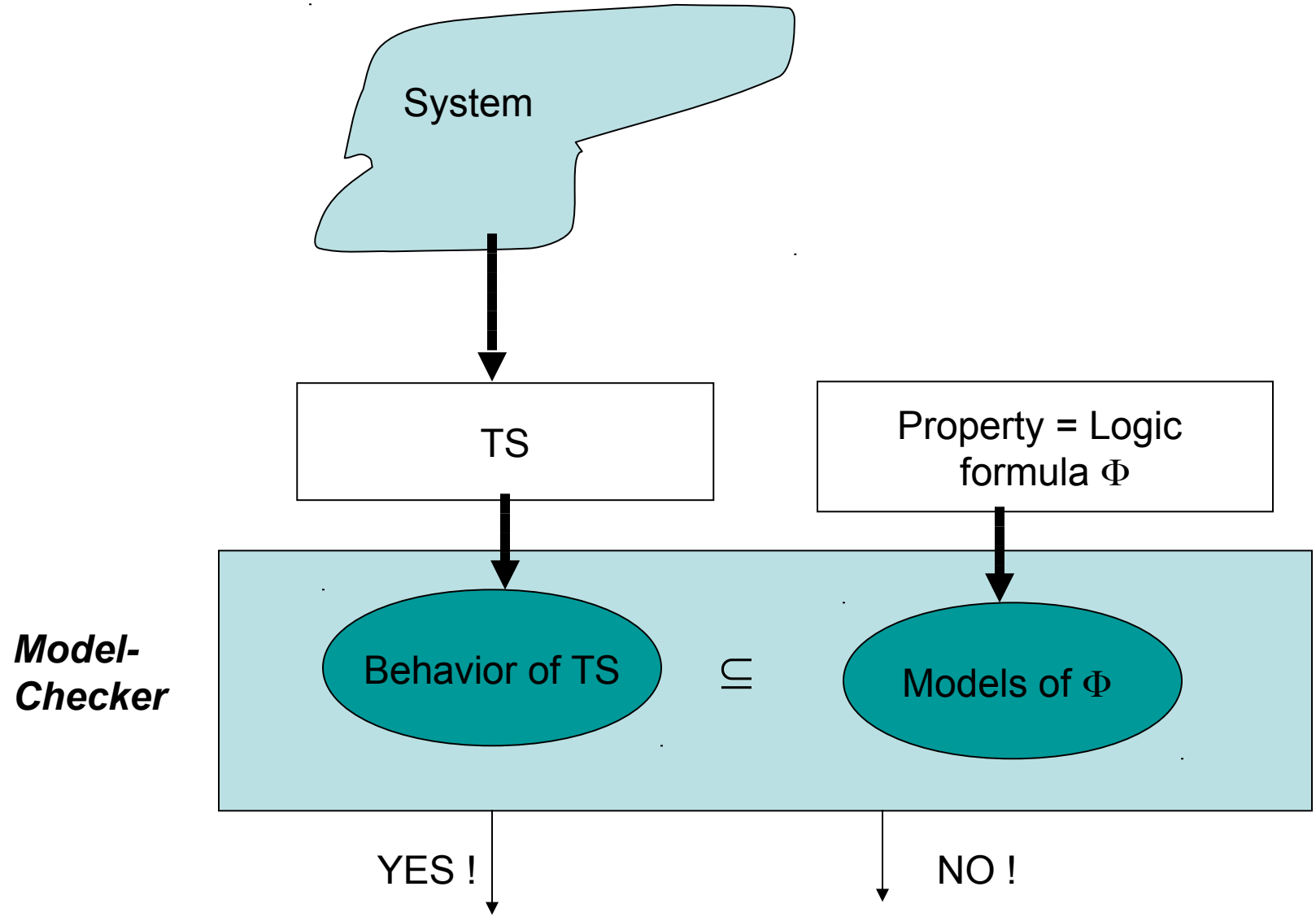
- The behavior of a transition system is:
  - Its set of runs.
  - Its set of computations.

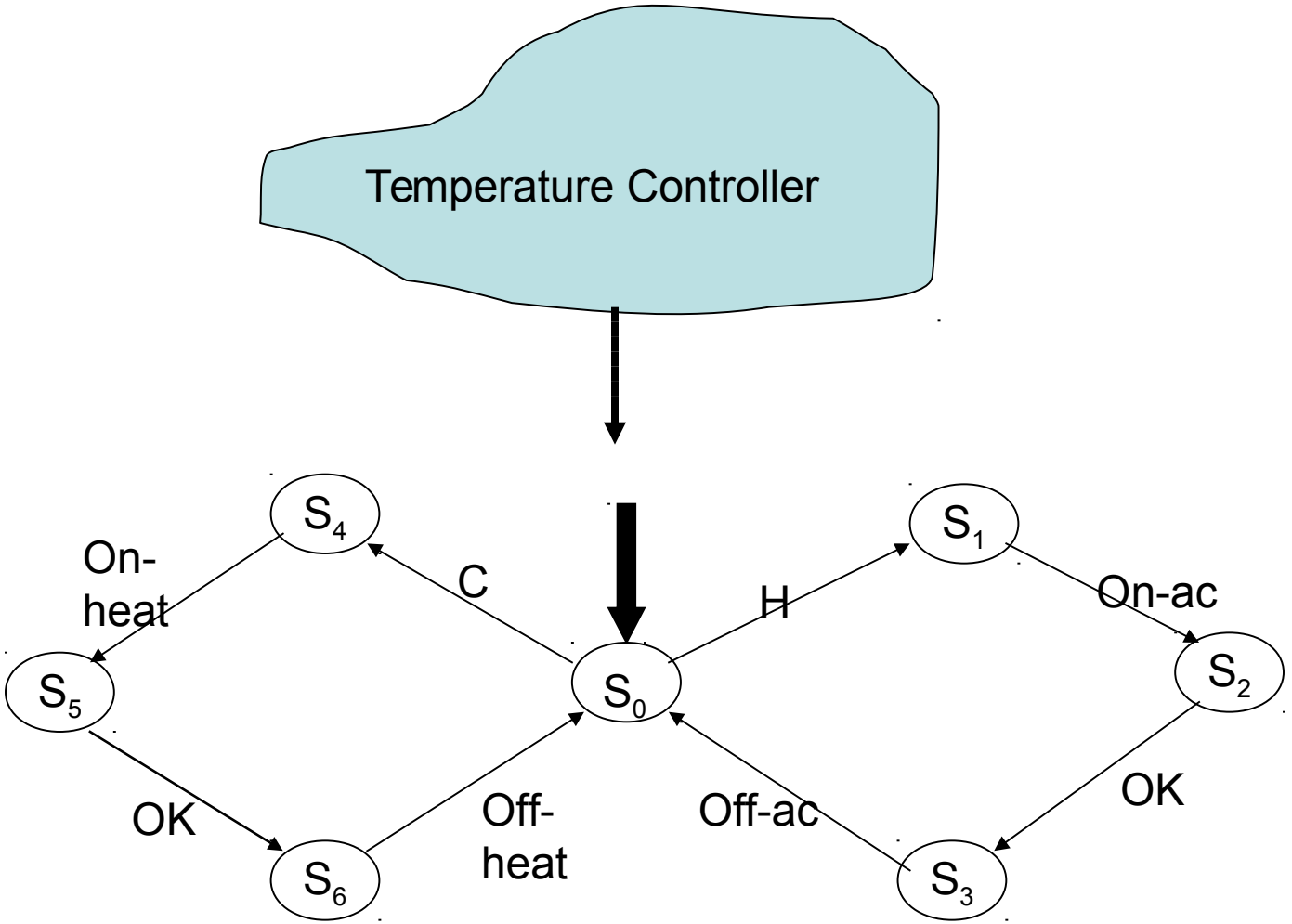
- Does the behavior of TS have the desired property?
  - Does every computation of the transition system have the desired property?
  - *In no computation, C is immediately followed by On-Ac.*

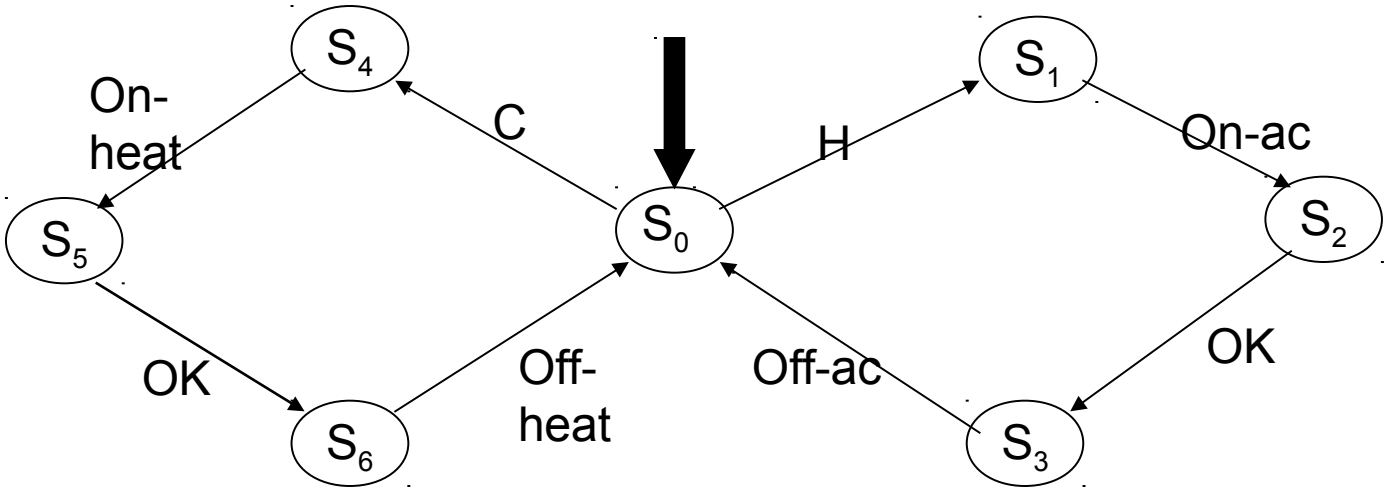
The Verification Setting



The Verification Setting




Temperature Controller
S4
On-
heat
C
S1
H
On-ac
S5
S0
S2
OK
Off-
heat
Off-ac
OK
S6
S3



H on-ac OK off-ac C on-heat OK off-heat

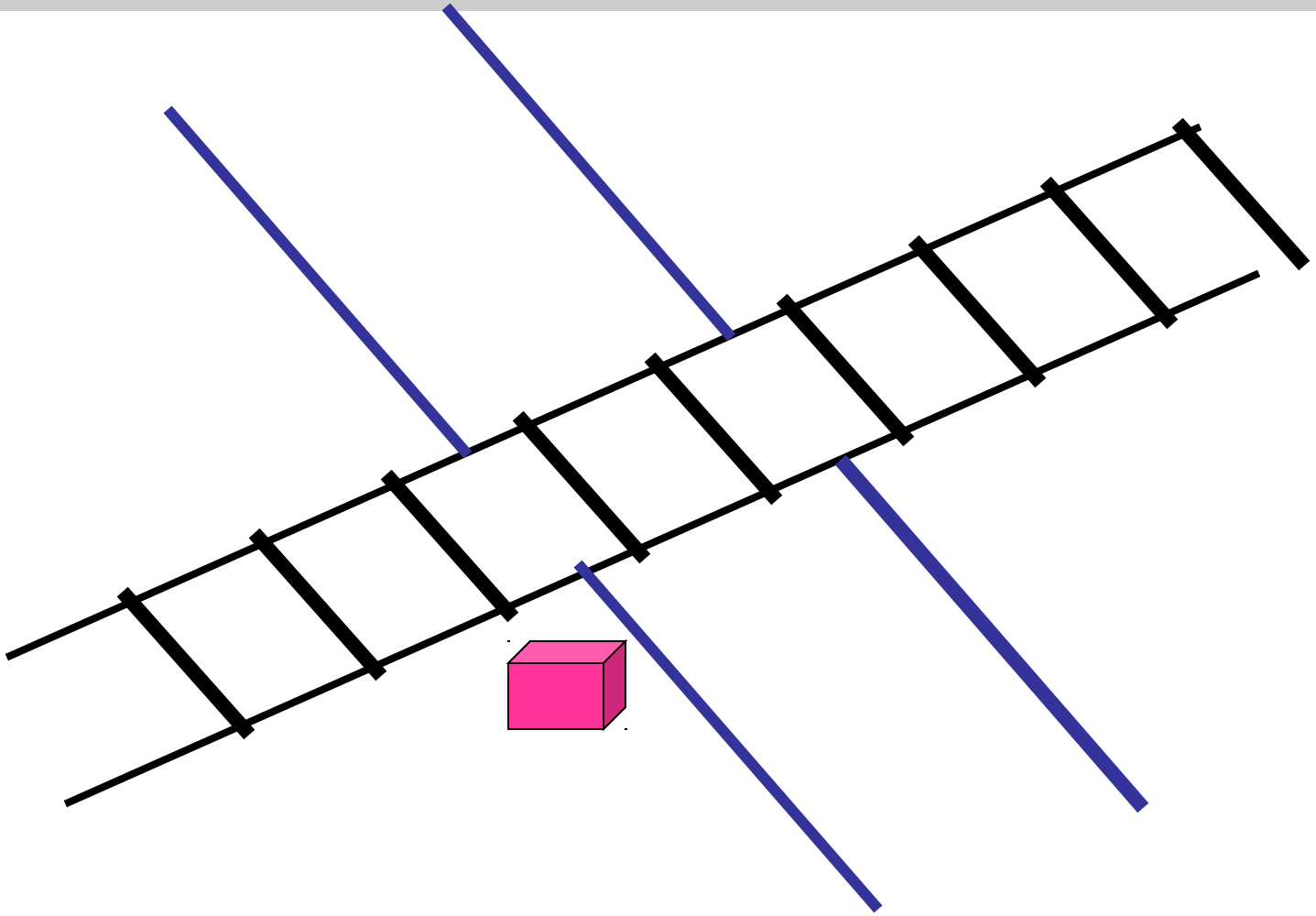
C on-heat OK off-heat C On-heat OK Off-heat

It is often convenient to consider both finite and infinite computations!

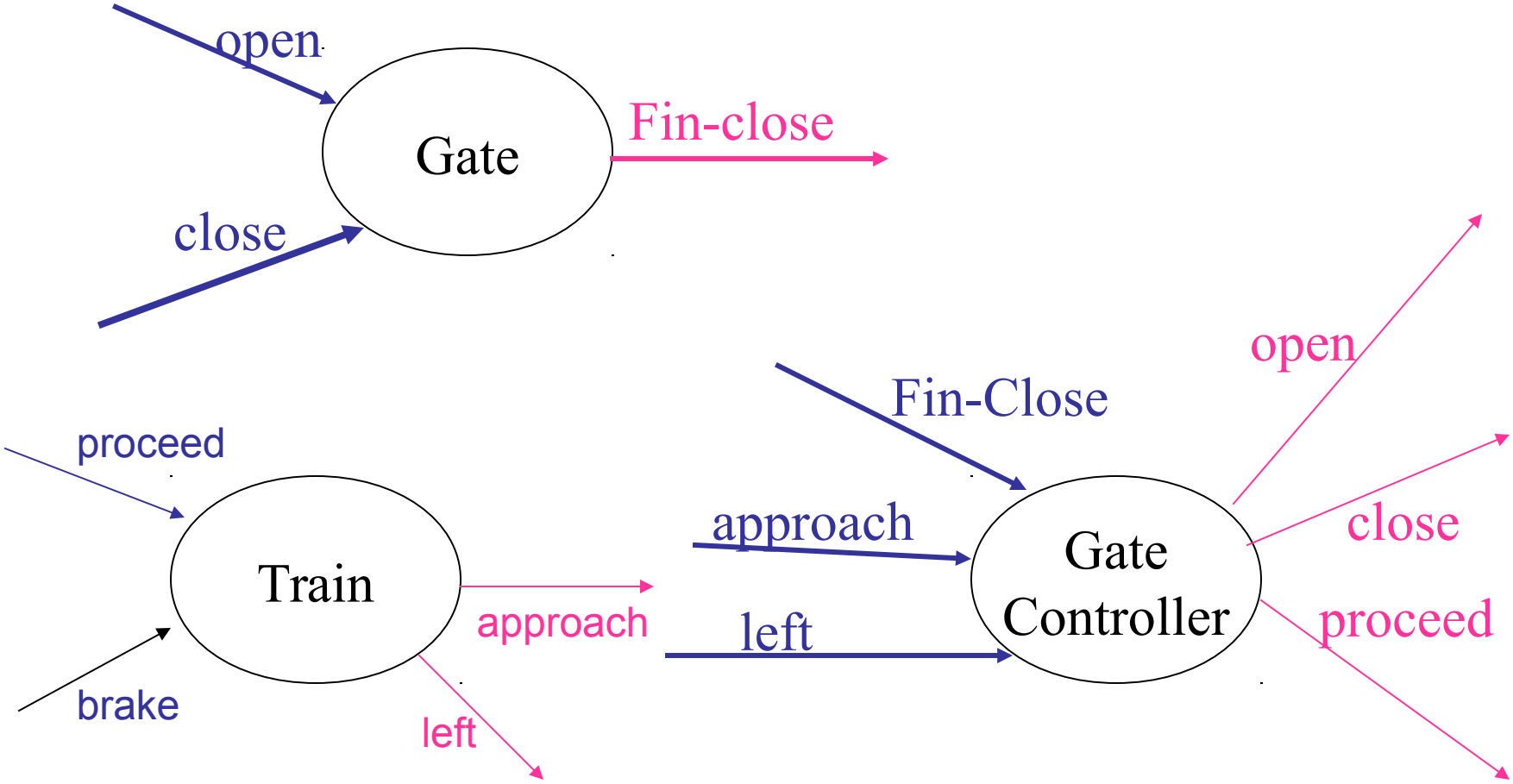
Network of Transition Systems

- In general, the system will contain multiple components.
- The components will coordinate by communication.
  - Send/receive messages (asynchronous)
  - Perform common actions together (synchronous, hand-shake).
    - hand-shake is usually a convenient abstraction.

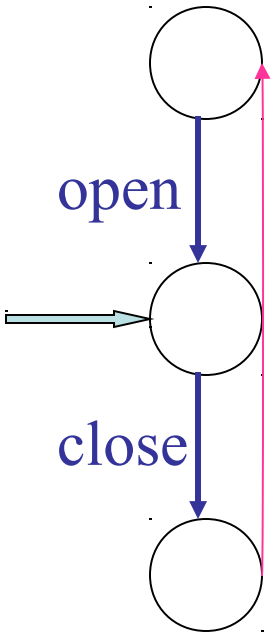
Train Crossing Example



The Signal Space

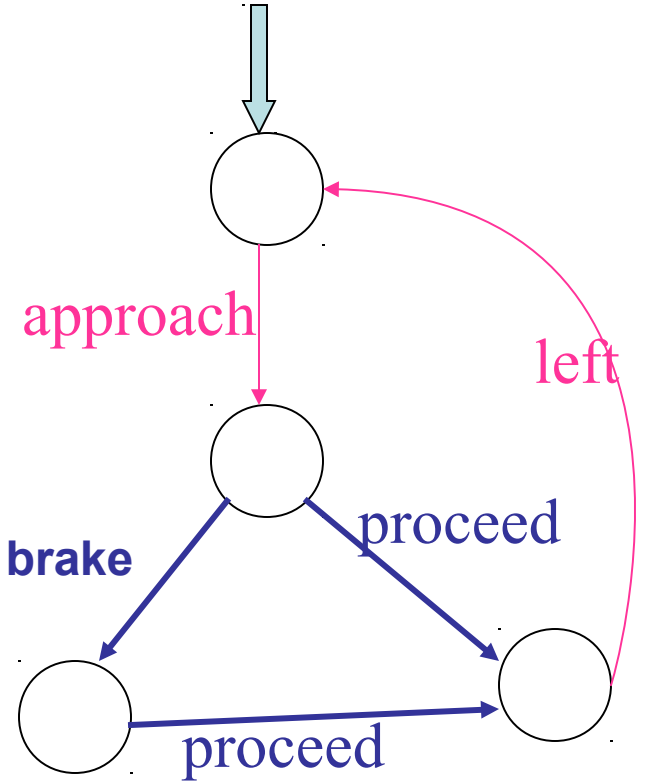


The Gate and Train Transition Systems



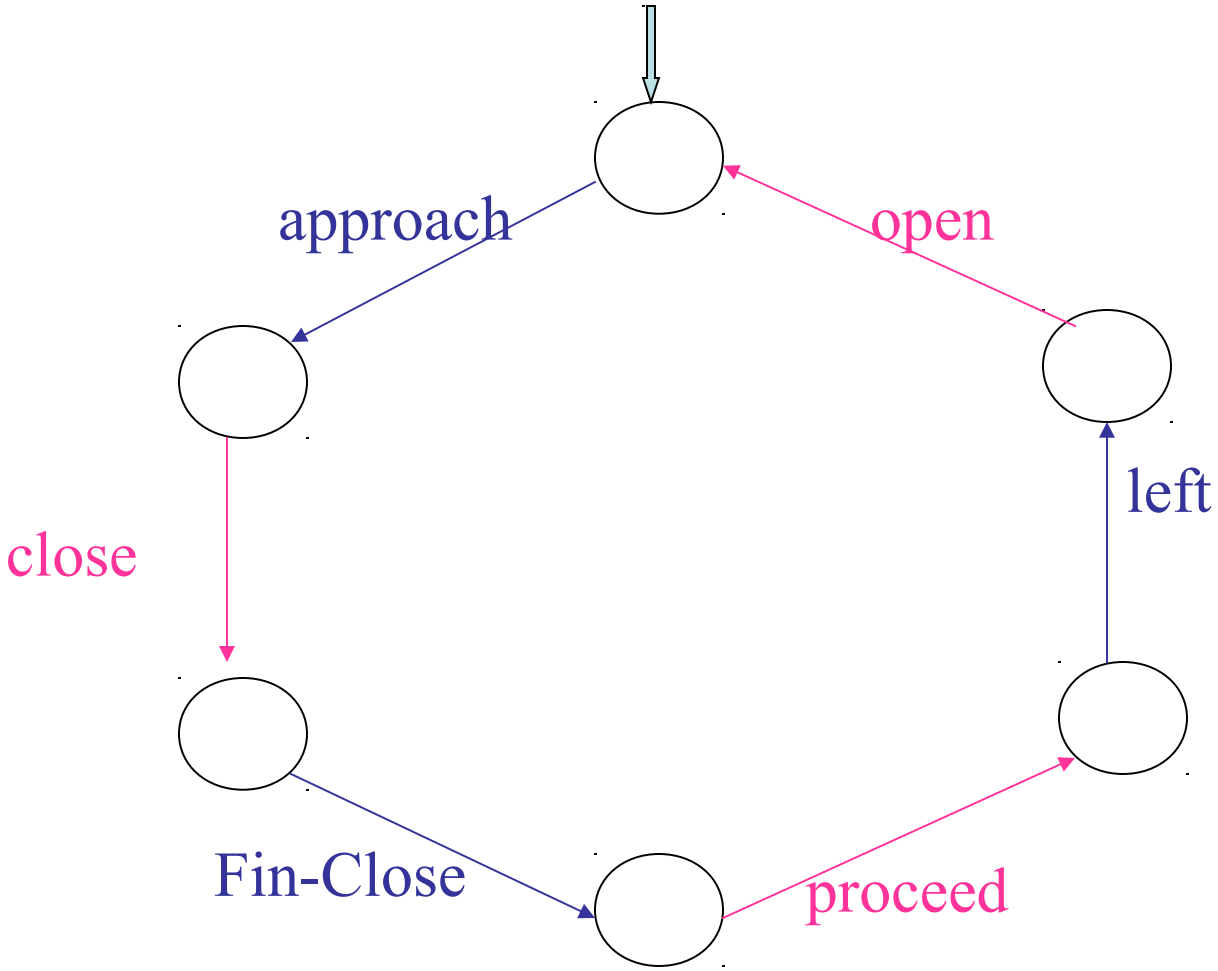
Gate

Fin-Close



Train

The Gate Controller Transition System

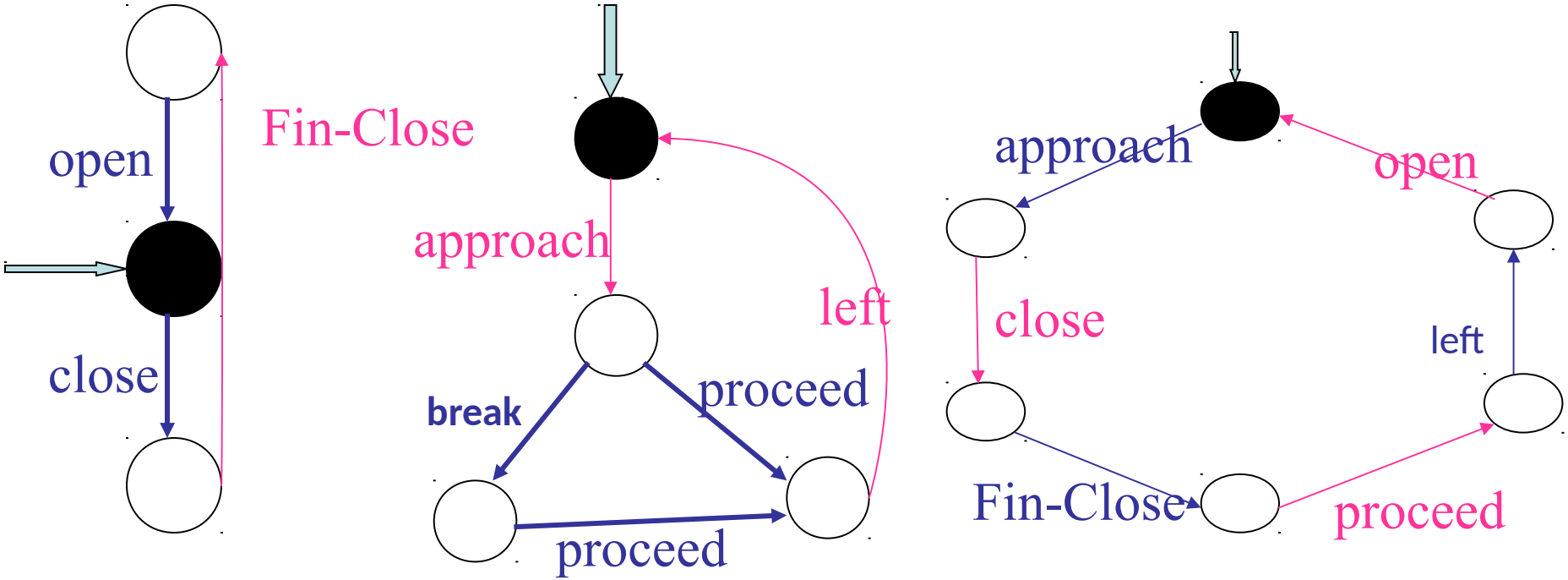


Parallel Composition

- The communication is synchronous/ hand-shake.
- Perform common actions together.

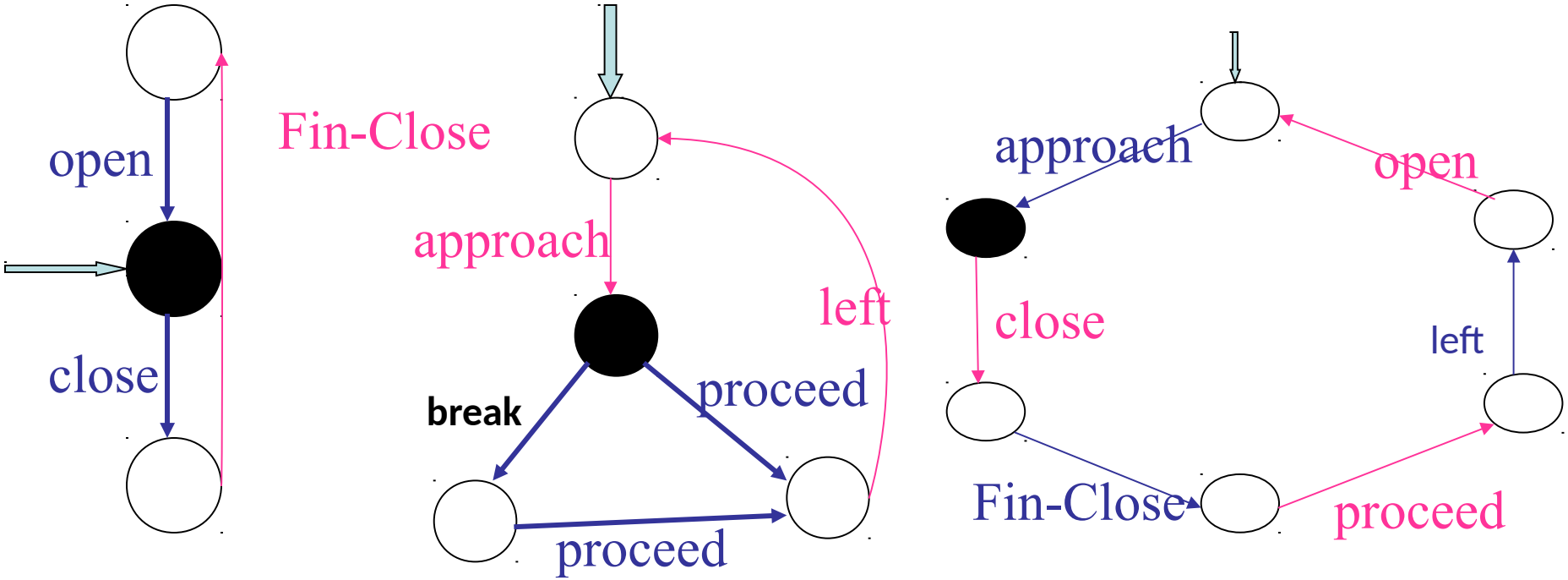
- TS = TrainTS || Gate-ControllerTS
  || GateTS

Parallel Composition



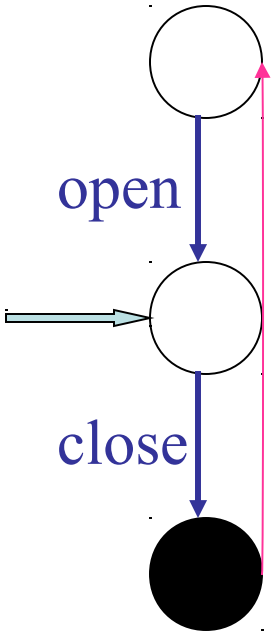
Enabled actions ?

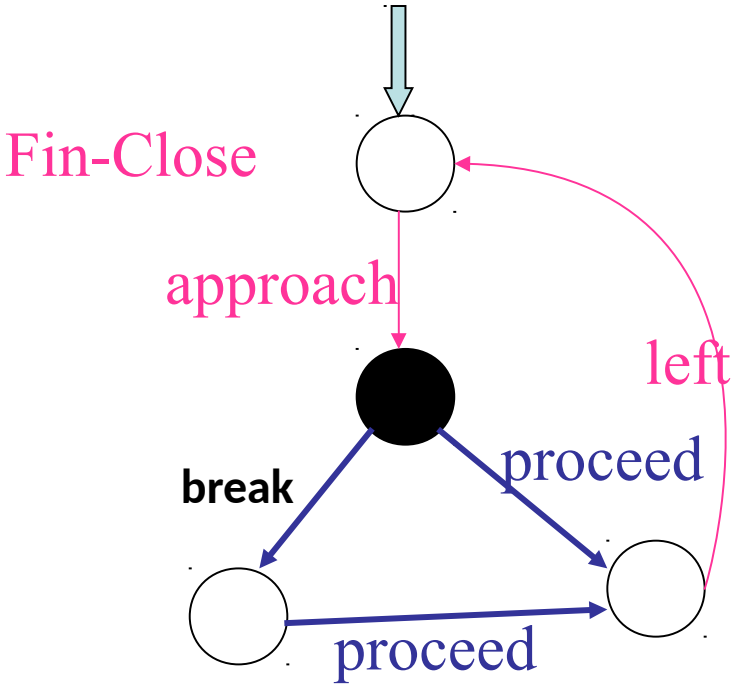
Parallel Composition

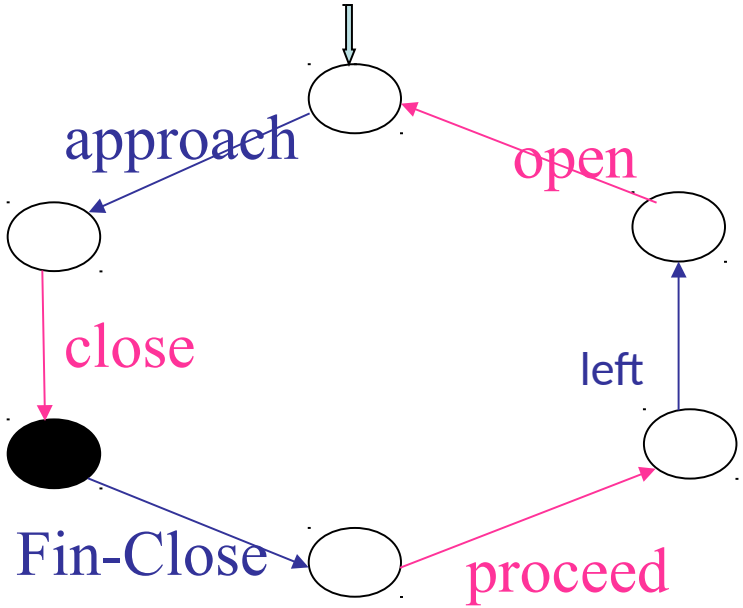


Enabled actions ?

Parallel Composition

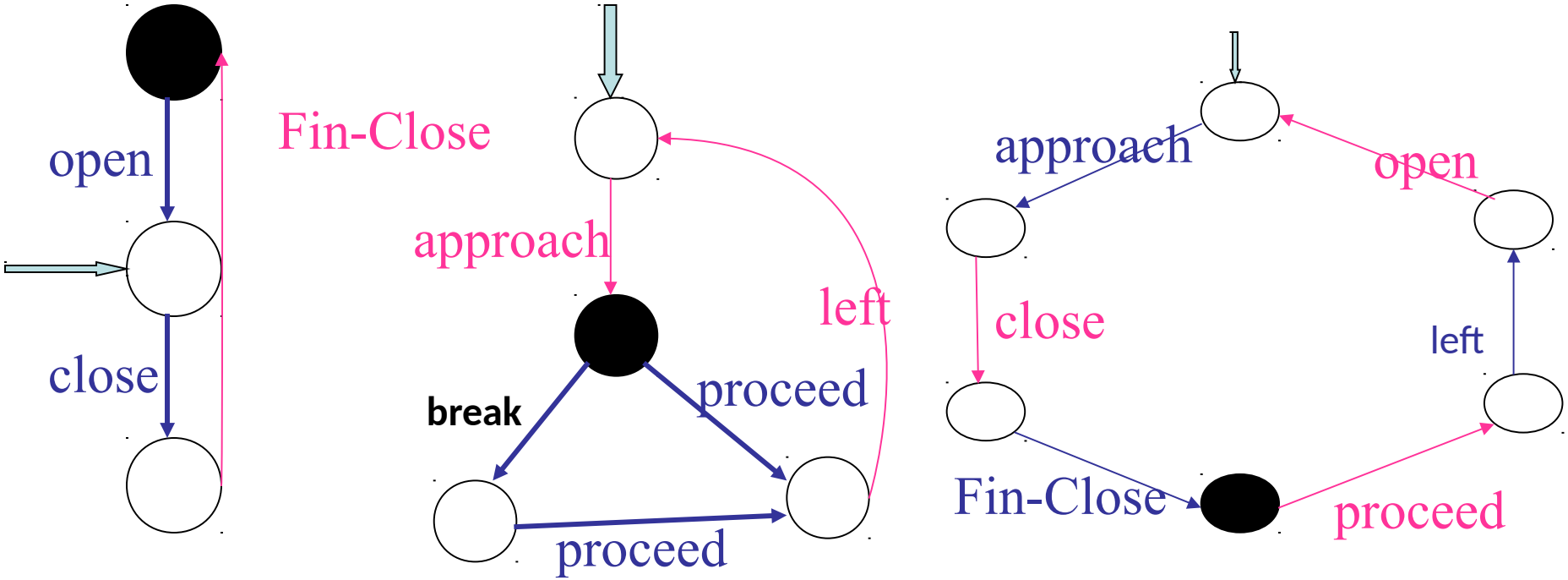






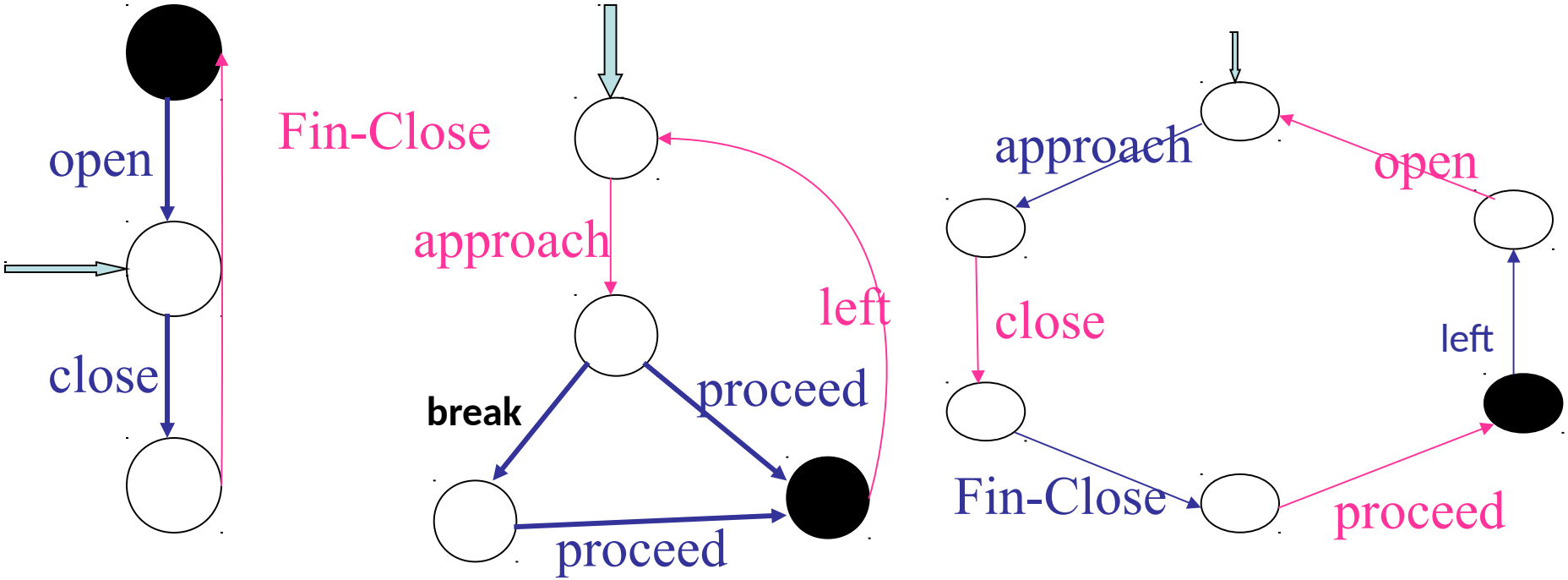
Enabled actions ?

Parallel Composition



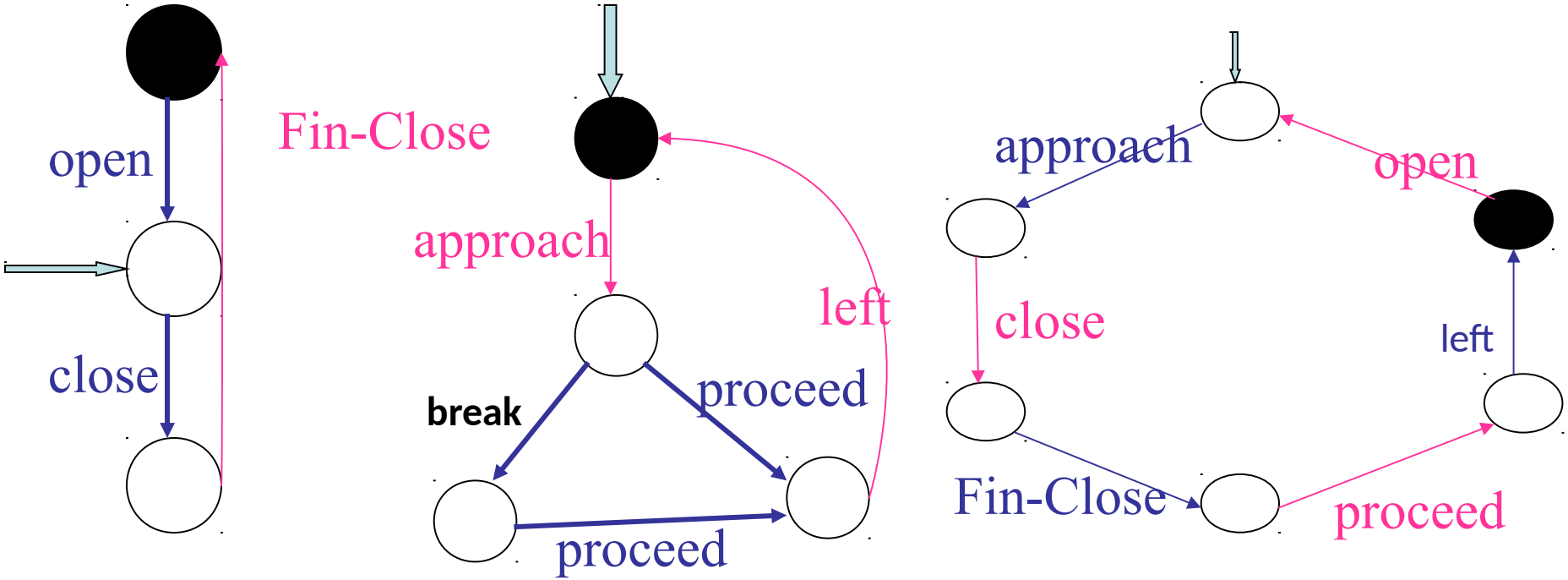
Enabled actions ?

Parallel Composition



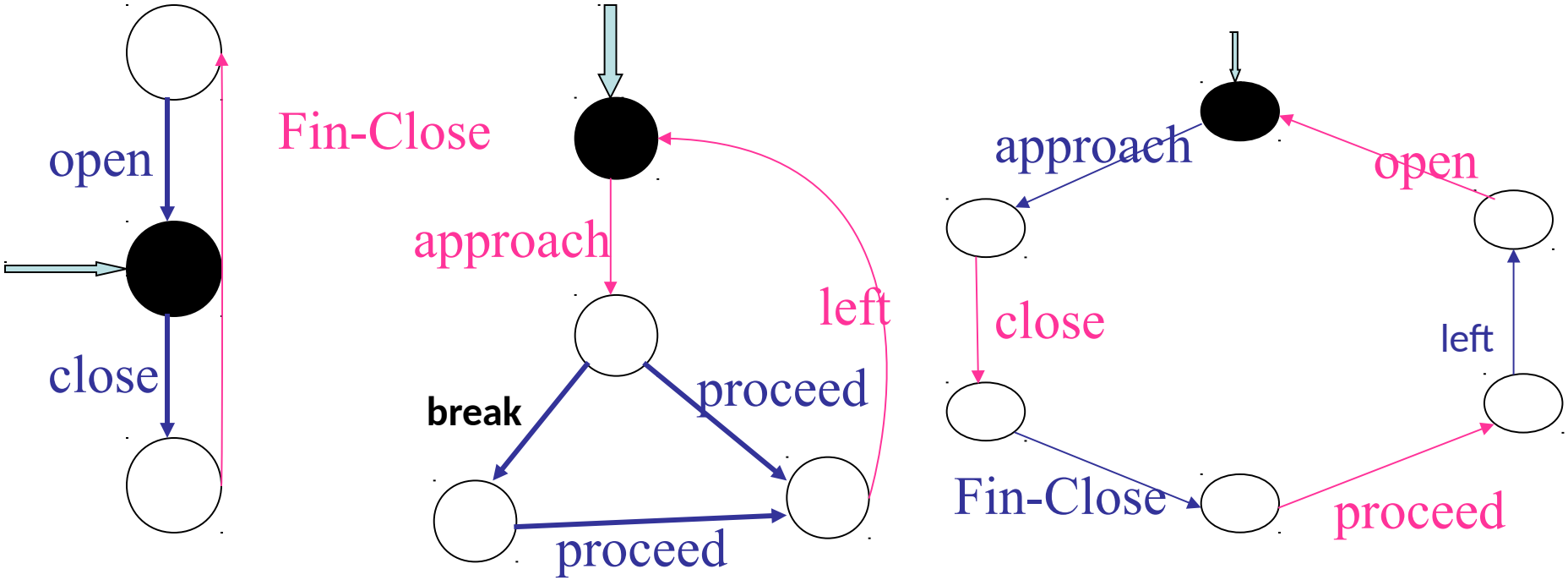
Enabled actions ?

Parallel Composition



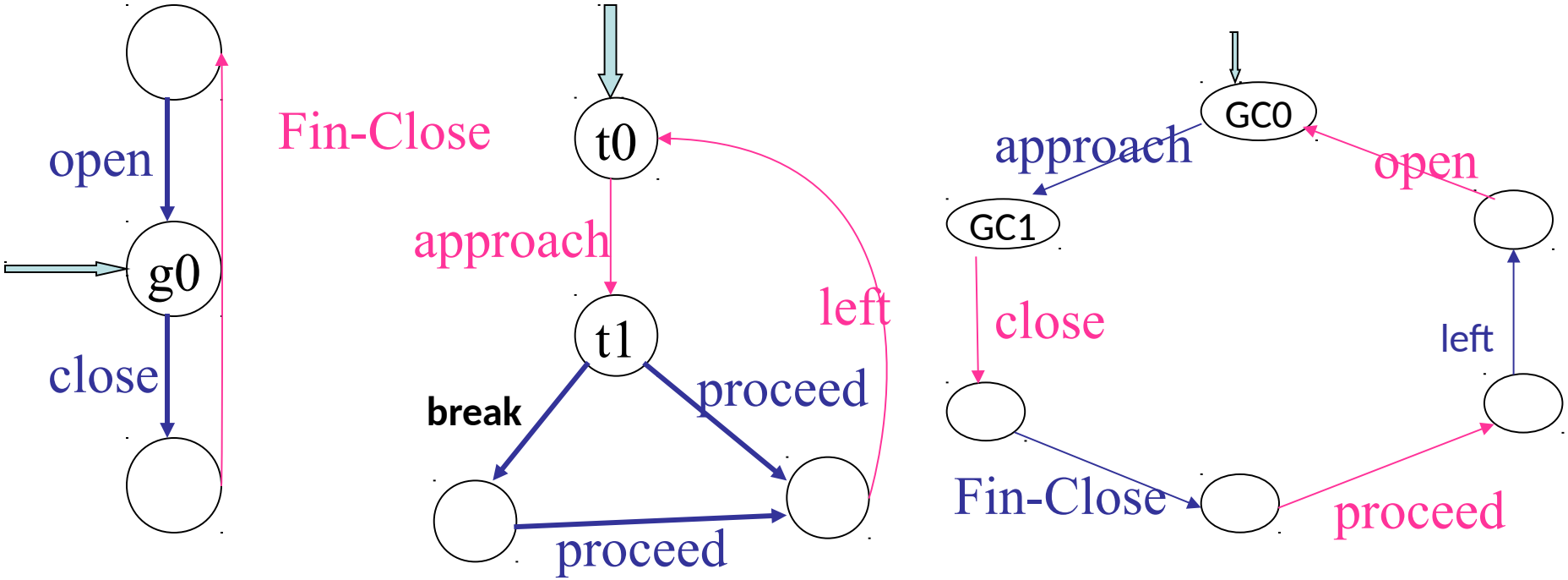
Enabled actions ?

Parallel Composition



Enabled actions ?

Parallel Composition



Parallel Composition

TS = TrainTS || Gate-ControllerTS || GateTS

s = (t, GC, g) A state of TS

(g0, t0, GC0) —approach→ (g0, t1, GC1)

t0 —approach→ t1 (TRAIN)

GC0 —approach→ GC1 (Gate-Controller)

State Space Explosion

- $TS = TS_1 \,||\, TS_2 \ldots \,||\, TS_n$
- TS is presented implicitly!
  - Fix a communication convention
  - Present $TS_1, TS_2, \ldots, TS_n$
- We wish to analyze TS and often implement TS.
- But constructing TS first explicitly is often hopeless.
- $|TS_i| = 10 \quad n = 6$
  - $|TS| = ?$ (worst case)