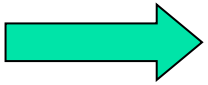
Course Outline (lectures)

- Introduction
  - Characteristics of RTS
- Real Time Operating Systems (RTOS)
  - OS support: tasking, scheduling, resource handling, OSEK
- Real Time Programming Languages
  - Language support, e.g. Ada tasking
- Scheduling and Timing Analysis
  - Worst-case execitiom and response time analysis
  - Resource reservation and resource servers
- Distributed real time systems
  - Real Time Communication: CAN Bus
- Workload Models (advanced topic)
  - Graph-based task models
- Multiprocessor real-time systems (advanced topics)
  - Architectures and real-time scheduling
- Design and Validation (advanced topics)
  - Modeling and Verification



Today's topic

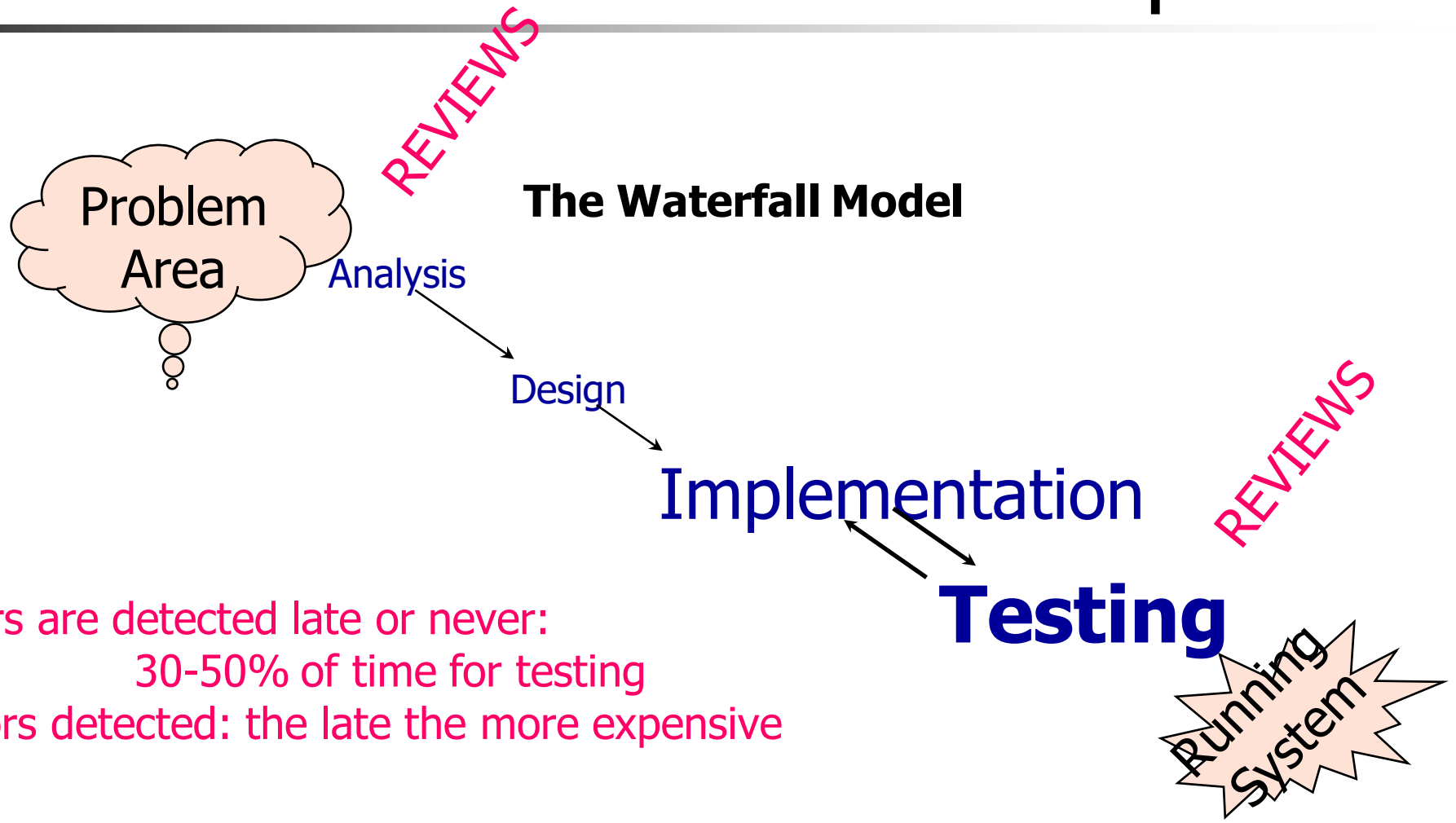
Modeling Real-Time Systems

Plan

- Motivation: Why Modeling?
  - Basic concepts
- Finite Automata: State Machines
  - Examples
- Timed Automata: State Machines with clocks
- Modeling Ada Programs as Automata

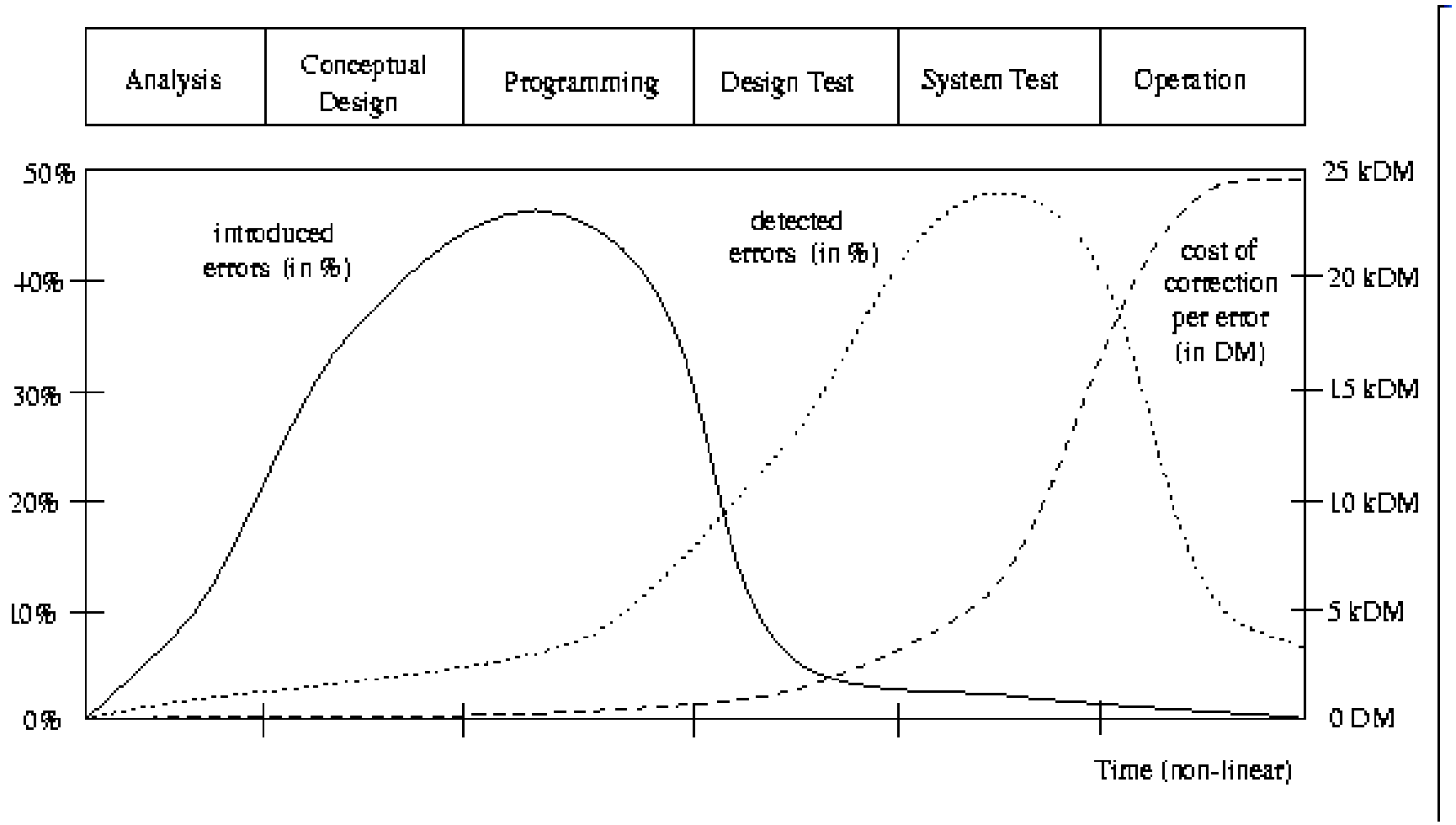
Motivation: Why modeling?

Traditional software development



- Errors are detected late or never:
  30-50% of time for testing
- Errors detected: the late the more expensive

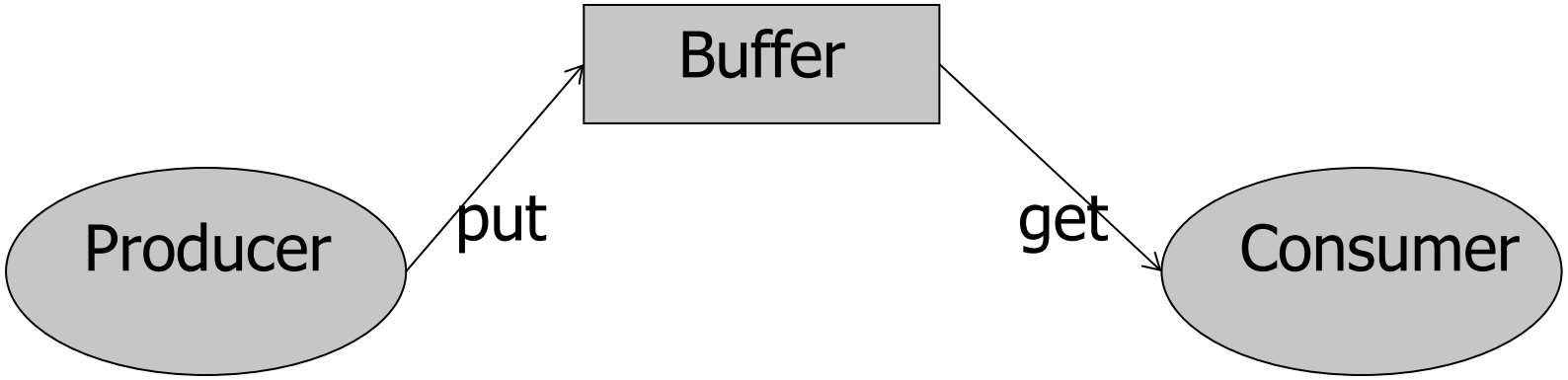
Introducing, Detecting and Correcting errors



Finding errors as early as possible!

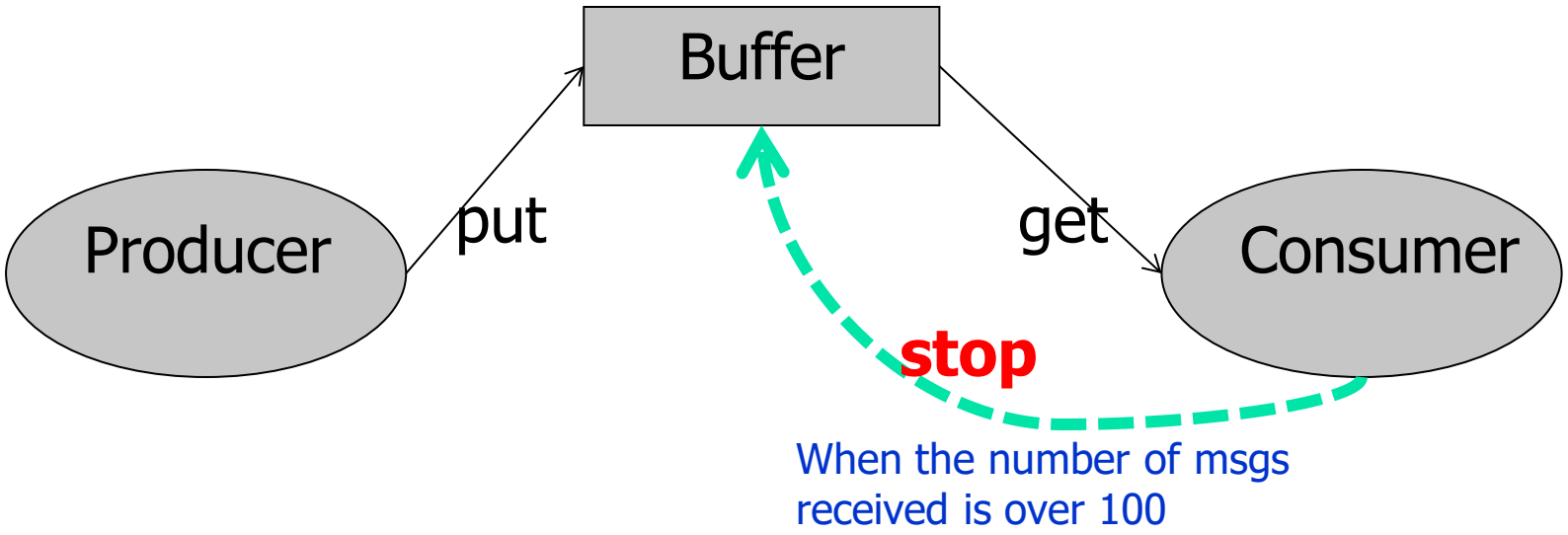
HOW?

Example: the Ada assignment (all 82 students were wrong!)

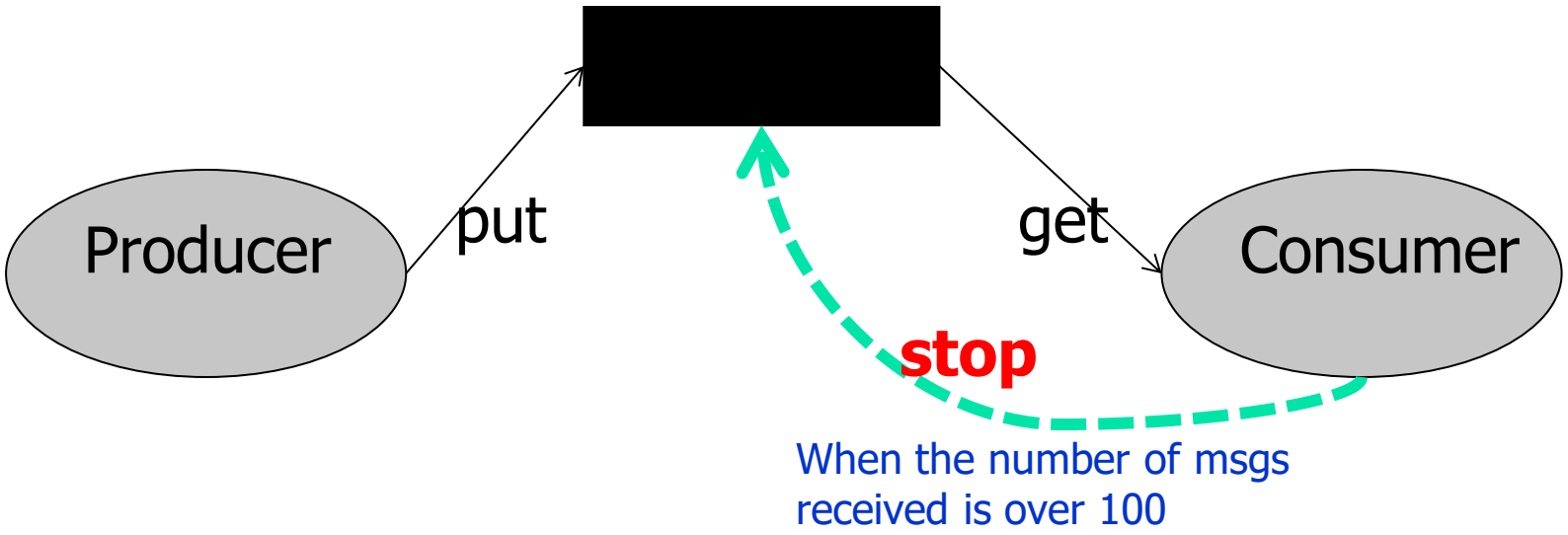


When the number of messages received is over 100, Consumer should **tell Buffer and Producer to stop**

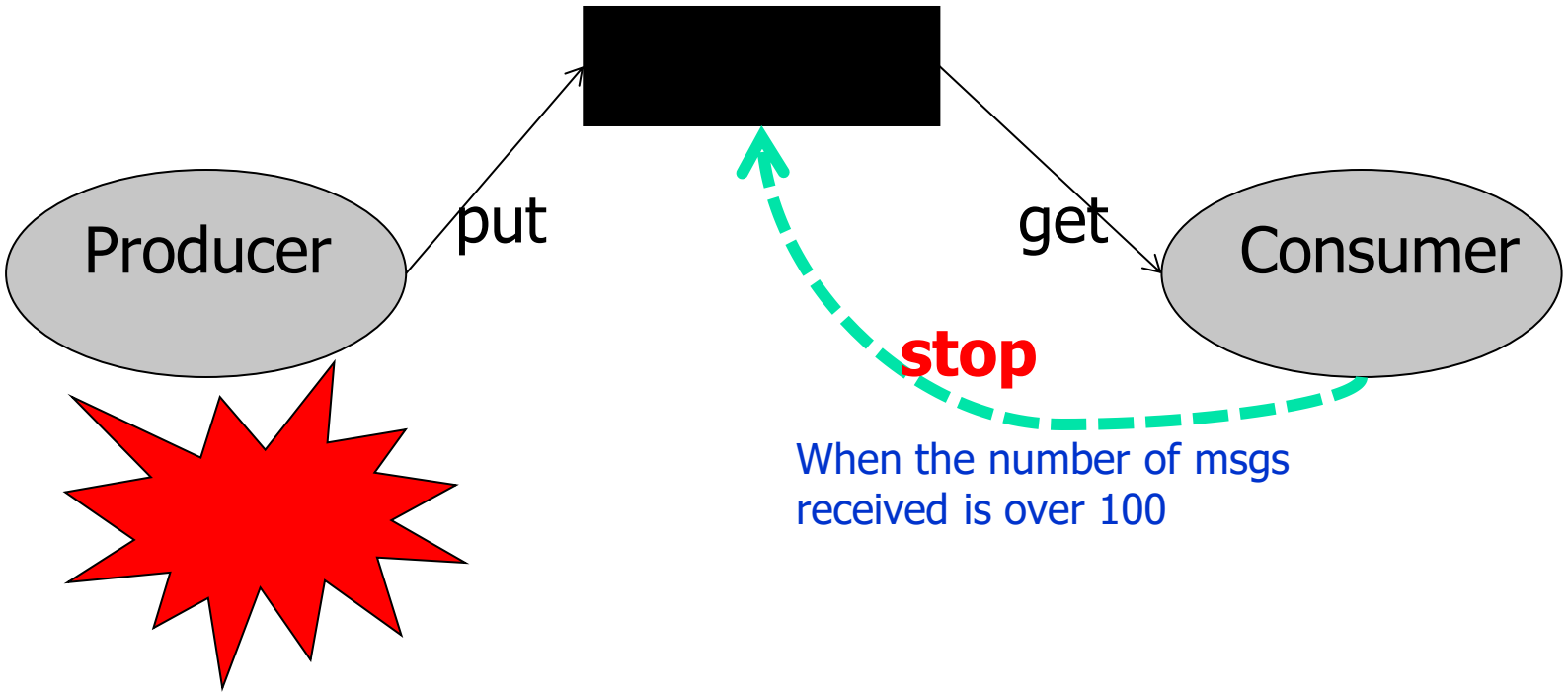
Example: the Ada assignment (all 82 students were wrong!)



Example: the Ada assignment (all 82 students were wrong!)

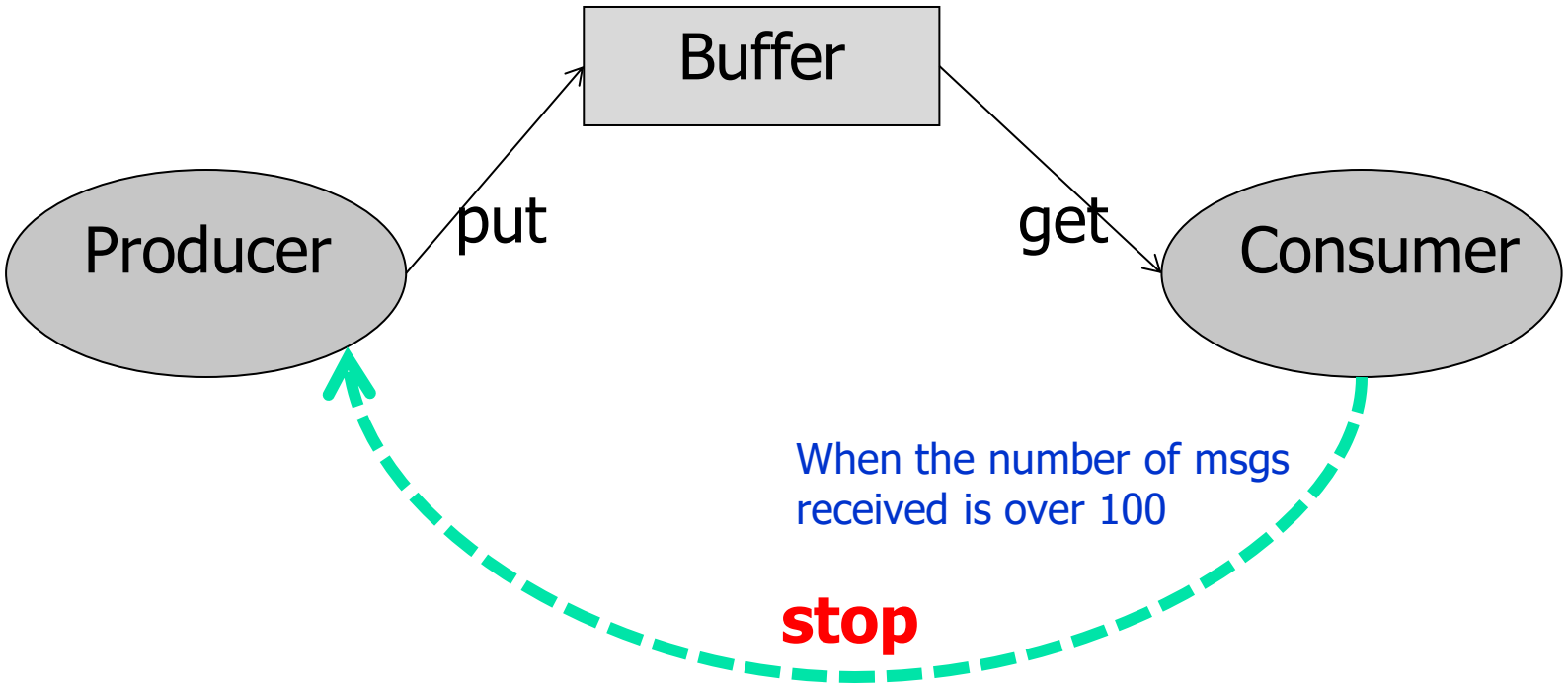


Example: the Ada assignment (all 82 students were wrong!)



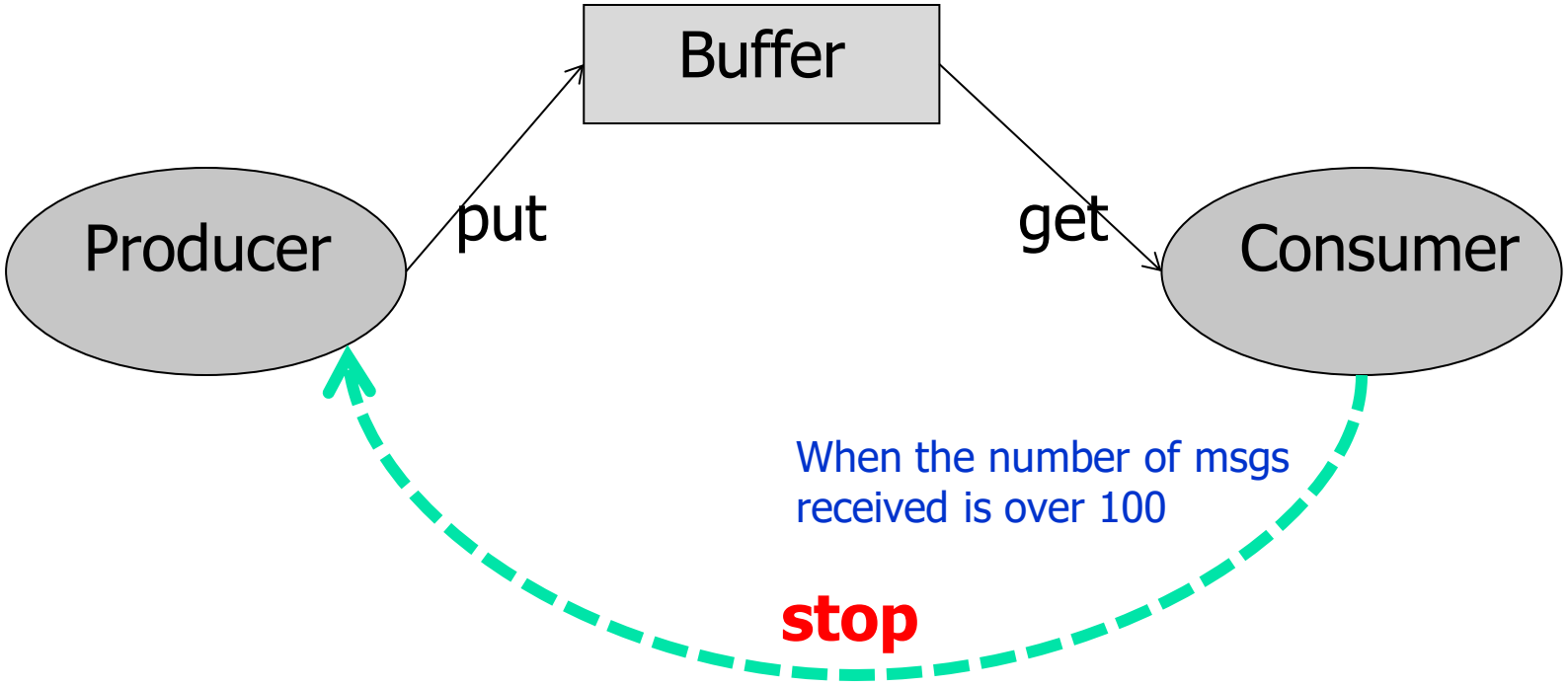
The run-time system will generate an Exception !!!!

Example: the Ada assignment (all 82 students were wrong!)



Solution?

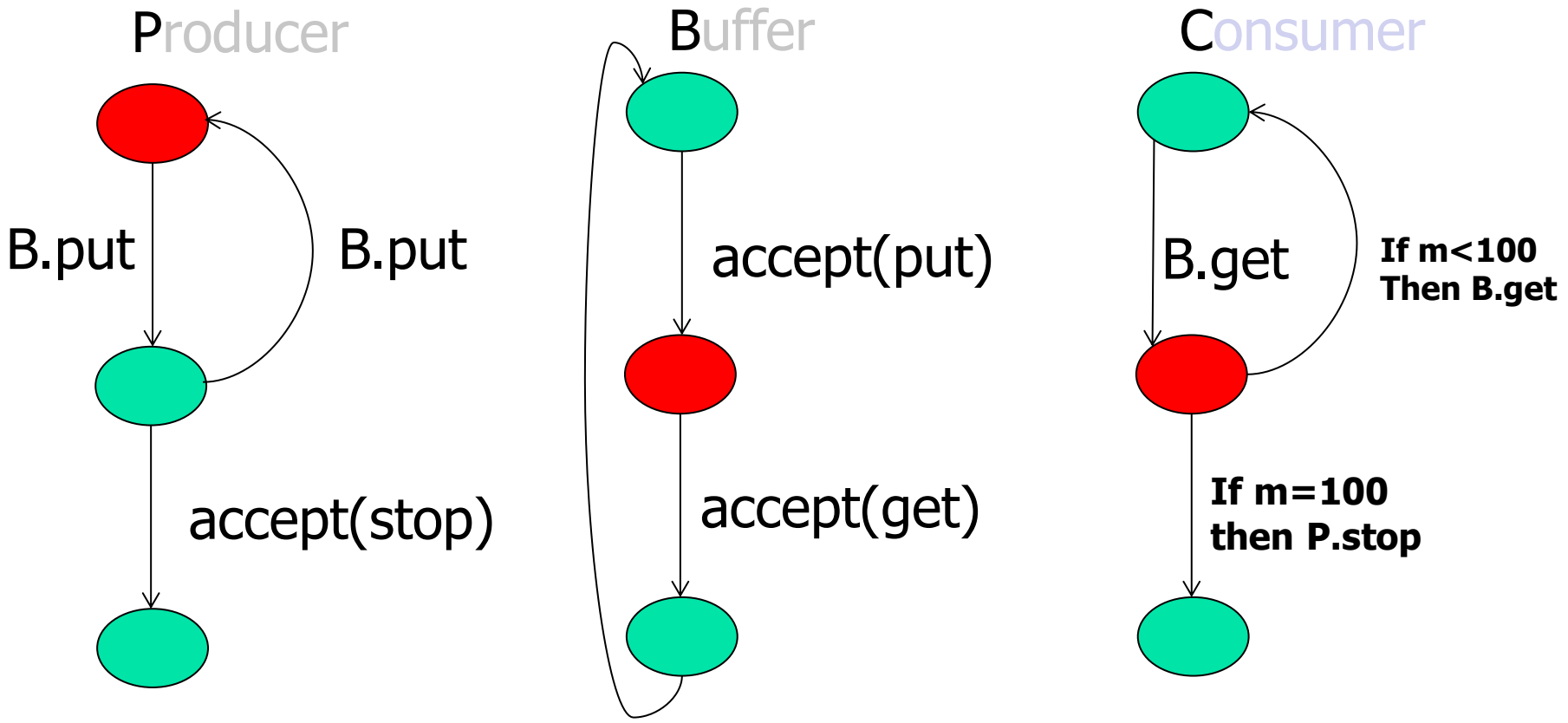
Example: the Ada assignment (all 82 students were wrong!)



Deadlock! ☹

If Buffer if full, Producer is trying to put and blocked, Buffer is waiting for Consumer to empty one-slot and Consumer is waiting for Producer to stop

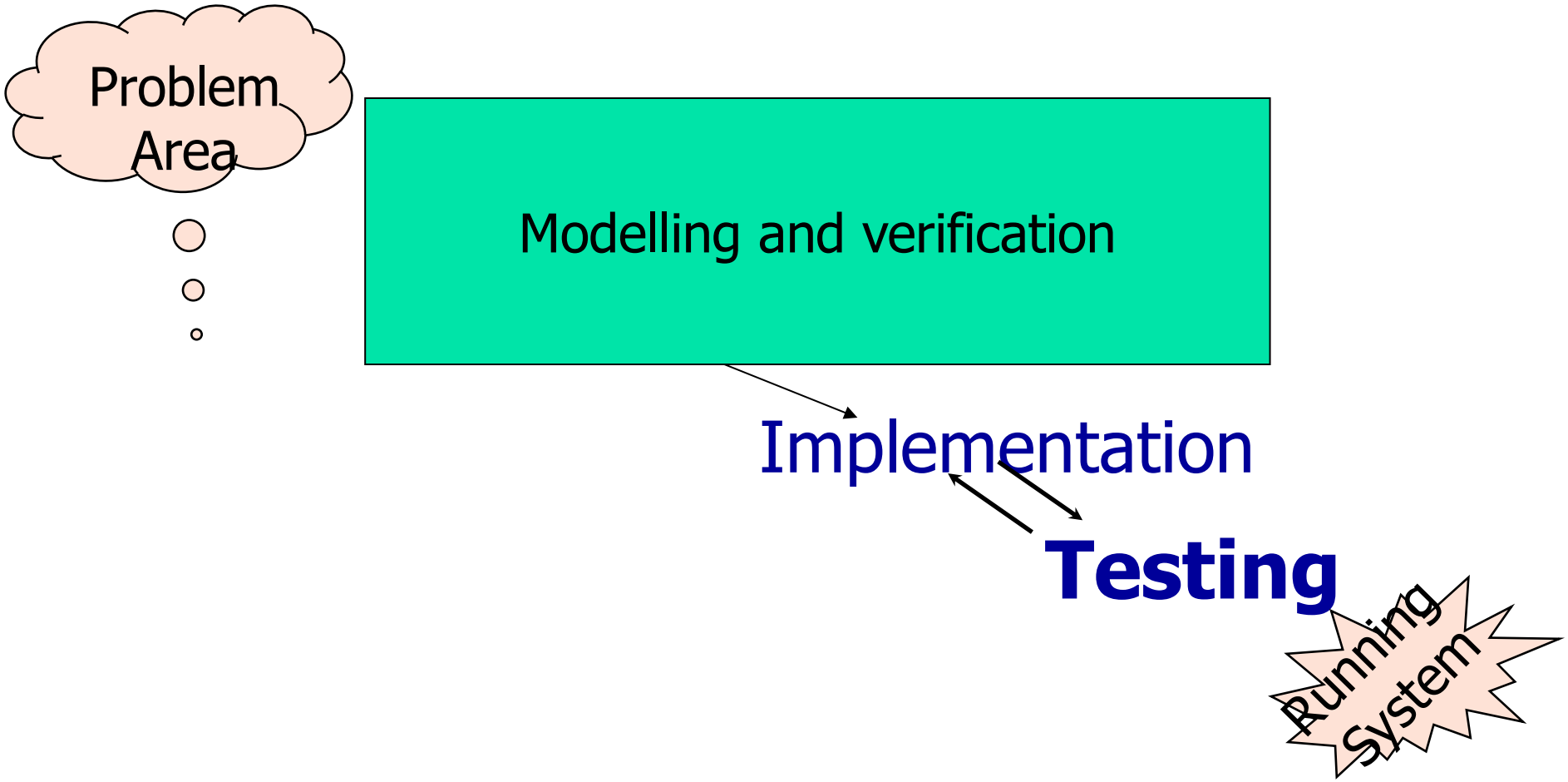
Here is the deadlock!



You can find the Bug using UPPAAL

Which is a software tool for modeling and Verification developed by Uppsala Uppsala (Sweden) & Aalborg University (Denmark)

Software development



Modeling and Verification

- ”modeling” is a design process: describe systematically the abstract behaviour of a system
  - It is to create a model or a design proposal of the system to be developed
- ”verification” is to check whether the model satisfies given requirement specifications
  - It is similar to testing, but testing on the model

Modeling and Verification with Finite Automata (i.e. State Machines)

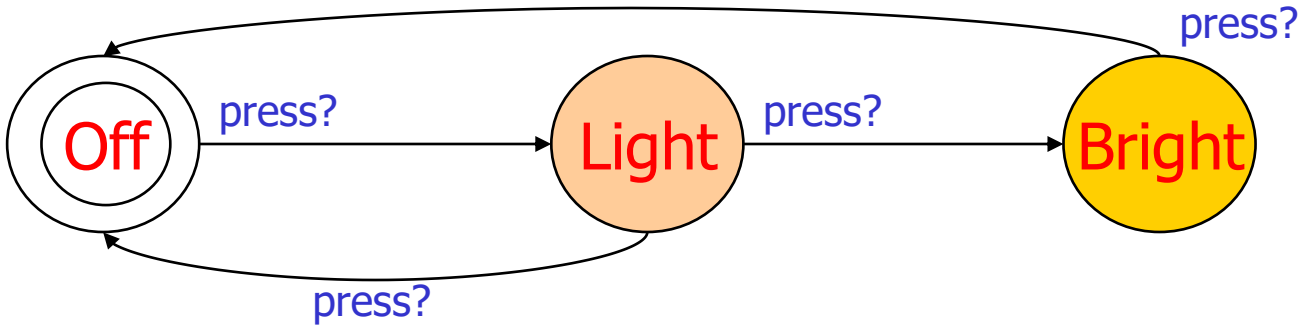
- Modeling
  - Use finite automata to describe

    all the possible executions of system components
  - Model a system as a network of automata
- Verification
  - Check properties of the state machines using software tools like UPPAAL

Finite Automata:

Examples of Modeling

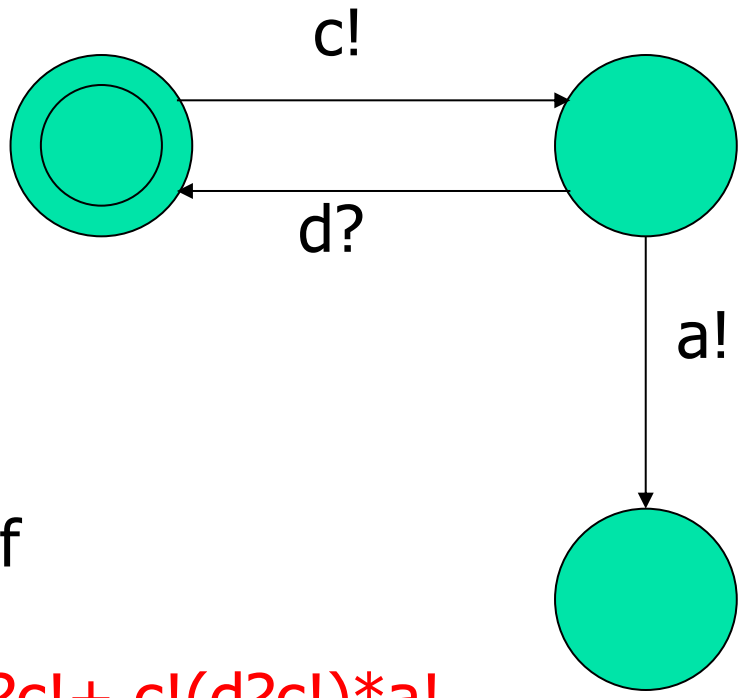
Intelligent Light Control



WANT: if press is issued twice quickly
then the light will get brighter; otherwise the light is turned off.

The double circle is the initial state

Finite Automata



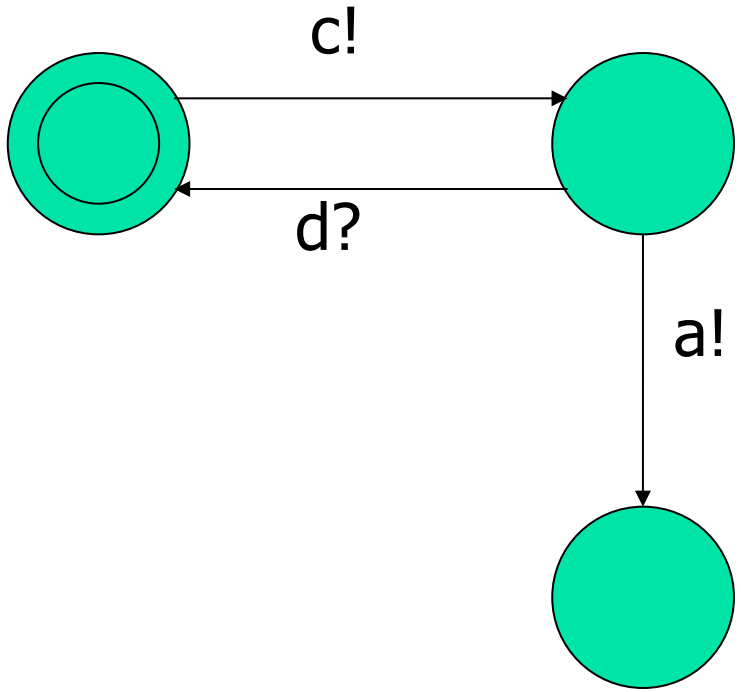
It has 3 states
and 3 transitions

And it accepts
sequences out of
the language:

c! + c!d? + c!d?c!+ c!(d?c!)*a!

Input and Output Actions

We use ? and ! to denote the pairs of complementary actions (i.e. call and accept) in rendezvous communication

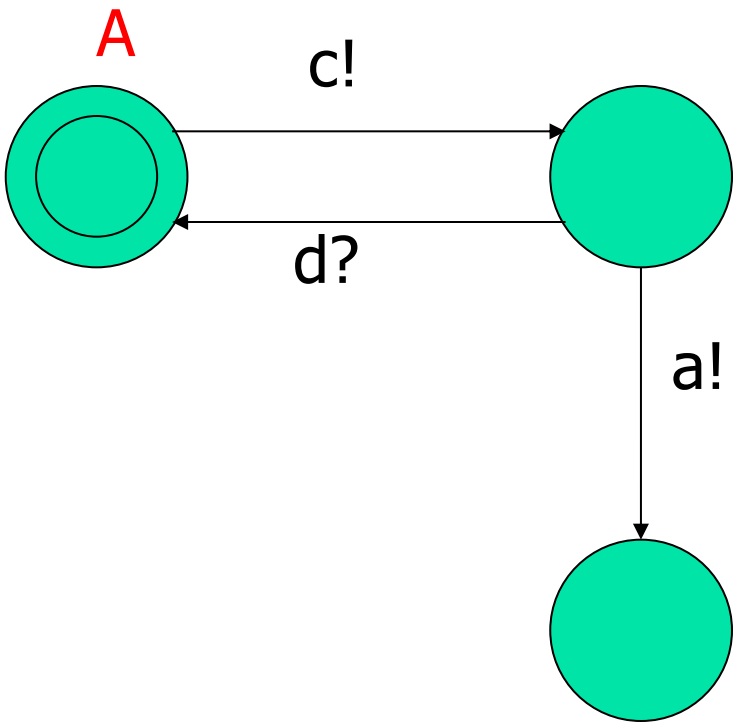


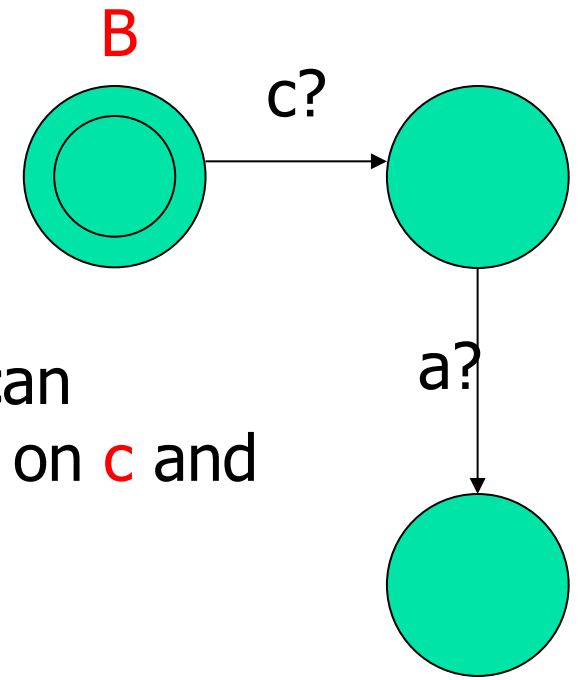
Automata as tasks

- We view an automaton as a task where the action labels stand for synchronization actions e.g. Ada's rendezvous
- The previous example could be a network protocol where c! and d? stand for ″connect !″ and ″disconnect?″ and a! for ″abort!″

Networks of automata

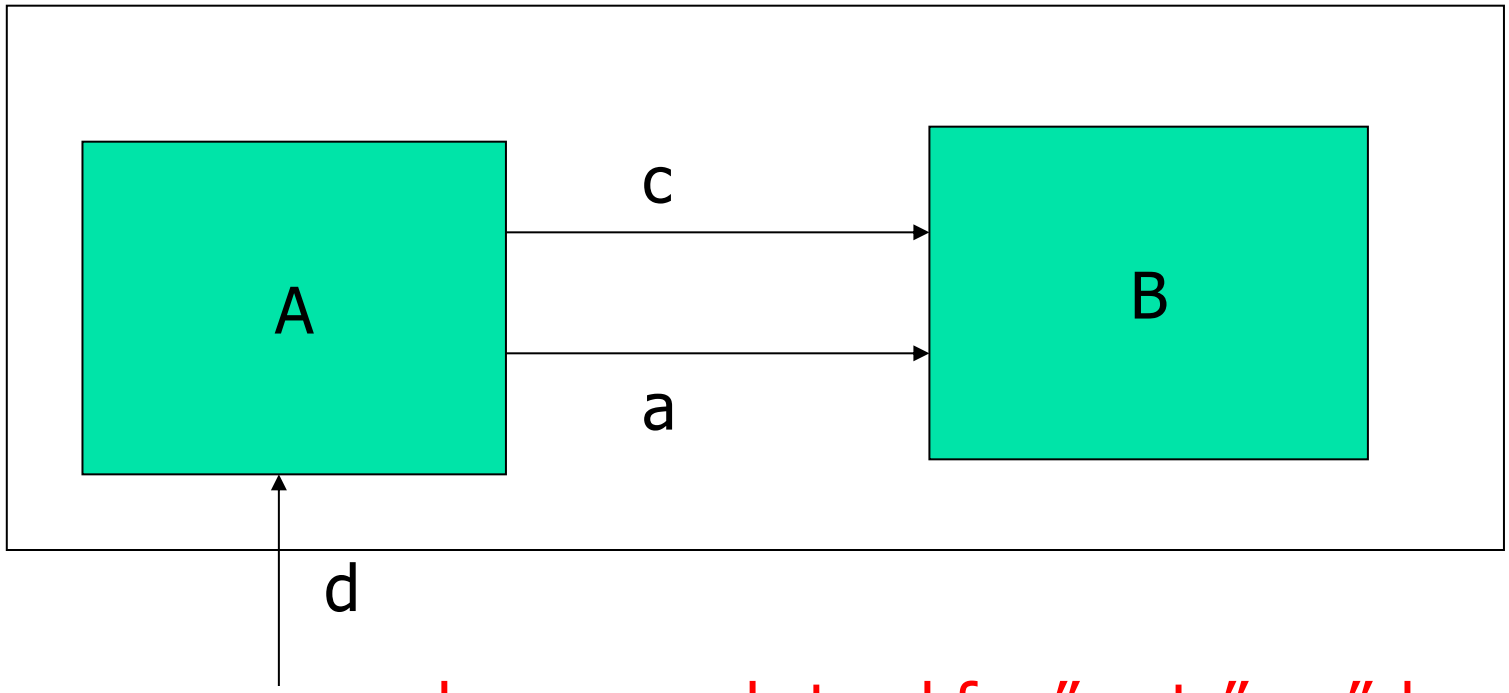
(think about Ada Tasking, entry and call etc)





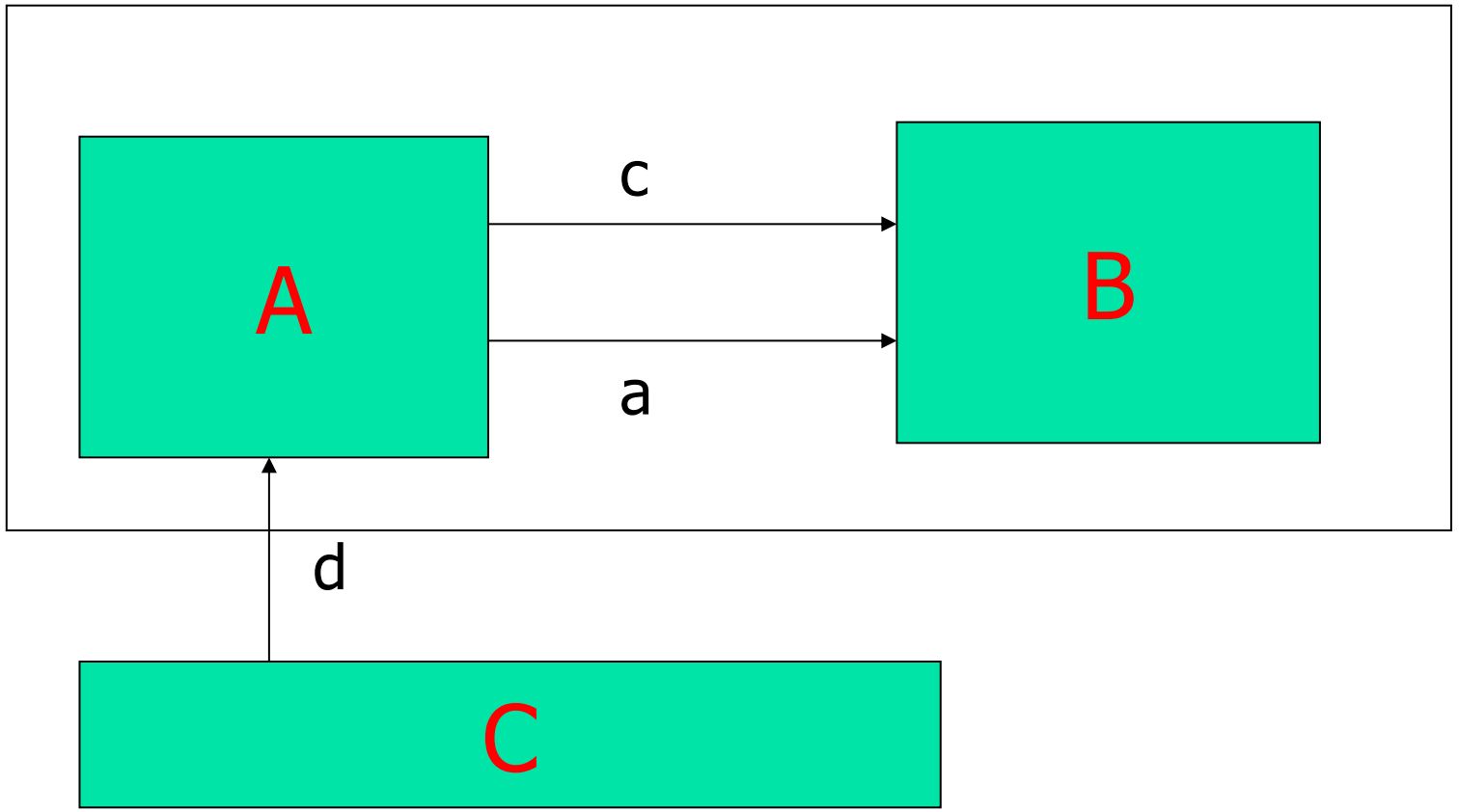
The tasks can synchronize on c and then a

The static architecture of a concurrent system: A || B

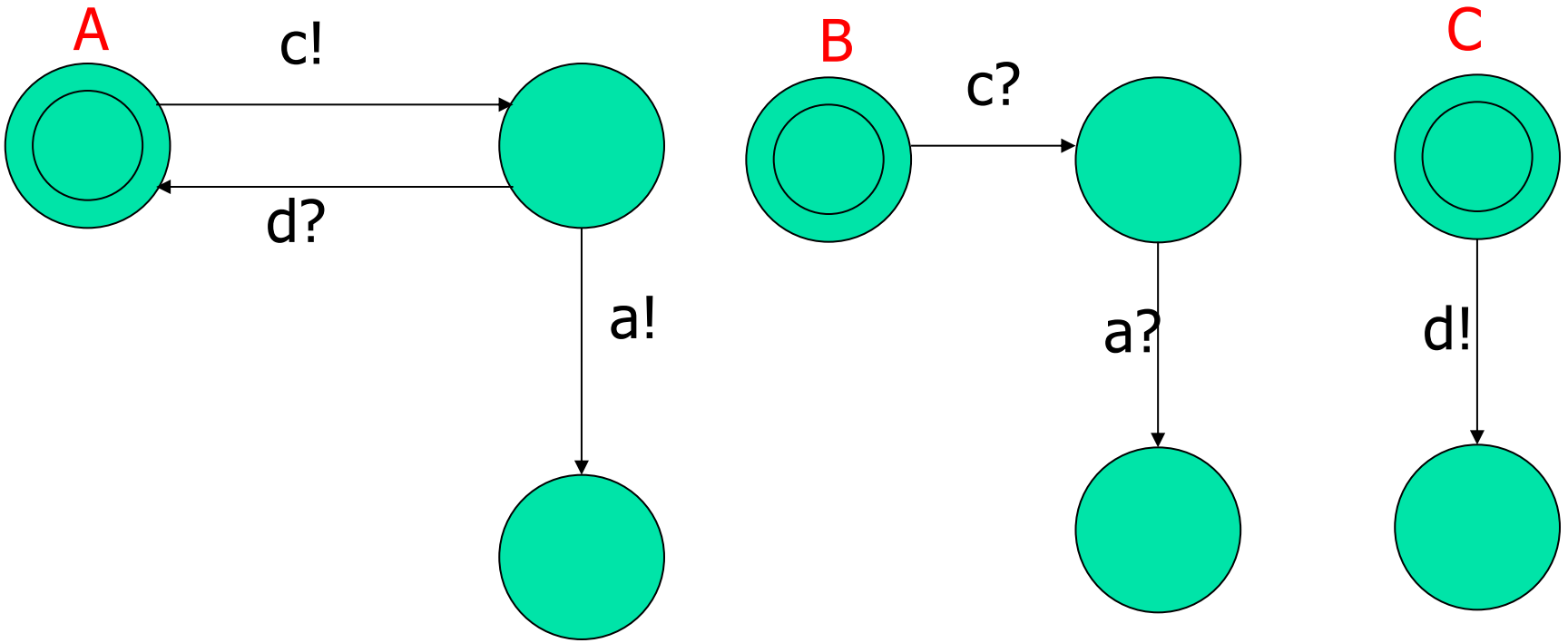


where c, a, d stand for ”ports” or ”channels”

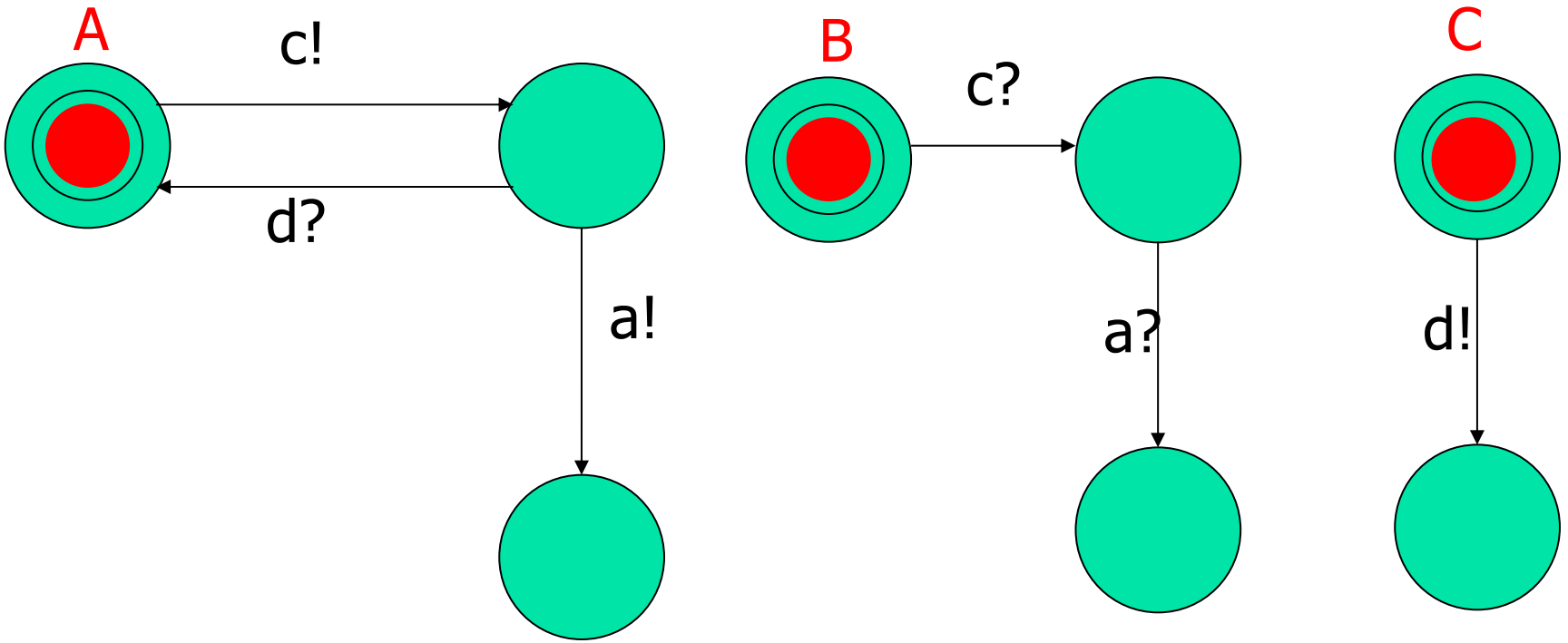
The static architecture of a concurrent system: A || B || C



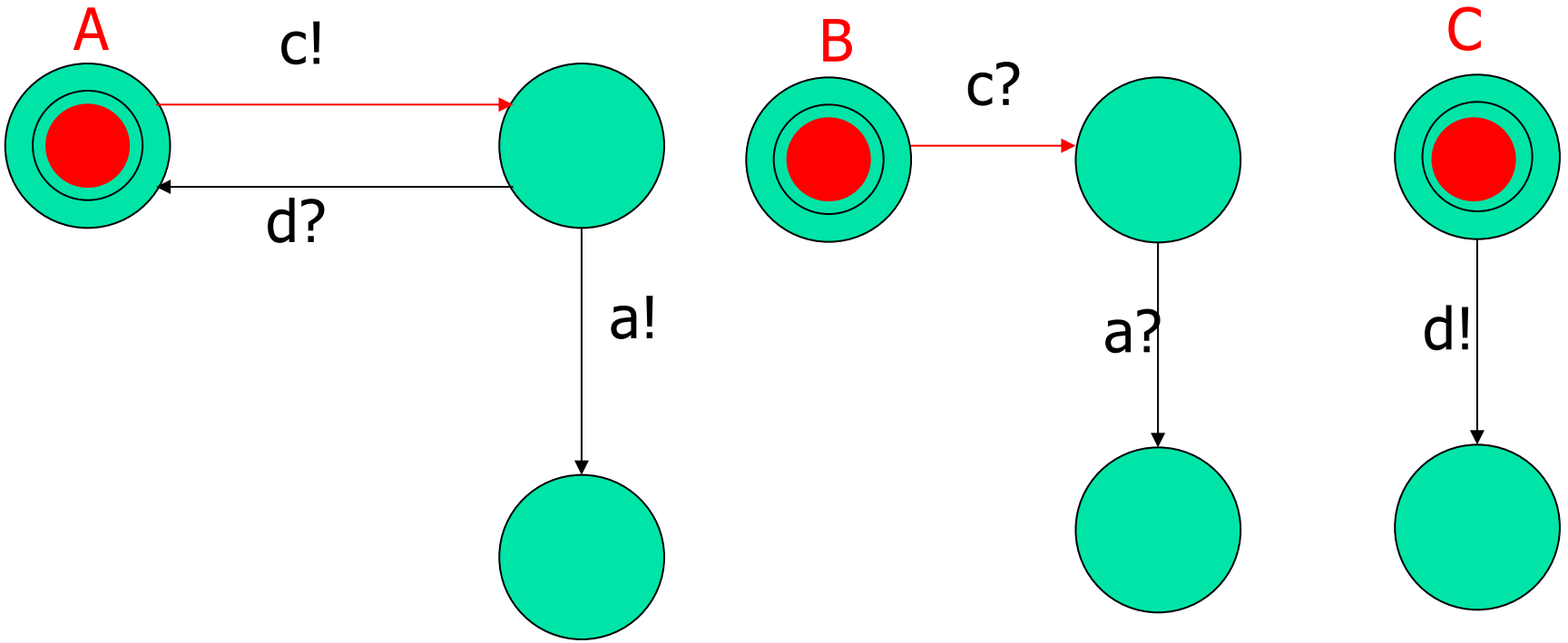
Networks of automata



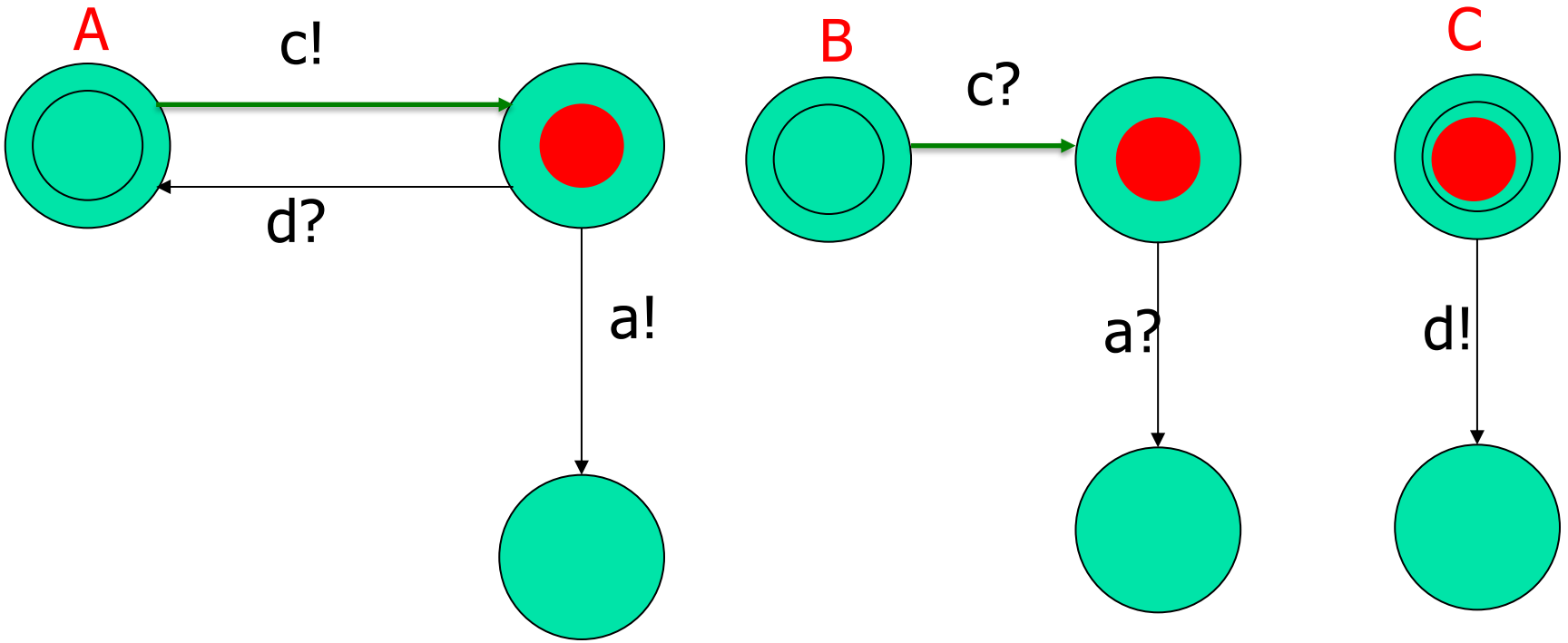
Networks of automata: initial state



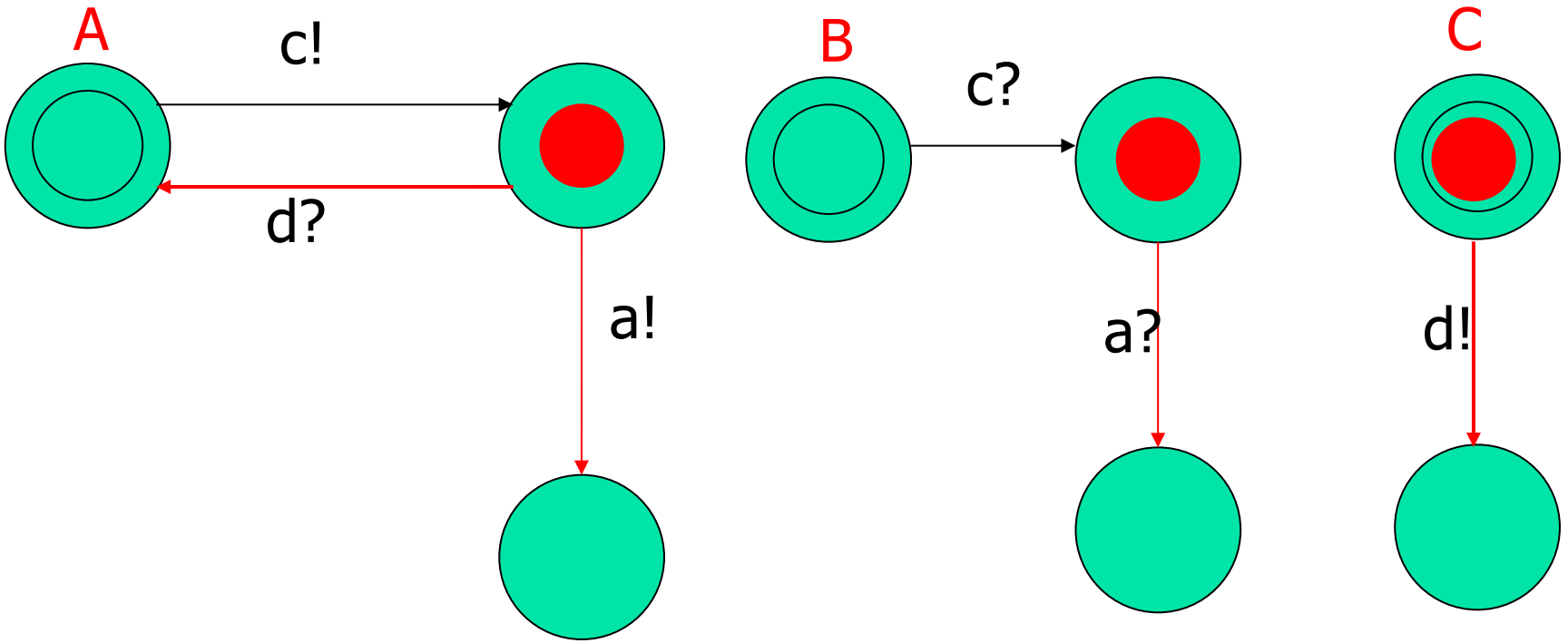
Networks of automata: enabled transition



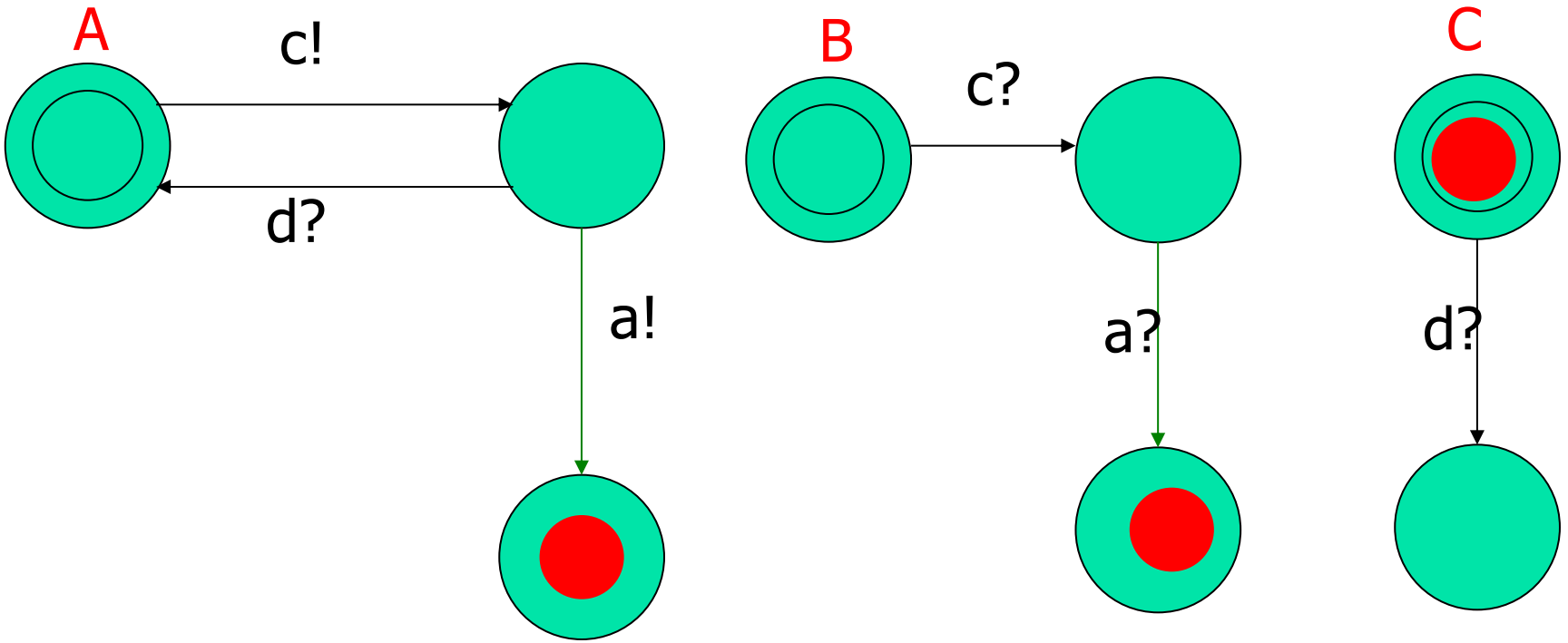
Networks of automata: transition taken



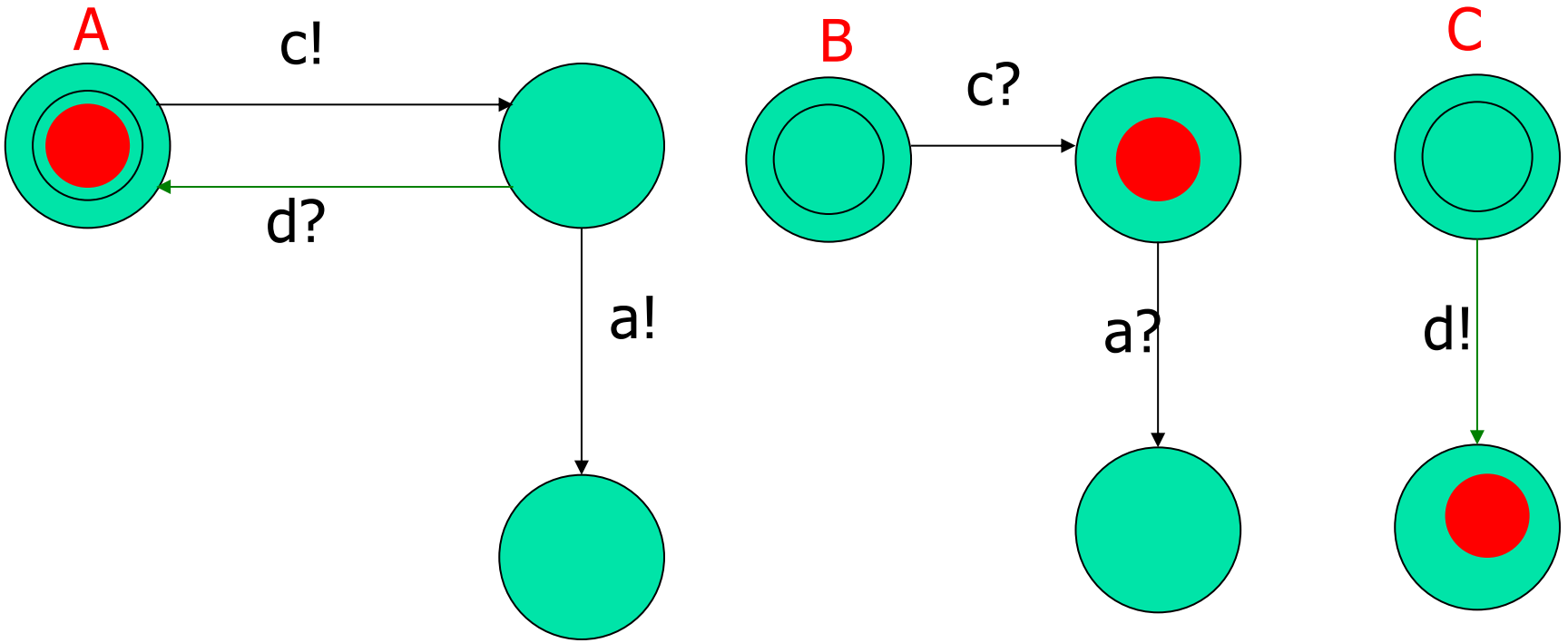
Networks of automata:
nondeterministic choices (transitions)



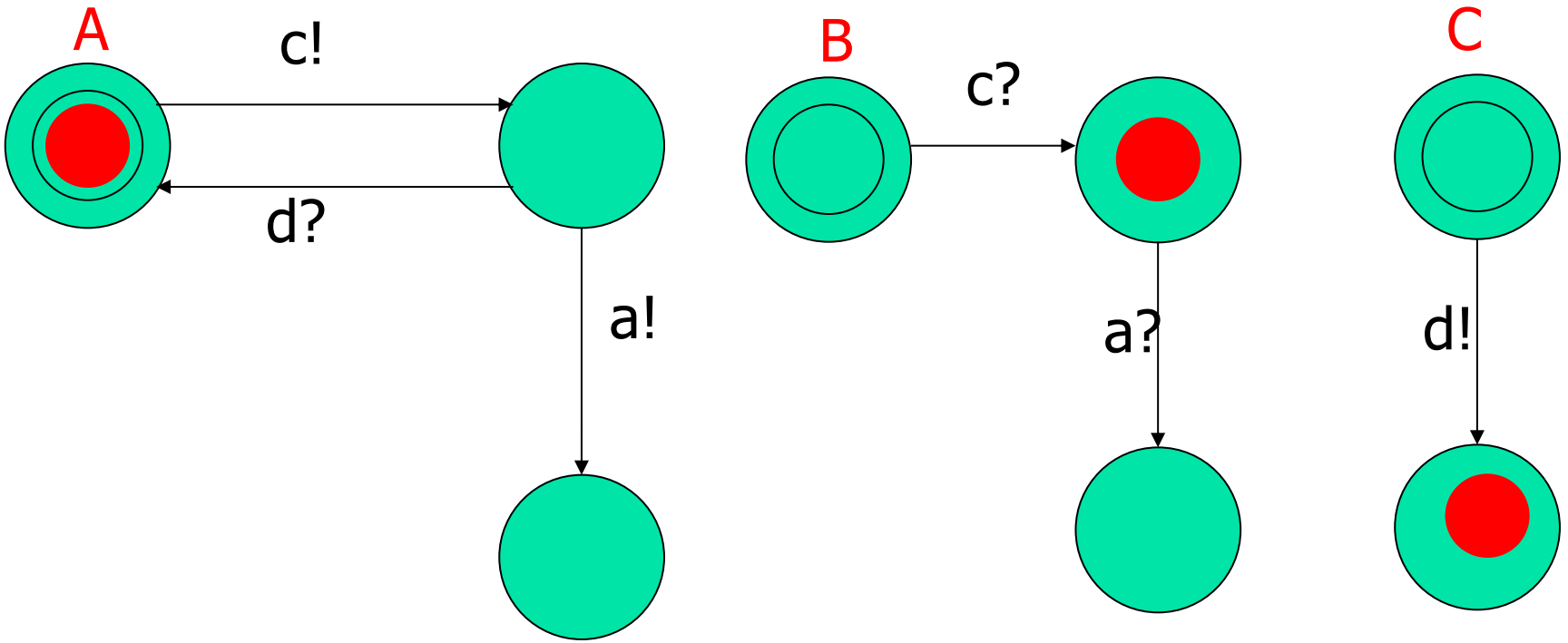
Networks of automata: non-deterministic transitions (1)



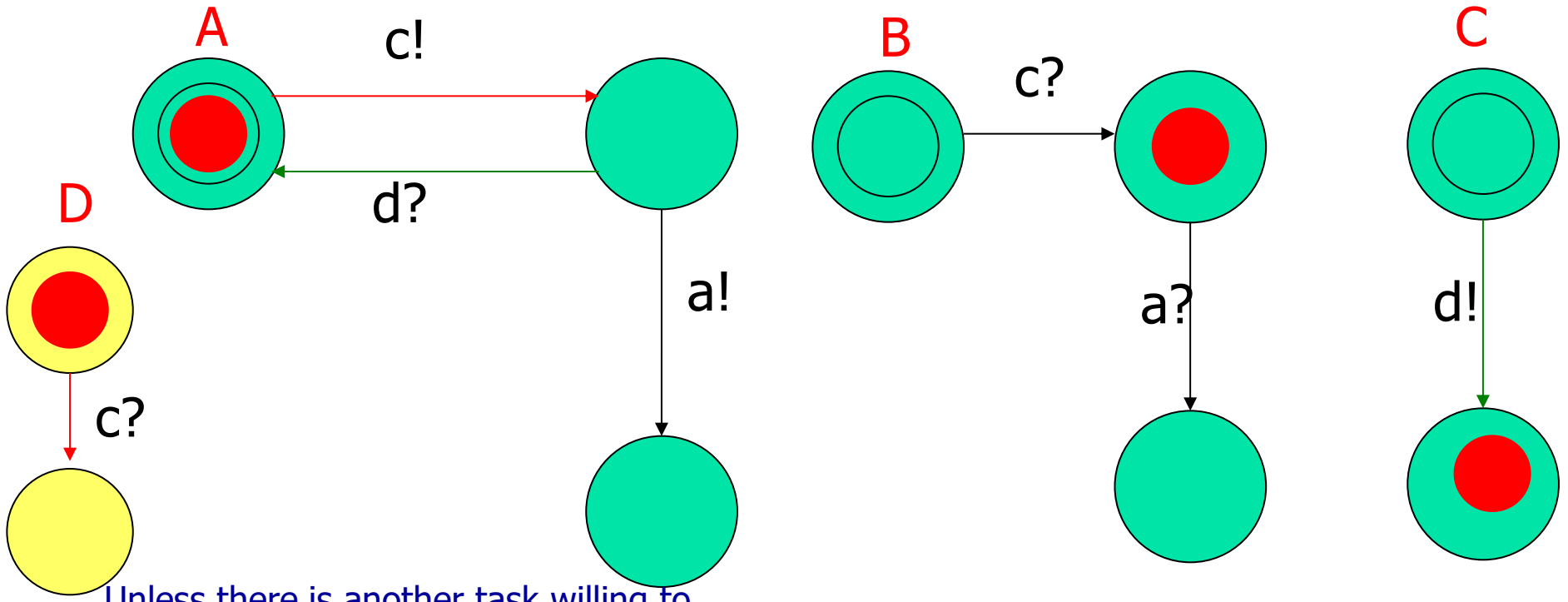
Networks of automata: non-deterministic transition (2)



Networks of automata: deadlock!



Networks of automata: deadlock!

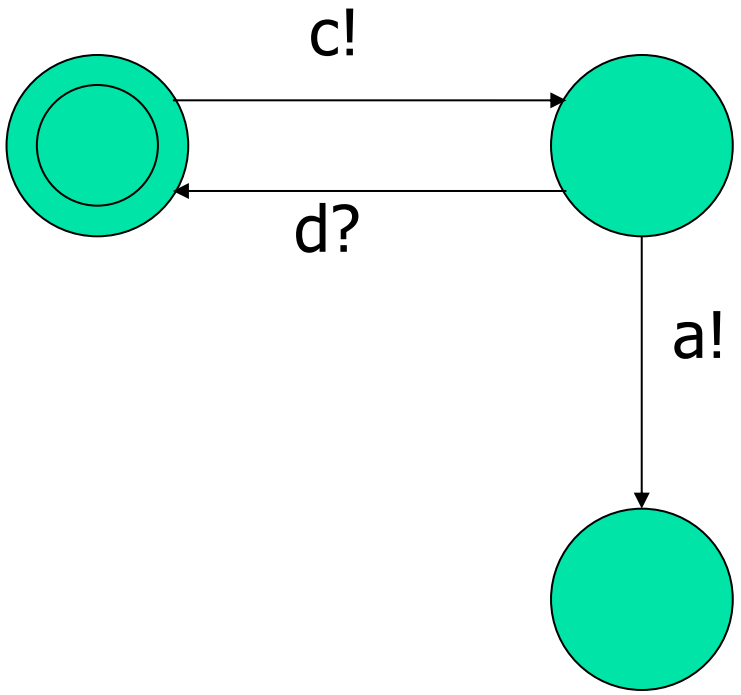


Unless there is another task willing to synchronize on any of the actions

Timed Automata:

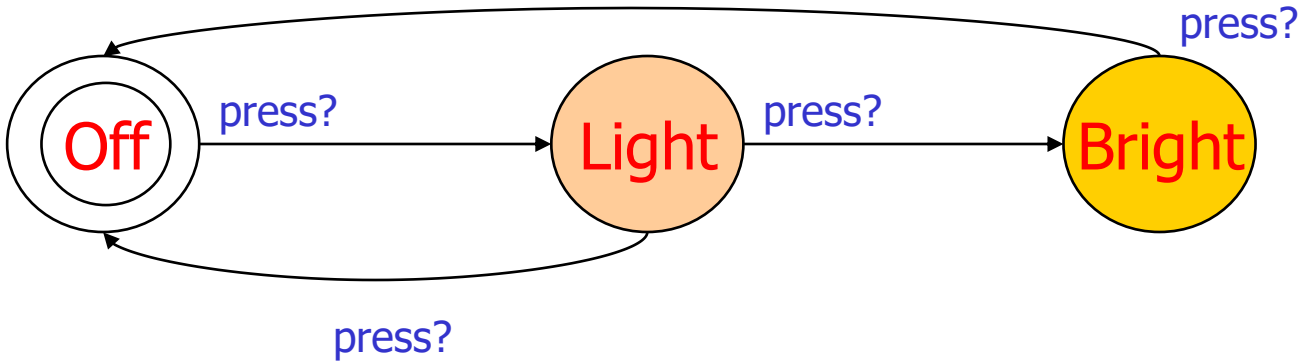
State Machines with Timing Information

Finite Automata



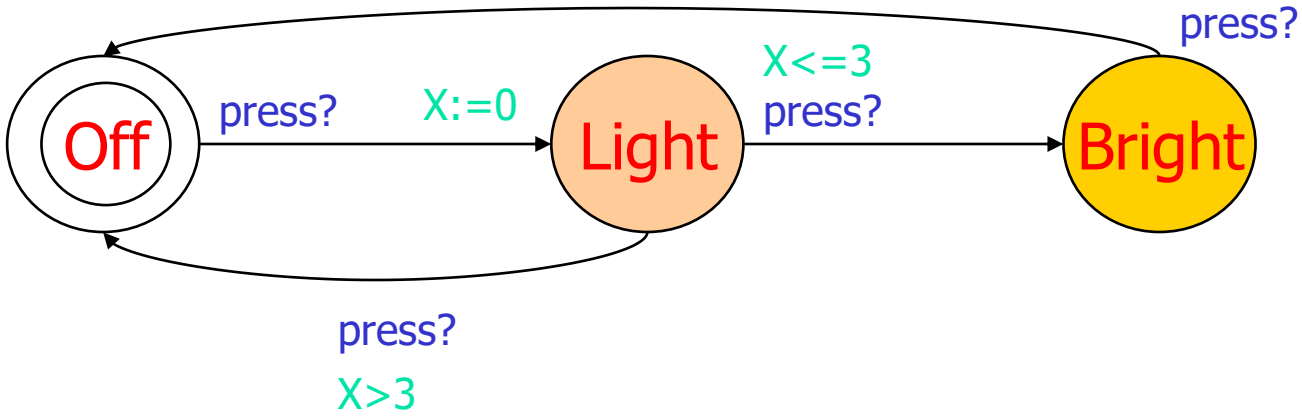
No information about when the transitions take place!
Let's add data types: Clock and Integer

Intelligent Light Control



WANT: if press is issued twice quickly
then the light will get brighter; otherwise the light is turned off.

Intelligent Light Control (with timer)

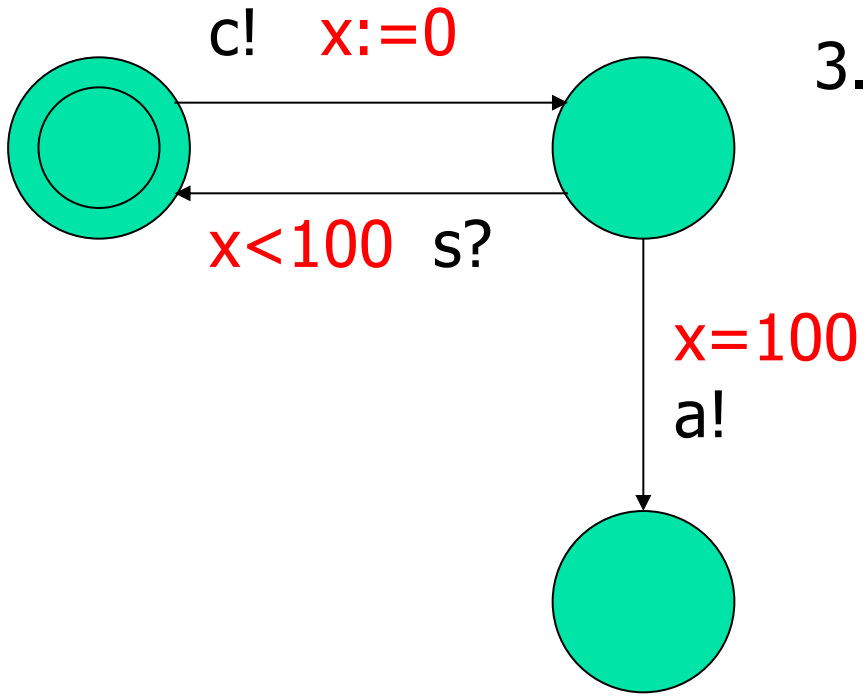


Solution: Add real-valued clock x

Clocks and timing constraints

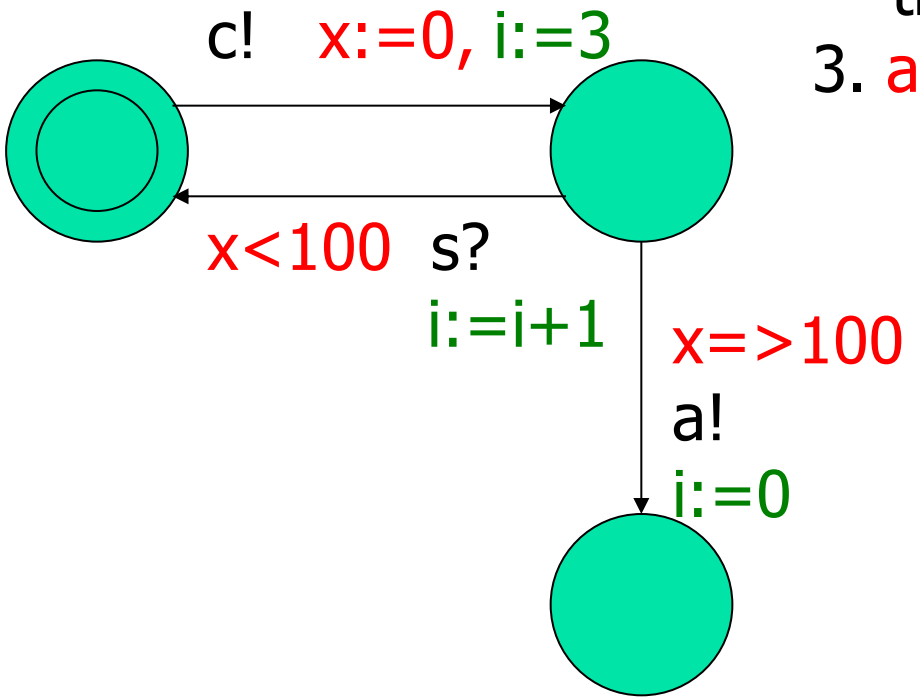
- Now we assume that the system has a finite number of clocks
  - The clocks start to increase from 0 when the system starts, and they run at the same rate
  - The clocks can be tested using e.g. $x<100$
  - The clocks can be reset to 0 on a transition using an assignment: $x:=0$

Timed automata = Finite Automata with **clock type**



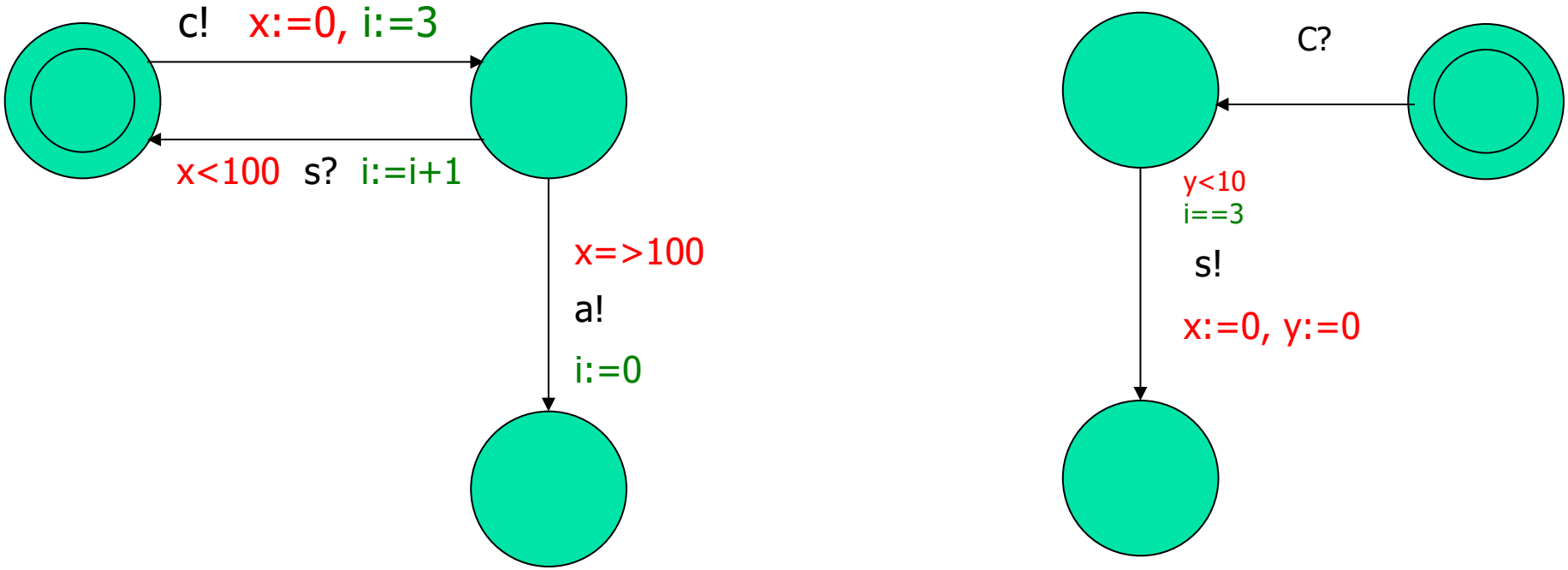
1. connect and reset x to 0
2. If succeed within 100ms, then go to initial
3. abort after 100ms !

Timed automata with **Integers**

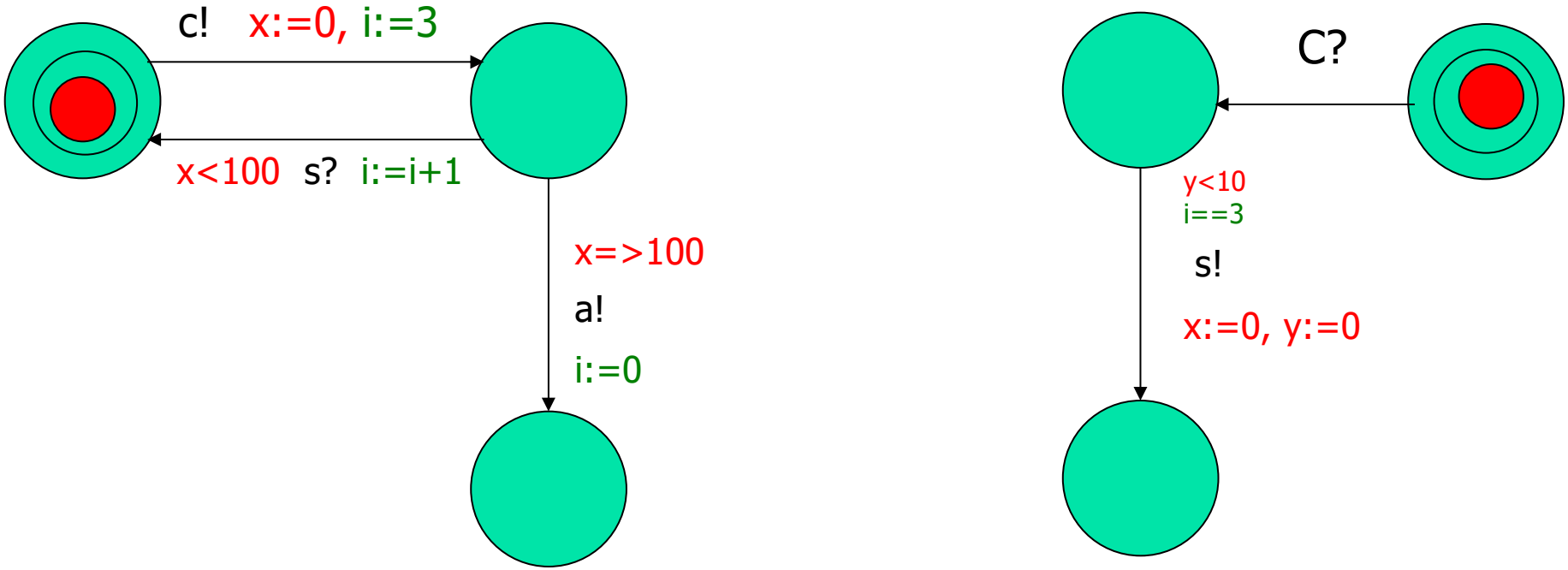


1. connect and reset x to 0
2. If succeed within 100ms, then go to initial
3. abort after 100 ms !

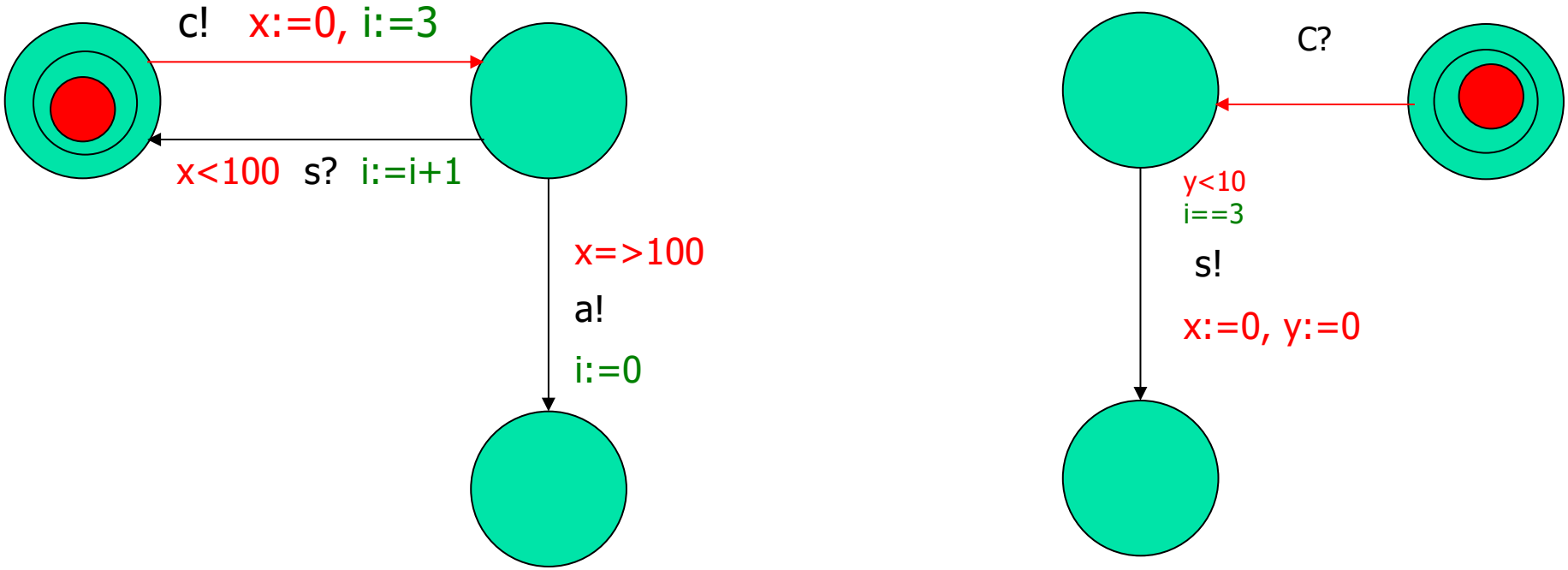
Network of Timed automata



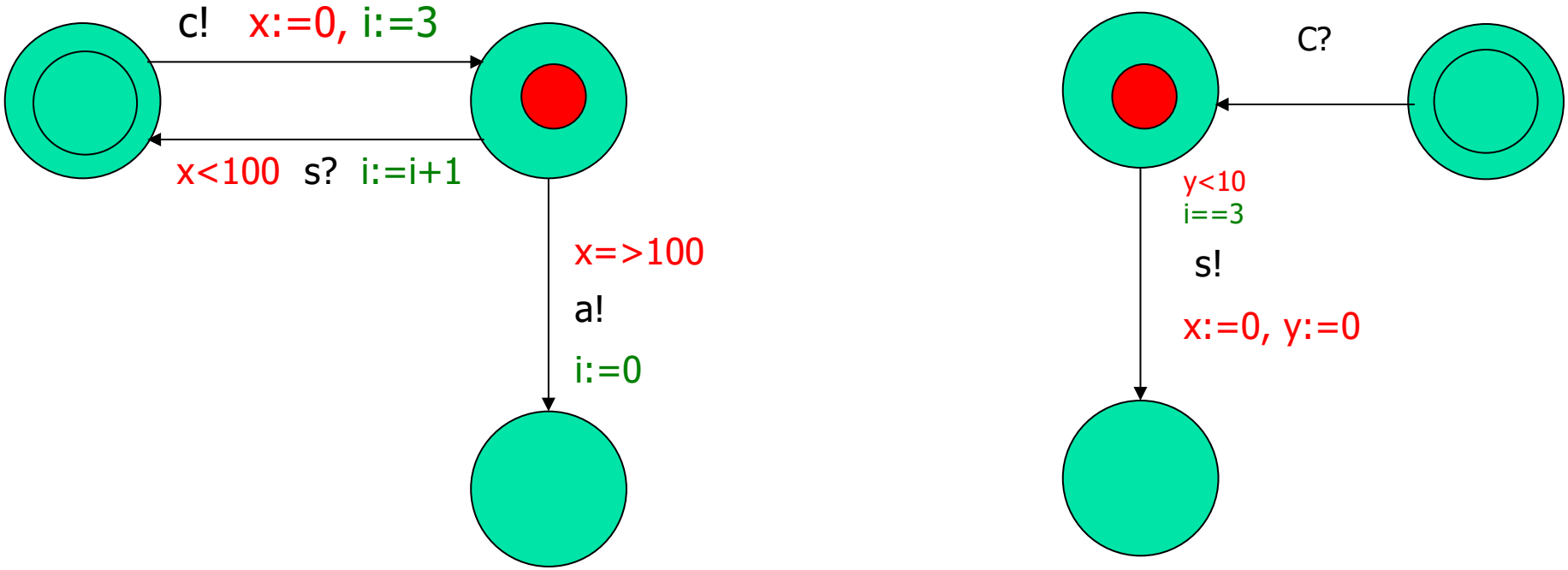
Network of Timed automata: initial states



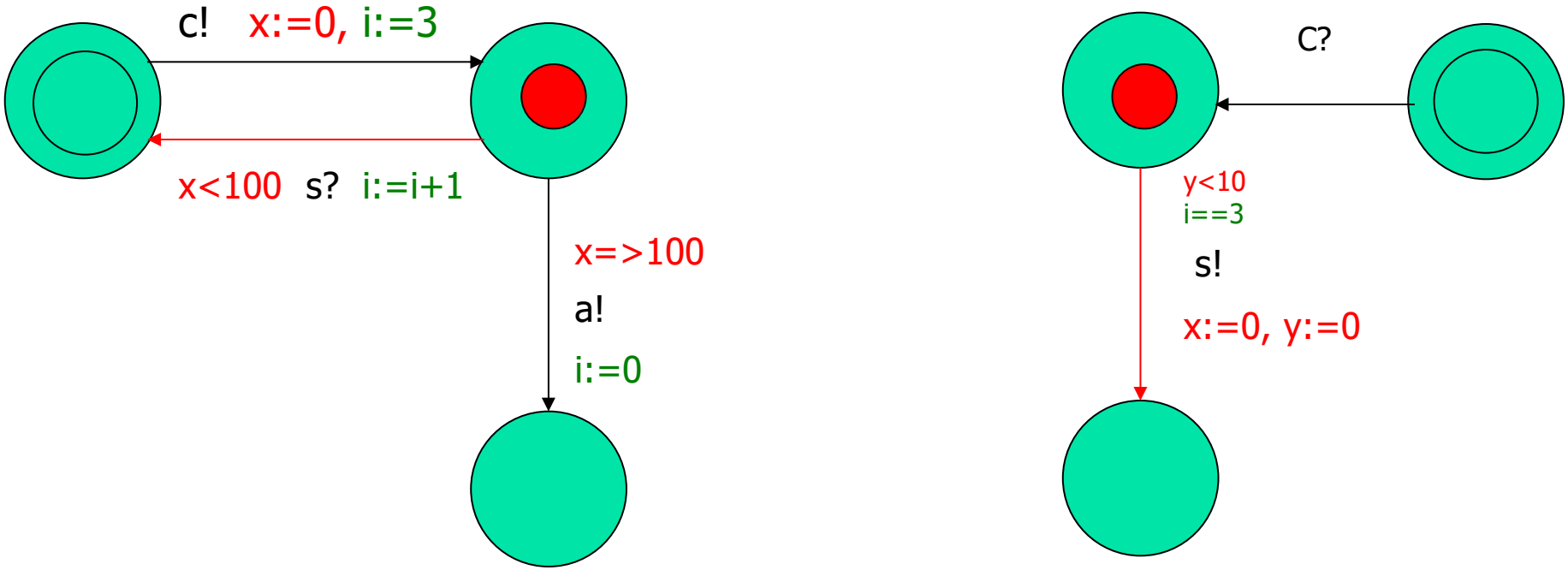
Network of Timed automata: enabled transition



Network of Timed automata: transition taken

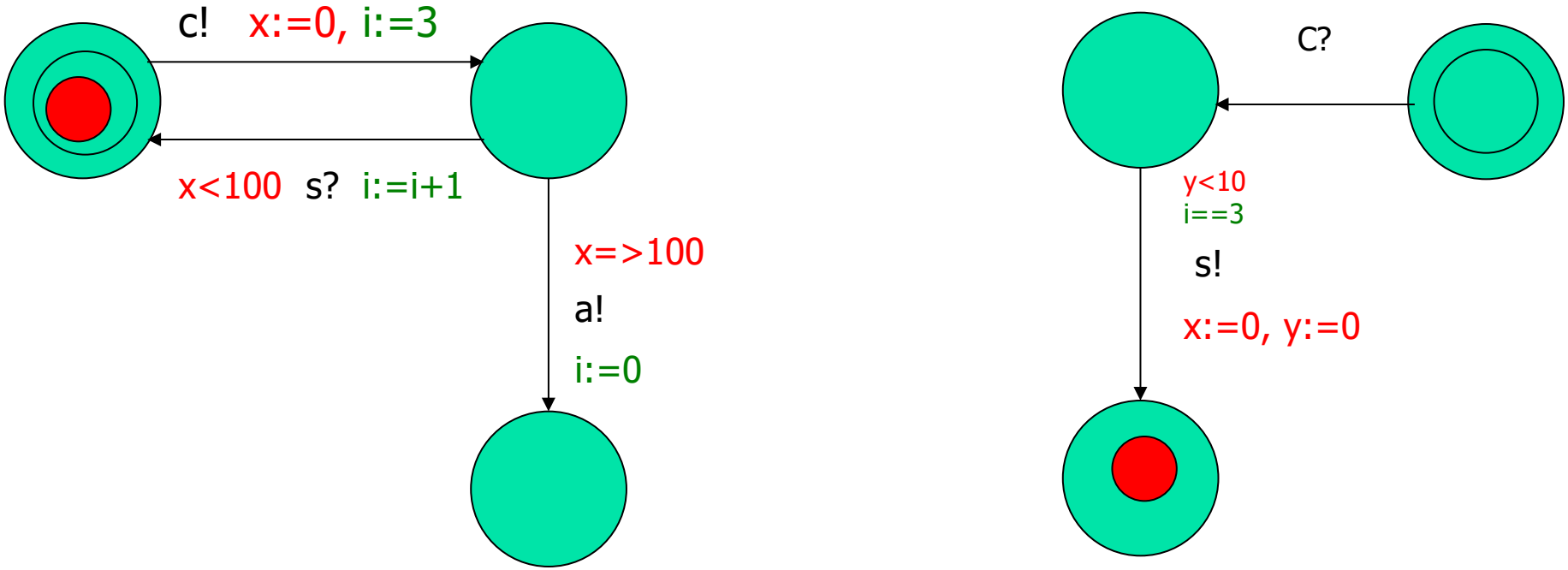


Network of Timed automata: enabled ransition



This depends on the values of x, y, and i

Network of Timed automata: transition taken



This depends on the values of x, y, and i

Timed automata (definition)

- a timed automaton is a finite graph
  - a finite many nodes N with an initial node
  - a finite many edges E between nodes
  - an edge may be labelled with three elements
    - guard
    - action (a?, a!, or nothing)
    - assignment

    (may be skipped if none)

Guard (i.e. Conditional Expression)

- a clock constraint
  - $g ::= x \leq n \mid x \geq n \mid x < n \mid x > n \mid g \wedge g$
  - where n is any natuaral number

- a predicate over data variables
  - "any logical expression" you may write in C

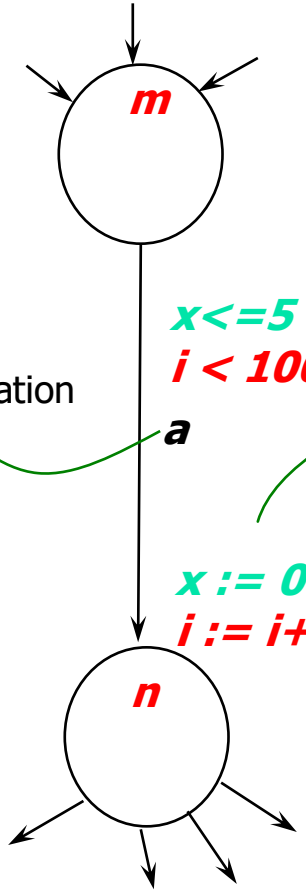
assignment

- a clock reset: x:=0 for any clock x

- a sequence of assignments in the form i:=e

  (or most of the programming constructs in C)

Timed Automata: Syntax



Action
used
for synchronization

Clocks: *x, y*
Data variable: *i*

Guard: clock constraint
or predicate on variables

Reset/assigment
on clocks and/or data variables

Nodes/locations: *m, n*

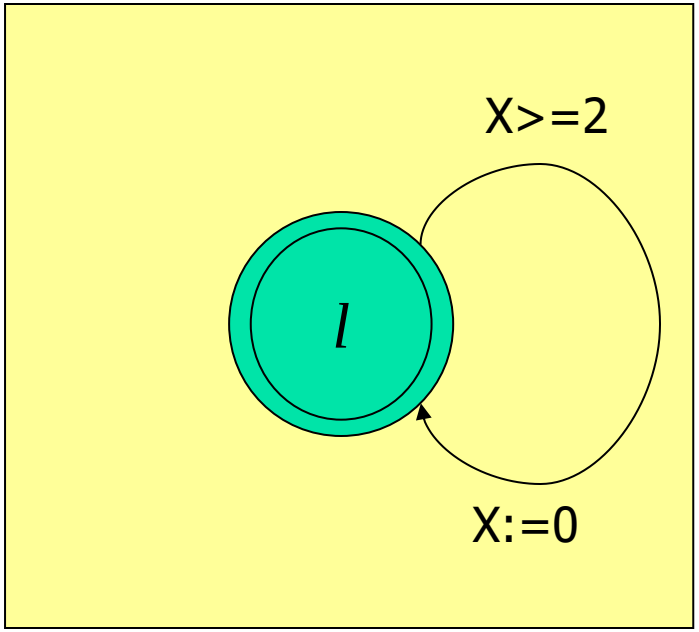
Edges/transitions: *m → n*

a

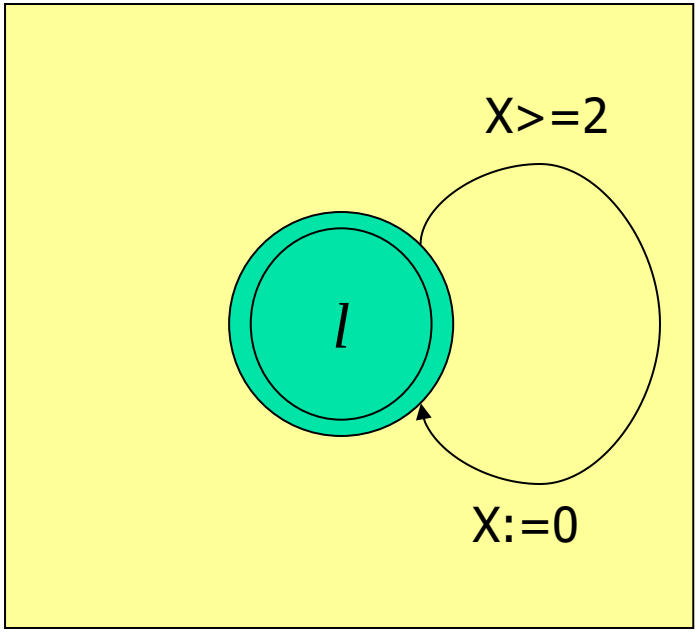
Initial node is a double circle
Initial values of all clocks and variables are 0

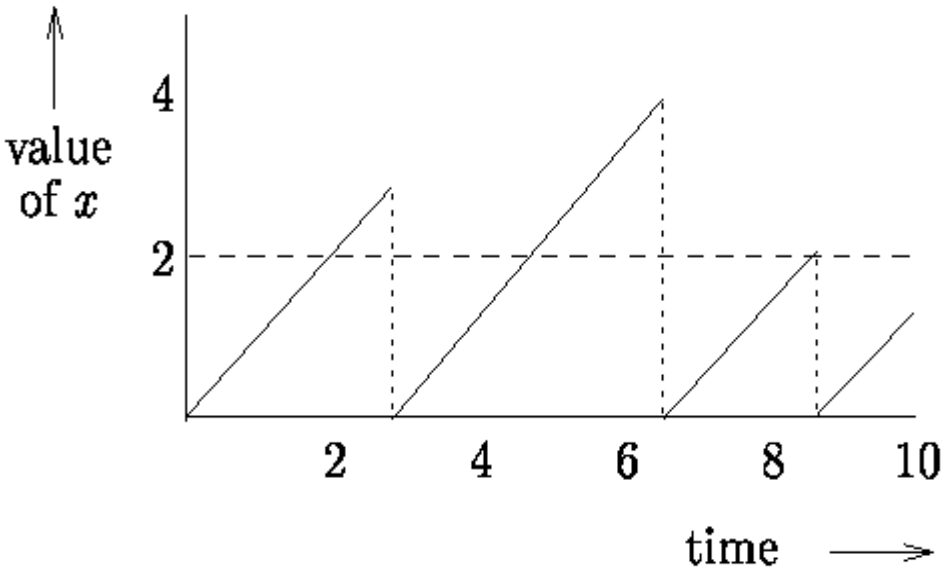
Examples of Timed Automata

Timed Automata: Example

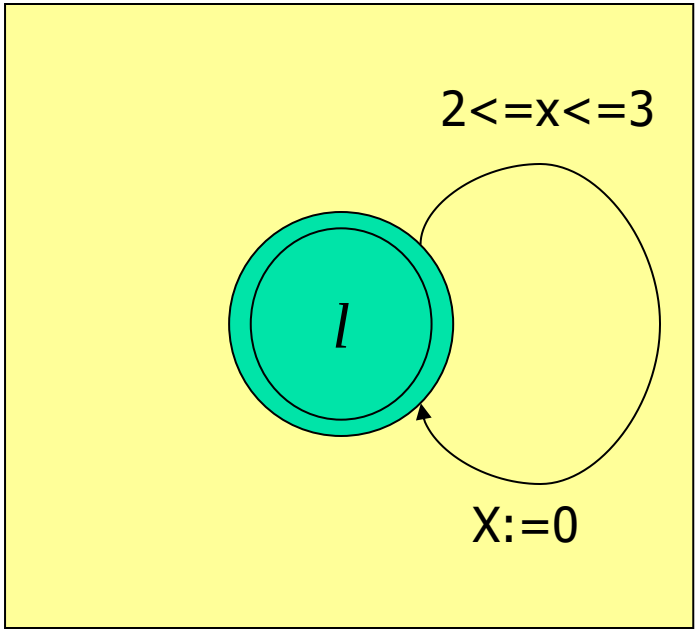


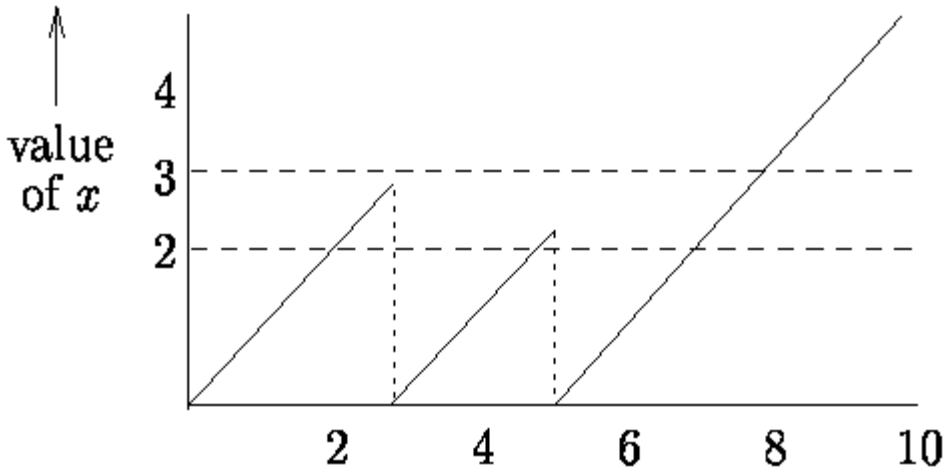
Timed Automata: Example



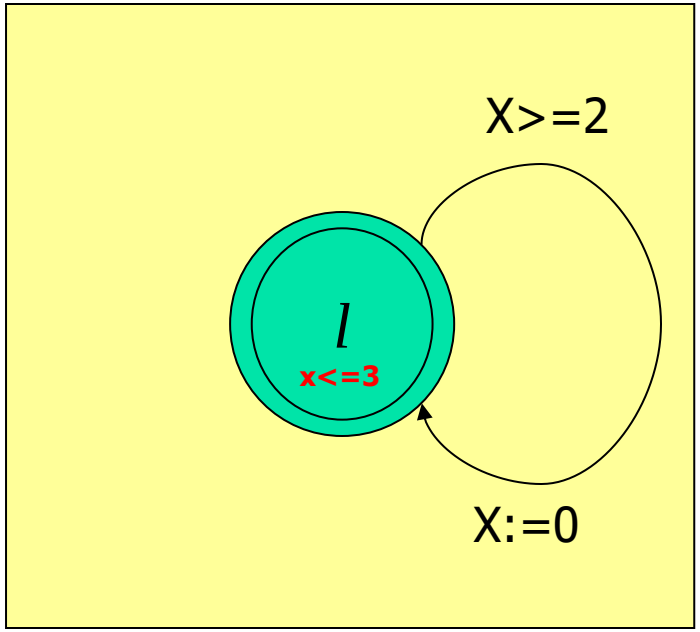


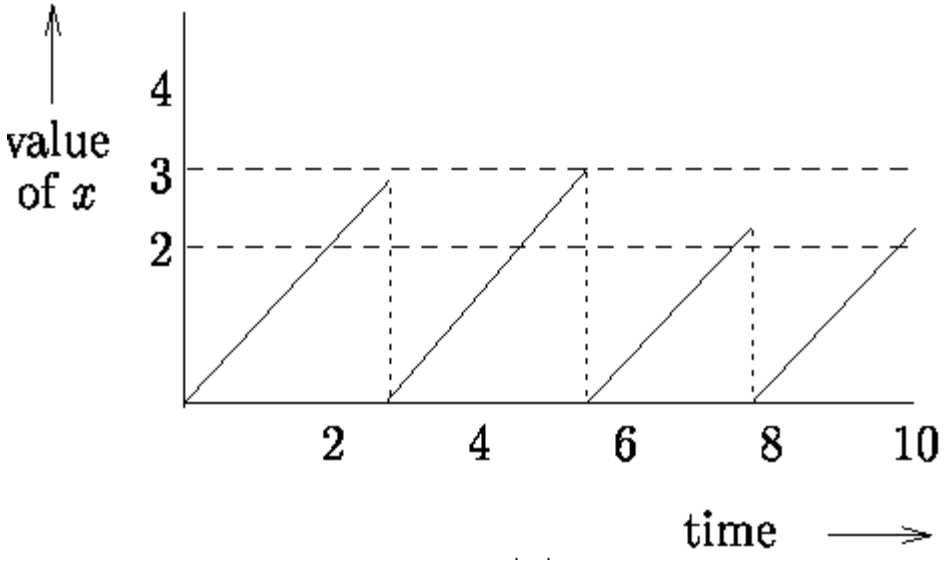
Timed Automata: Example





Timed Automata: Example

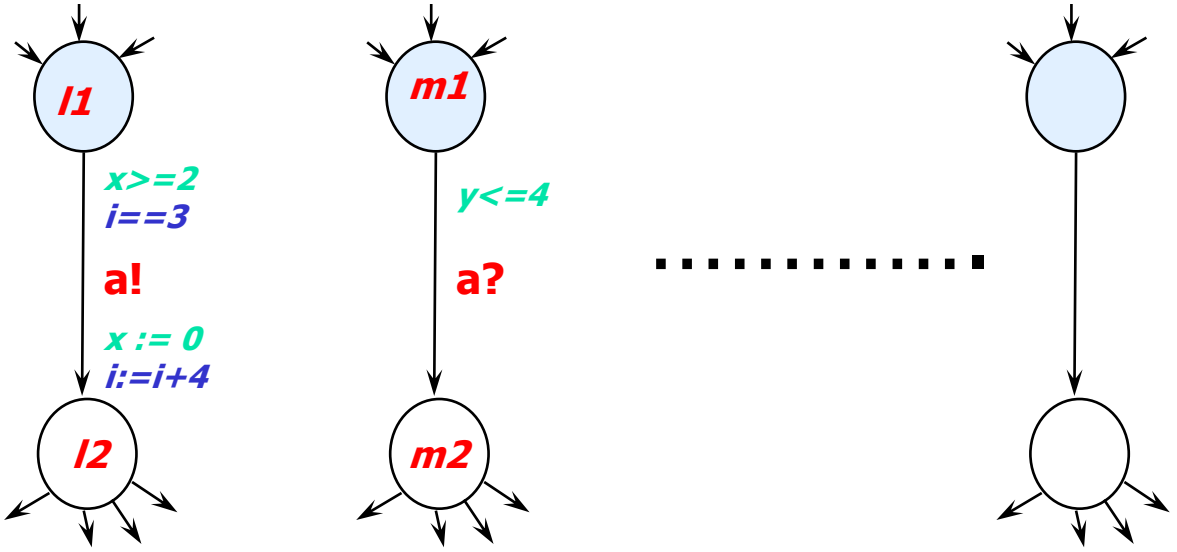




Modelling of Real-Time Systems

- Real Time
  - Clocks
- Synchronization
  - Rendezvous
  - Shared variables
- Concurrency
  - Networks of timed automata

Networks of Timed Automata



Two-way synchronization on *complementary* actions

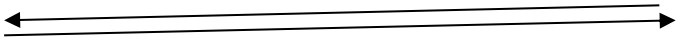
Example transitions

(***l1***, ***m1***,……, x=2, y=3.5, i=3,…..) ⟶ (***l2***,***m2***,……..,x=0, y=3.5, i=7,…..)

Modeling Ada Programs

Rendezvous

```
task body A is
begin
...
B.Call;
...
end A
```

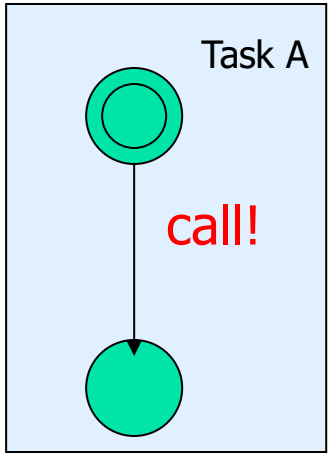


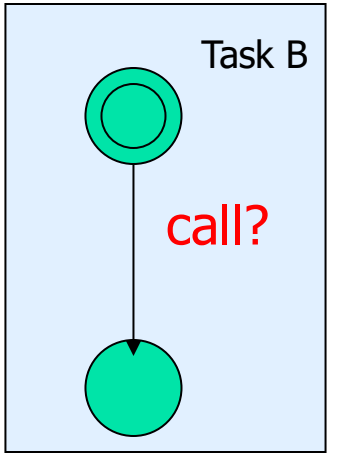
```
task body B is
begin
...
accept Call do
....
end Call
...
end A
```

Rendezvous

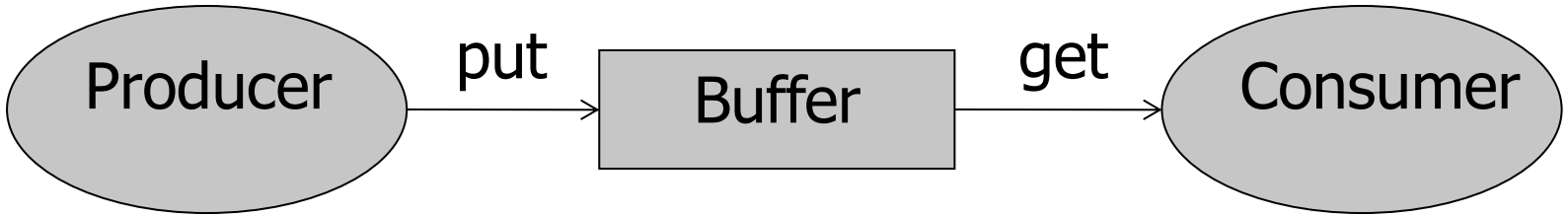
```
task body A is
begin
...
B.Call;
...
end A
```

```
task body B is
begin
...
accept Call do
....
end Call
...
end A
```





Producer, Buffer and Consumer



? ? ?

Buffer

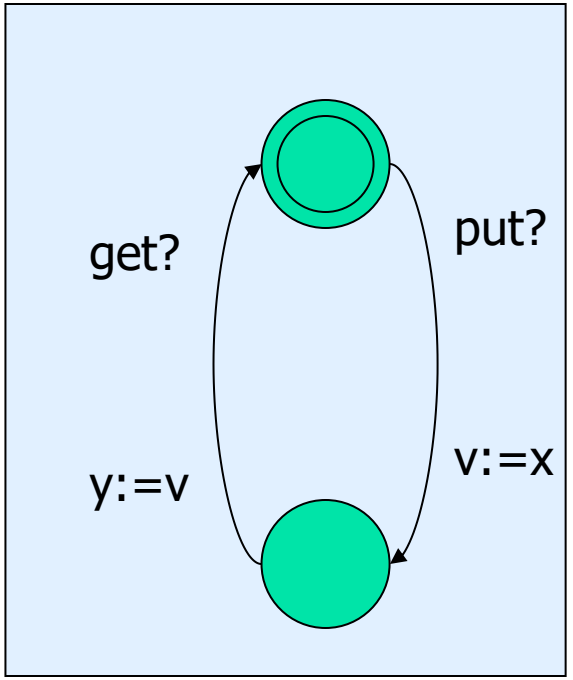
```
task buffer is
entry put(X: in integer)
entry get(x: out integer)
end;

task body buffer is
  v: integer;
begin
loop accept put(x: in integer) do v:= x end put;
       accpet get(x: out integer) do x:= v end get;
end loop;
end buffer;
---
buffer.put(...)      ←---------- other tasks (users)!!
buffer get(...)
----
```

Buffer

```
task buffer is
entry put(x: in integer)
entry get(x: out integer)
end;

task body buffer is
 v: integer;
begin
loop accept put(x: in integer); v:= x end put;
       accpet get(y: out integer); y:= v end get;
end loop;
end buffer;
```



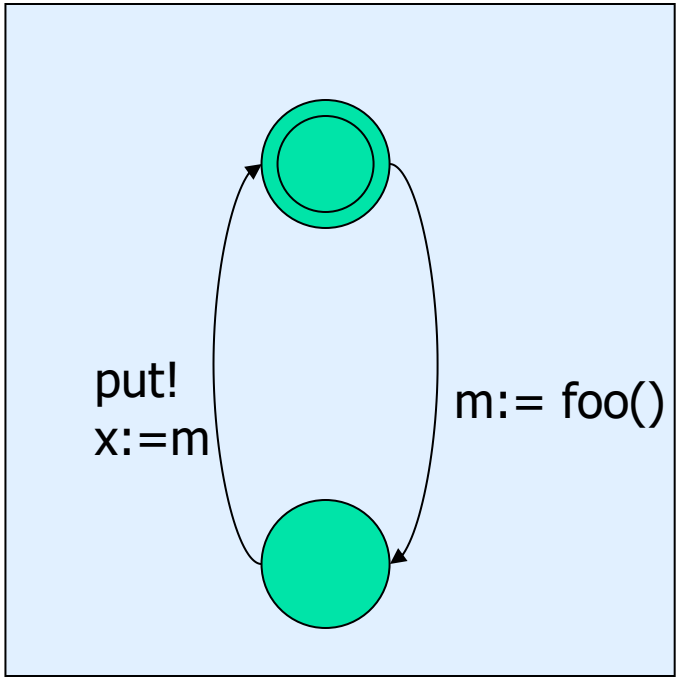
---

buffer.put(...) ←---------- other tasks (users)!!

buffer get(...)

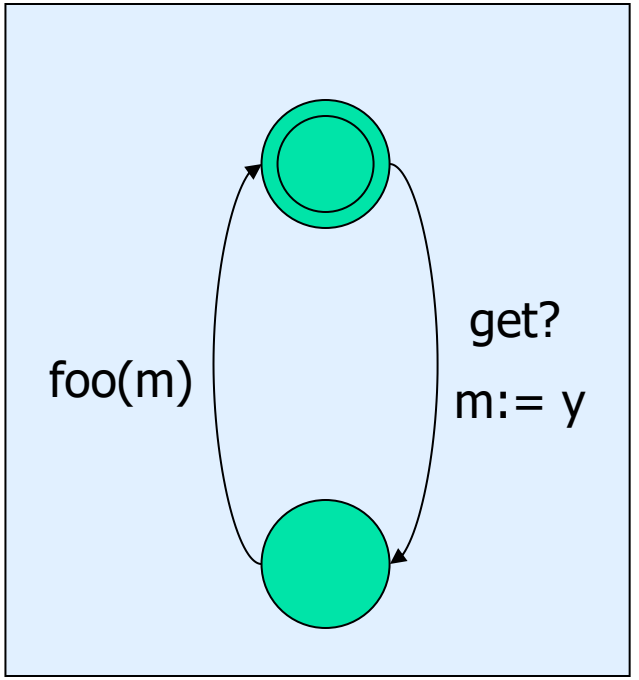
Producer

```
task Producer;
task body Producer is
  m: integer;
 begin
    loop
    m := foo() -- produce a new message;
    Buffer.put(m);
    end loop;
end Producer;
```

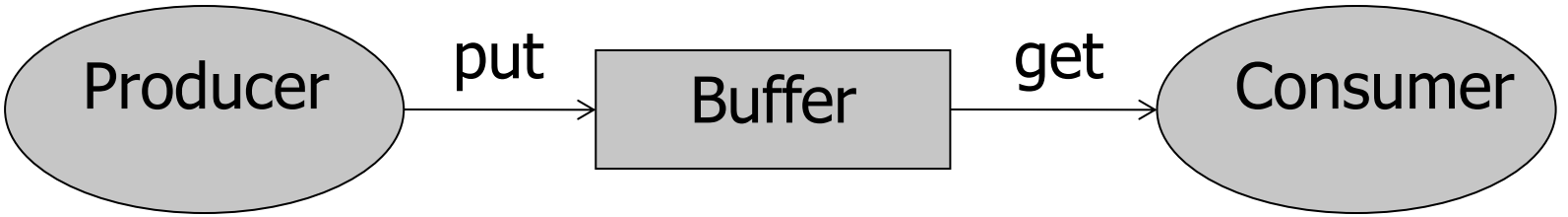


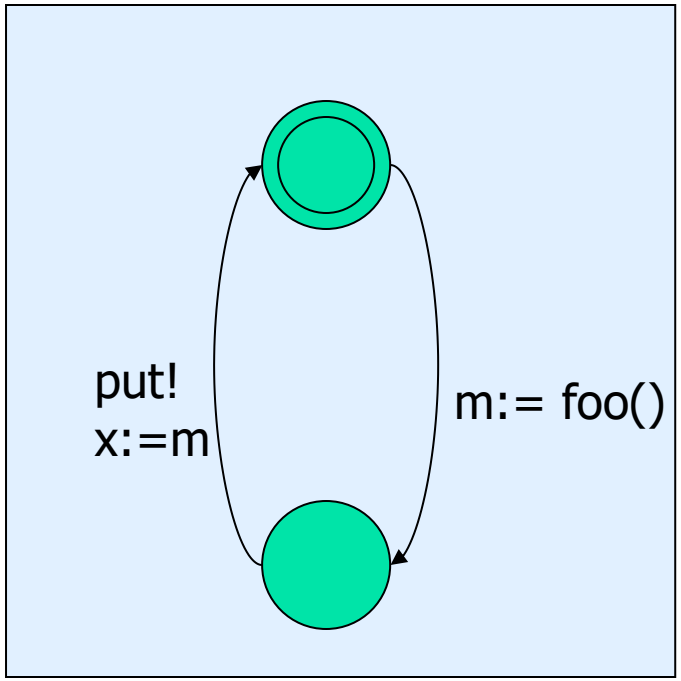
Consumer

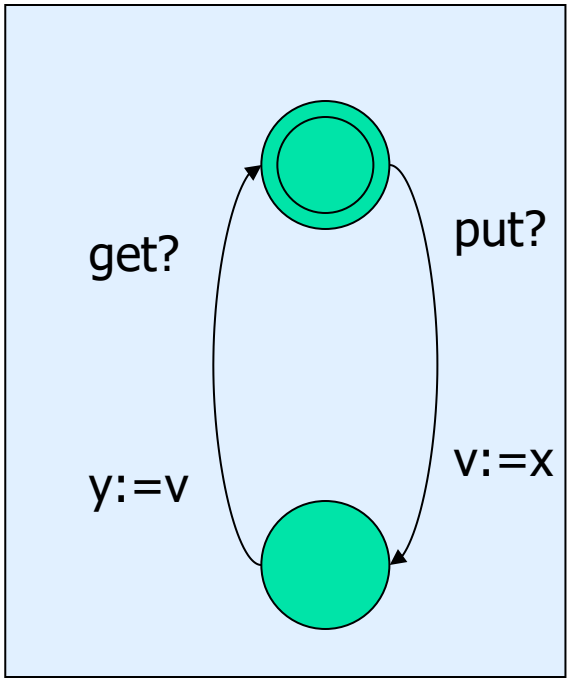
```
task Consumer;
task body Consumer is
  m: integer;
 begin
    loop
    Buffer.get(m);       -- get a new message;
    foo(m);              -- do something with m;
    end loop;
end Consumer;
```

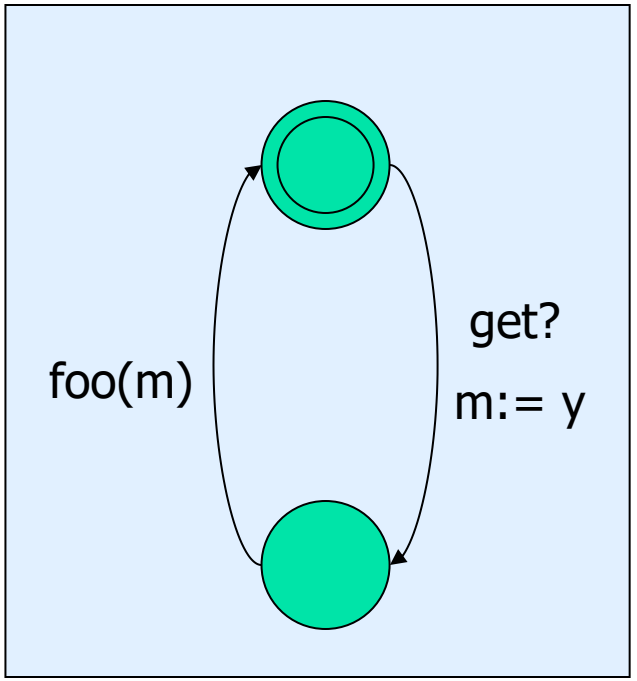


Producer, Buffer and Consumer







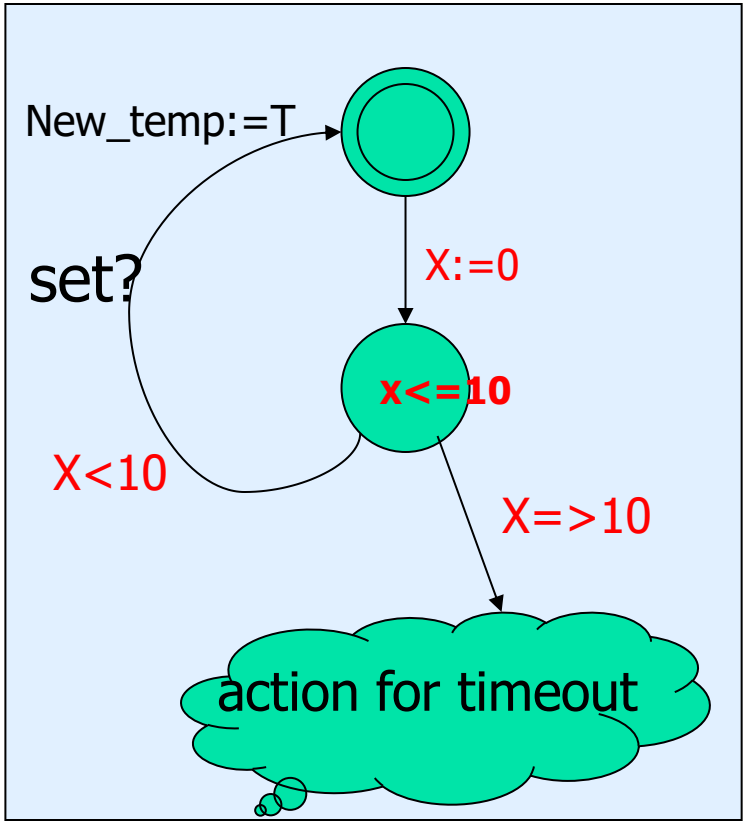


Timeout and message passing

```
loop
    select
        accept set(T : temperature) do
                New_temp:=T;
        end Call;
    or
        delay 10.0;
                --action for timeout
    end select;
    --other actions
end loop;
```

Timeout and message passing

```
loop
    select
        accept set(T : temperature) do
                New_temp:=T;
        end Call;
    or
        delay 10.0;
                --action for timeout
    end select;
    end loop;
```



Periodic Activity

```
task body T is
   Interval : constant Duration := 5.0;
   Next_Time : Time;
begin
   Next_Time := Clock + Interval;
   loop
      Action;
      delay until Next_Time;
      Next_Time := Next_Time + Interval;
   end loop;
end T;
```

Periodic Activity

```
task body TaskP is
   T : constant Duration := 5.0;
   Next_Time : Time;
begin
   Next_Time := Clock + T;
   loop
      Action;
      delay until Next_Time;
      Next_Time := Next_Time + T;
   end loop;
end TaskP;
```