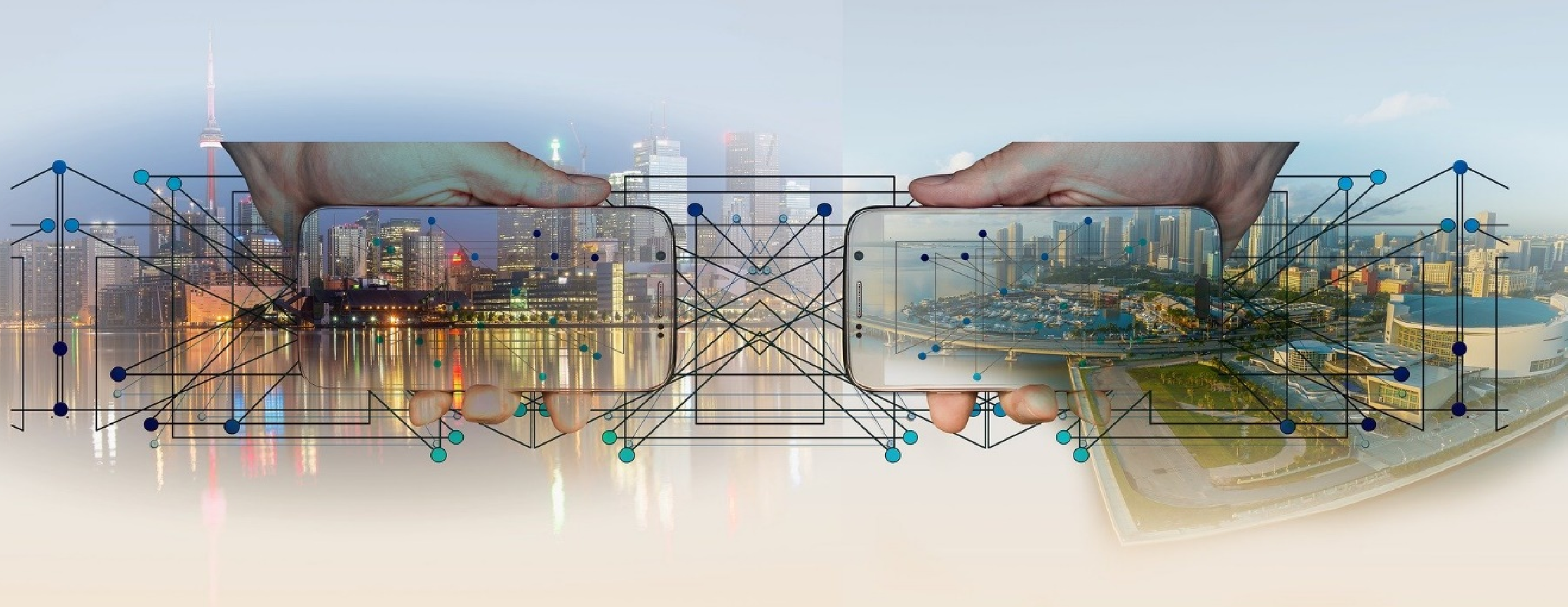
Lecture 9

Program Correctness



Program correctness

The State-transition model

The set of global **states** =

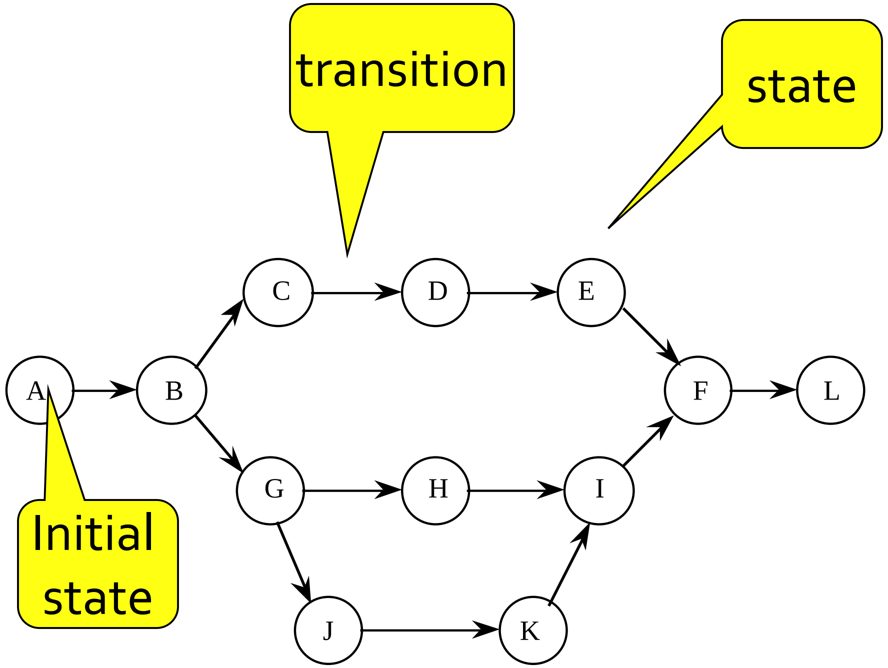
$s_0 \times s_1 \times \cdots \times s_m$

$\{s_k$ is the set of local states of process k$\}$

$S0 \xrightarrow{action} S1 \xrightarrow{action} S2 \xrightarrow{action}$

Each **transition** is caused by an action of an eligible process.

We reason using **interleaving semantics**



Correctness criteria

- Safety properties
  - Bad things never happen
- Liveness properties
  - Good things eventually happen

Example 1: Mutual Exclusion

<u>*Process 0*</u>

do true →

    Entry protocol

    Critical section

    Exit protocol

od

<u>*Process 1*</u>

do true →

    Entry protocol

    Critical section

    Exit protocol

od

Safety properties

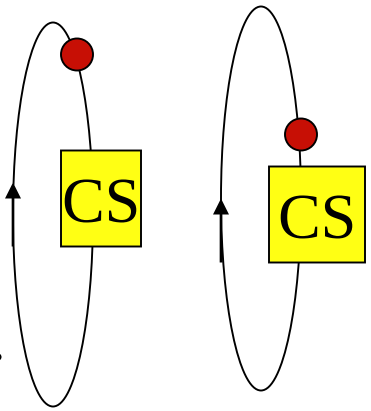
(1) There is no deadlock

(2) At most one process enters the critical section.

Liveness property

A process trying to enter the CS must eventually succeed.

(This is also called the *progress property*)



Testing vs. Proof

- Testing: Apply inputs and observe if the outputs satisfy the specifications. Fool proof testing can be painfully slow, even for small systems. Most testing are partial.

- Proof: Has a mathematical foundation, and a complete guarantee. Sometimes not scalable.

Correctness proofs

- Since testing is not a feasible way of demonstrating the correctness of program in a distributed system, we will use some form of mathematical reasoning as follows:
  - Assertional reasoning of proving safety properties
  - Use of well-founded sets of proving liveness properties
  - Programming logic
  - Predicate transformers

Review of Propositional Logic

- <u>Example</u>: Prove that $P \Rightarrow P \lor Q$

- Pure propositional logic is sometimes not adequate for proving the properties of a program, since propositions can not be related to program variables or program state. Yet, an extension of propositional logic, called *predicate logic*, will be used for proving the properties.

Review of Predicate Logic

- Predicate logic is an extension of propositional logic

  cf. A proposition is a statement that is either true or false.

- A predicate specifies the property of an object or a relationship among objects. A predicate is associated with a set, whose properties are often represented using the universal quantifier ____ (for all) and the existential quantifier _____(there exists).

<quantifier><bound variable(s)>:<range>::<property>

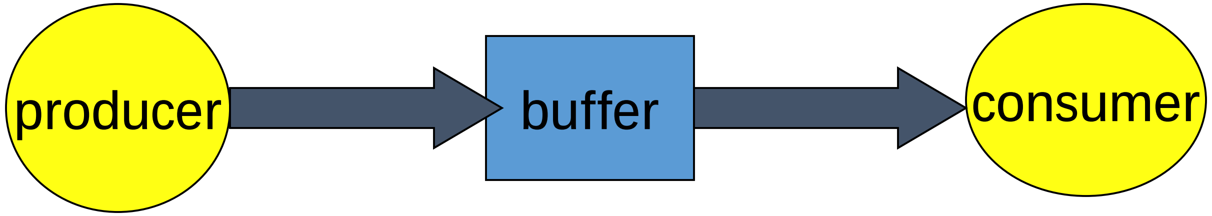
(ex) $\exists\, j: j \in N(i) :: c[j] = c[i] + 1 \bmod 3$

Examples of Safety invariant
Well-known synchronization problems

Invariant means: a logical condition which should always be true.

1. *The mutual exclusion problem*. **$N_{CS} \leq 1$**,
   where N_{CS} is the Total number of processes in CS at any time

2. *Producer-consumer problem*. **$0 \leq N_P - N_C \leq$ buffer capacity**
   (N_P = no. of items produced, N_C = no. of items consumed)



Exercise

What can be a safety invariant for the readers and writers problem?

- **Only one write can write to the file at a time.**
- **When a writer write to the file, no process can read.**
- **Many processes can read at the same time.**

Let N_W denote the number of writer processes updating the file and N_R denote the number of reader processes reading the file.

➔ $((N_W = 1) \land (N_R=0)) \lor ((N_W =0) \land (N_R \geq 0))$

Assertional reasoning of proving safety properties (1)

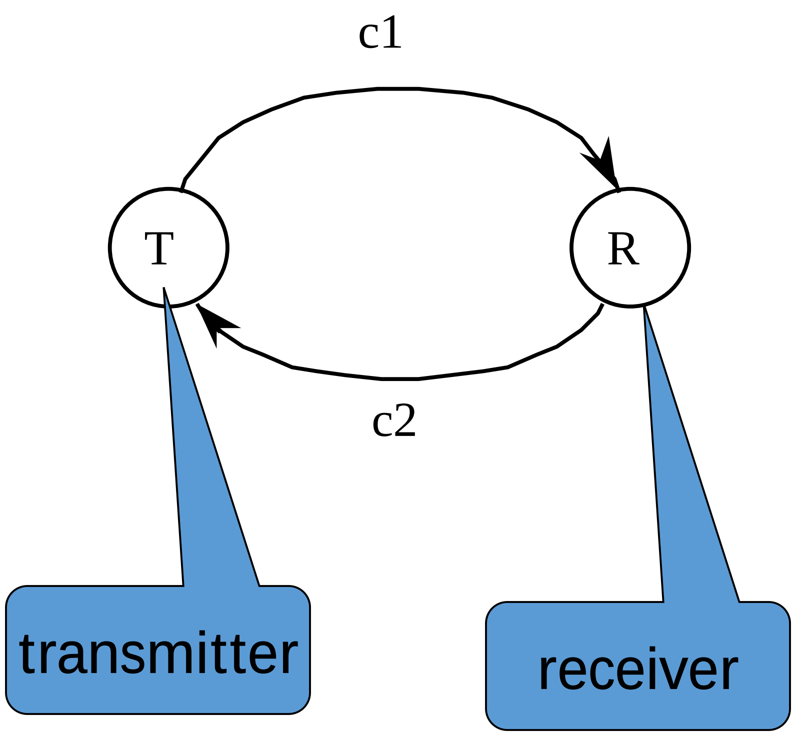
```
define   c1, c2 : channel; {init c1 = Φ, c2 = Φ}
         r, t : integer; {init r = 5, t = 5}

{program for T}
1    do   t > 0→           send msg along c1; t := t -1
2    □    ¬empty (c2) →    rcv msg from c2; t := t + 1
     od

{program for R}
3    do   ¬empty (c1) →    rcv msg from c1; r := r+1
4    □    r > 0       →    send msg along c2; r := r-1
     od
```

We want to prove the safety property P:

$P \equiv n1 + n2 \leq 10$

n1= # of messages in c1

n2= # of messages in c2

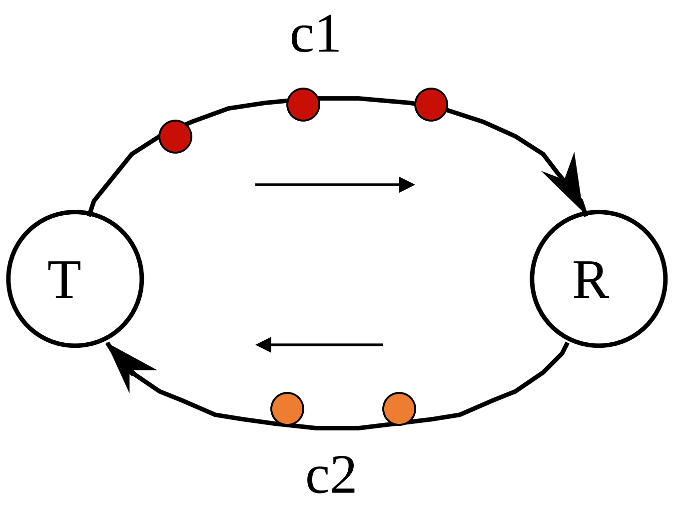


Assertional reasoning of proving safety properties (2)

n1, n2 = # of msg in c1and c2 respectively.
We will establish the following invariant:

$I \equiv (t \geq 0) \wedge (r \geq 0) \wedge (n1 + t + n2 + r = 10)$

(I implies P). Check if I holds after every action.



{program for T}

```
1    do   t > 0→      send msg along c1; t := t -1
2    □  ¬empty (c2) →  rcv msg from c2; t := t+1
     od
```

{program for R}

```
3    do  ¬empty (c1) →  rcv msg from c1; r := r+1
4    □      r > 0      → send msg along c2; r := r-1
     od
```

Use the method of induction

Liveness properties

- Eventuality is tricky. There is no need to guarantee *when* the desired thing will happen, as long as it happens.

Type of Liveness Properties

Progress Properties

- ♦ If the process want to enter its critical section, it will eventually do.
- ♦ No deadlock?

Reachability Properties

: The question is that S_t is reachable from S_0?

- ♦ The message will eventually reach the receiver.
- ♦ The faulty process will be eventually be diagnosed

Fairness Properties

: The question is if an action will eventually be scheduled.

Termination Properties

- ♦ The program will eventually terminate.

Proving liveness

Use of well-founded sets of proving liveness properties

Global state

Global state

S1 → S2 → S3 → S4 →

↓f ↓f ↓f ↓f

w1 w2 w3 w4

- o w1, w2, w3, w4 ∊ WF
- o WF is a well-founded set whose elements can be ordered by]

f is called a measure function

If there is no infinite chain like

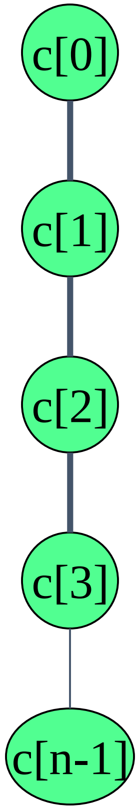
w1] w2] w3] w4 ..., *i.e.*

If an action changes the system state from s1 to s2

$f(s_i)$] $f(s_{i+1})$] $f(s_{i+2})$ ...

then the computation will definitely terminate!

Proof of liveness: an example



Clock phase synchronization

System of n clocks ticking at the same rate.

Each clock is 3-valued, i,e it ticks as 0, 1, 2, 0, 1, 2...

A failure may arbitrarily alter the clock phases.

The clocks need to return to the same phase.

Proof of liveness: an example

Clock phase synchronization

{Program for each clock}

(c[k] = phase of clock k, initially arbitrary)

do $\exists$ j: j $\in$ N(i) :: c[j] = c[i] +1 mod 3

→ c[i] := c[i] + 2 mod 3

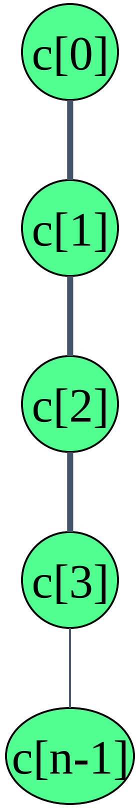
□ $\forall$ j: j $\in$ N(i) :: c[j] ≠ c[i] +1 mod 3

→ c[i] := c[i] + 1 mod 3

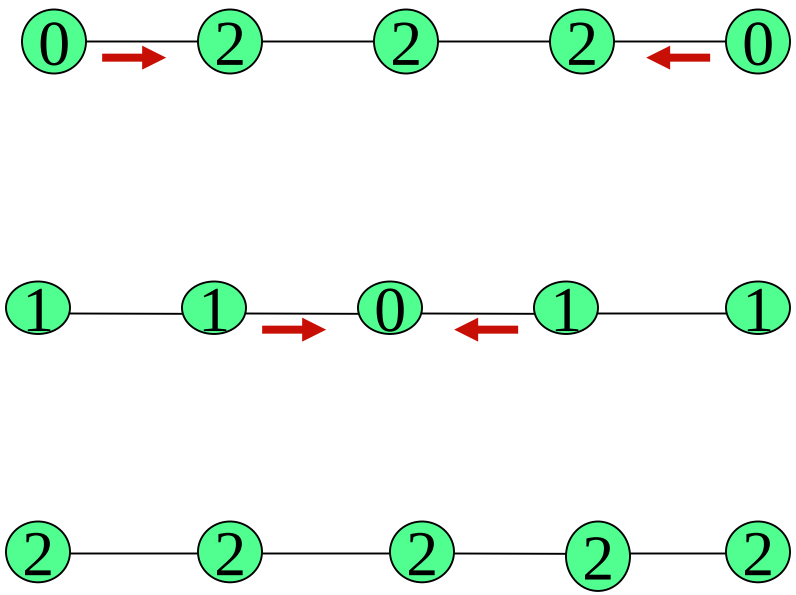
od

Show that eventually all clocks will return to the same phase (convergence), and continue to be in the same phase (closure)

c[k] $\in$ {0,1,2}



Proof of convergence



Understand the game of arrows

Let **D** = d[0] + d[1] + d[2] + ... + d[n-1]

d[i] = 0 if no arrow points towards clock **i**;
= i + 1 if a ← pointing towards clock **i**;
= n – i if a → pointing towards clock **i**;
= 1 if both ← and → point towards clock **i**.

By definition, D ≥ 0.

Also, D decreases after every step in the system. So the number of arrows must reduce to 0.

D= 0 means all the clocks are synchronized.