The State-transition model

The set of global states =
\[ s_0 \times s_1 \times \ldots \times s_m \]
\{s_k \text{ is the set of local states of process } k\}

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**

```plaintext
Process 0
  do true →
    Entry protocol
    Critical section
    Exit protocol
  od

Process 1
  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**)

---

*CS*
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.
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

Example: 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 \textit{predicate logic}, will be used for proving the properties.
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).

\(<\text{quantifier}>\langle\text{bound variable(s)}\rangle:\langle\text{range}\rangle::\langle\text{property}\rangle\>

(ex) \(\exists j: j \in N(i) :: c[j] = c[i] + 1 \mod 3\)
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 \text{buffer capacity}
   (N_p = \text{no. of items produced, } N_c = \text{no. of items consumed})
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.

$$
((N_w = 1) \land (N_r = 0)) \lor ((N_w = 0) \land (N_r \geq 0))
$$
define \ c_1, \ c_2 \ : \ \text{channel}; \ \{\text{init} \ c_1 = \Phi, \ c_2 = \Phi\}\\
r, t : \text{integer}; \ \{\text{init} \ r = 5, \ t = 5\}\\

\{\text{program for T}\}\{\text{program for R}\}\n1 \quad \text{do} \quad t > 0 \rightarrow \text{send msg along } c_1; \ t := t - 1\\
2 \quad \square \quad \neg \text{empty} \ (c_2) \rightarrow \text{rcv msg from } c_2; \ t := t + 1\\
\quad \text{od}\\
3 \quad \text{do} \quad \neg \text{empty} \ (c_1) \rightarrow \text{rcv msg from } c_1; \ r := r+1\\
4 \quad \square \quad r > 0 \quad \rightarrow \quad \text{send msg along } c_2; \ r := r-1\\
\quad \text{od}\\

We want to prove \textit{the safety property} \( P \):\\
P \equiv n_1 + n_2 \leq 10
n1, n2 = # of msg in c1 and c2 respectively.
We will establish the following invariant:

\[ I \equiv (t \geq 0) \land (r \geq 0) \land (n1 + t + n2 + r = 10) \]  
(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
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 wants to enter its critical section, it will eventually do.
- No deadlock?

**Reachability Properties**
- The question is whether $S_t$ is reachable from $S_o$?
- The message will eventually reach the receiver.
- The faulty process will be eventually 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

If there is no infinite chain like

\[ w_1 \downarrow w_2 \downarrow w_3 \downarrow w_4 \ldots, i.e. \]

If an action changes the system state from \( s_1 \) to \( s_2 \)

\[ f(s_i) \downarrow f(s_{i+1}) \downarrow f(s_{i+2}) \ldots \]

then the computation will definitely terminate!

\( f \) is called a measure function

\( w_1, w_2, w_3, w_4 \in WF \)

WF is a well-founded set whose elements can be ordered by \( \downarrow \)
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.
Clock phase synchronization

{Program for each clock}
(c[k] = phase of clock k, initially arbitrary)

\[
\text{do } \exists j: j \in N(i) :: c[j] = c[i] + 1 \mod 3 \rightarrow \\
\qquad c[i] := c[i] + 2 \mod 3 \\
\forall j: j \in N(i) :: c[j] \neq c[i] + 1 \mod 3 \rightarrow \\
\qquad c[i] := c[i] + 1 \mod 3
\]

od

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


$d[i] = 0$ 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, $D \geq 0$.

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.