Program Correctness

View a distributed computation in terms of *states* and *state transitions*.

A *computation* (also called a behavior) = ABGHIFL

*History* (set of all behaviors) = {ABCDEFL, ABGHIFL, ABGJKIFL}

Note that history does not capture parallelism.
Testing vs. Proof

One possible proof technique involves enumerating all possible behaviors, and checking out each of them. Unfortunately, it is not scalable. Consider four processes A, B, C, D:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>b0</td>
<td>c0</td>
<td>d0</td>
</tr>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
<td>d2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
<td>d3</td>
</tr>
</tbody>
</table>

What is the total number of possible interleavings?

*The state explosion problem.*
Safety and Liveness properties

Safety property = Bad things *never* happen.
Liveness property = Good things *eventually* happen.

**Examples.** Consider the mutual exclusion problem.

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>do true ‖ do true ‖</td>
<td></td>
</tr>
<tr>
<td>Entry protocol Entry protocol</td>
<td></td>
</tr>
<tr>
<td><em>Critical section</em> <em>Critical section</em></td>
<td></td>
</tr>
<tr>
<td>Exit protocol Exit protocol</td>
<td></td>
</tr>
<tr>
<td>od od</td>
<td></td>
</tr>
</tbody>
</table>

**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 in doing so. This is also called the *progress property*. 
Exercise

Verify the liveness and safety properties in the sample program described below:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>define</strong> busy: shared Boolean;</td>
<td><strong>define</strong> busy: shared Boolean;</td>
</tr>
<tr>
<td><strong>initially</strong> busy = false;</td>
<td><strong>initially</strong> busy = false;</td>
</tr>
<tr>
<td><strong>do true ‖</strong></td>
<td><strong>do true ‖</strong></td>
</tr>
<tr>
<td><strong>do busy ‖ skip od;</strong></td>
<td><strong>do busy ‖ skip od;</strong></td>
</tr>
<tr>
<td>busy:= true;</td>
<td>busy:= true;</td>
</tr>
<tr>
<td><em>critical section</em>;</td>
<td><em>critical section</em></td>
</tr>
<tr>
<td>busy := false;</td>
<td>busy := false</td>
</tr>
<tr>
<td>{remaining codes}</td>
<td>{remaining codes}</td>
</tr>
<tr>
<td><strong>od</strong></td>
<td><strong>od</strong></td>
</tr>
</tbody>
</table>
Safety invariants

Most safety properties can be specified by an invariant (implies that some predicate must always hold).

{Mutex}

The number of processes in the CS ≤ 1

{Bounded capacity channel}

0 ≤ nP - nC ≤ channel capacity

{Absence of deadlock}

□(G0 G1 G2 ... Gk)□ postcondition

{Partial Correctness} If program terminates, then the result will be correct. It does not determine if the program will terminate. (Termination is a liveness property).

Total correctness = partial correctness + termination.
Example of partial correctness

{
\begin{center}
\begin{tikzpicture}
\node (P0) at (0,0) {P0};
\node (P1) at (1,1) {P1};
\node (P2) at (2,1) {P2};
\node (P3) at (1,-2) {P3};
\draw (P0) -- (P1);
\draw (P0) -- (P2);
\draw (P0) -- (P3);
\end{tikzpicture}
\end{center}

\color{c \in \{0, 1, 2, 3\}}

\textbf{program} \textbf{colorme} \{\text{for process } P_i \}

\textbf{do } \forall j : j \in N(i) :: (c[i] = c[j]) \quad

\quad c[i] := (c[i]+2) \mod 4

\textbf{od}

Is the program partially correct?

Does it terminate?
Liveness Properties

Eventuality can be tricky. There is no need to guarantee *when* the desired thing will happen, as long as it happens. It is hard to demonstrate its falsehood from finite number of observations.

**Examples**

The message will eventually reach the receiver.

The process will eventually enter its critical section.

The faulty process will be eventually be diagnosed.

Fairness (if an action will eventually be scheduled)

The program will eventually terminate.