CSP

**Guarded Commands**
- Monitor: begin executing every call as soon as possible, waiting if the object is not in a proper state and signaling when the state is proper.
- CSP: the called object establishes conditions under which the call is accepted; calls not satisfying these conditions are held pending (no need for programmed wait/signal operations).

**Rendezvous**
- Monitor: the monitor is passive (has no independent task/thread/activity).
- CSP: synchronization between peer, autonomous activities.

CSP

**Distribution:**
- Monitor: inherently nondistributed in outlook and implementation.
- CSP: possibility for distributed programming using synchronous message passing.

---

**Communicating Sequential Processes (CSP)**
- Single thread of control
- Autonomous
- Encapsulated
- Named
- Static
- Synchronous
- Reliable
- Unidirectional
- Point-to-point
- Fixed topology

---

**Guarded Commands**
- Syntax: `<guard> -> <command list>`
- `<guard>` is a boolean expression, only one `?` must be at end of guard, considered true if message pending.

**Examples**
- `n < 10 -> A!index(n); n := n + 1;`
- `n < 10; A?index(n) -> next = MyArray(n);`
Communicating Sequential Processes (CSP)

Alternative Command
\[ \{ G_1 \rightarrow S_1 \}, \{ G_2 \rightarrow S_2 \}, \ldots, \{ G_n \rightarrow S_n \} \]
1. evaluate all guards
2. if more than one guard is true, nondeterministically select one.
3. if no guard is true, terminate.
Note: if all true guards end with an input command for which there is no pending message, then delay the evaluation until a message arrives. If all senders have terminated, then the alternative command terminates.

Repetitive Command
\* \{ G_1 \rightarrow S_1 \}, \{ G_2 \rightarrow S_2 \}, \ldots, \{ G_n \rightarrow S_n \}
repeatedly execute the alternative command until it terminates.

Examples:
\[ \{ x \geq y \rightarrow x := x \}, \{ y \geq x \rightarrow n := y \} \]
i := 0; \* \{ i < size; content(i) != n \rightarrow i := i + 1 \} \]
\* \{ n : integer; X!insert(n) \rightarrow INSERT \}
\{ n : integer; X!has(n) \rightarrow SEARCH \}

BoundedBuffer:
buffer : (0..9) portion;
in, out : integer; in := 0; out := 0;
\* \{ in < out + 10; producer?buffer(in mod 10) \rightarrow in := in + 1; \}
\{ out < in; consumer?more() \rightarrow consumer!buffer(out mod 10); out := out + 1; \}

ADA Example

\begin{verbatim}
    task bounded-buffer is
        entry store(x : buffer);
        entry remove(y : buffer);
    end;
    task body bounded-buffer is
        ...declarations...
    begin
        loop
            select
                when head < tail + 10 =>
                    accept store(x : buffer) ... end store;
            or
                when tail < head =>
                    accept remove(y : buffer) ... end remove;
        end select;
        end loop
    end
\end{verbatim}