**Dependent Composition**  \( P \cdot Q \) (sequential execution)

\( P \) and \( Q \) must have exactly the same state variables.

**Independent Composition**  \( P \| Q \) (parallel execution)

\( P \) and \( Q \) must have completely different state variables, and the state variables of the composition are those of both \( P \) and \( Q \).

Ignoring time and space variables

\[
P \| Q = P \land Q
\]

Example: in variable \( x \)

\[
x := x+1 = x' = x+1
\]

in variables \( y \) and \( z \)

\[
y := y+2 = y' = y+2 \land z' = z
\]

in variables \( x \), \( y \), and \( z \)

\[
x := x+1 \| y := y+2 = x' = x+1 \land y' = y+2 \land z' = z
\]
Partitioning:

If either \( x' \) or \( x:= \) appears in a process specification, then \( x \) belongs to that process, so neither \( x' \) nor \( x:= \) can appear in the other process specification. If neither \( x' \) nor \( x:= \) appears at all, then \( x \) can be placed on either side of the partition.

\[
x:= y \parallel y:= x \quad = \quad x'=y \land y'=x \land z'=z
\]

\( x \) belongs to the left process

\( y \) belongs to the right process

\( z \) belongs to either process

implementation of a process makes a private copy of the initial value of a variable belonging to the other process if the other process contains an assignment to that variable.
In boolean variable \( b \) and integer variable \( x \),

\[
\begin{align*}
b &:= x=x \parallel x:= x+1 & \text{replace } x=x \text{ by } T \\
&= b:= T \parallel x:= x+1 \\
\end{align*}
\]

\[
(x:= x+1. \ x:= x-1) \parallel y:= x \\
= ok \parallel y:= x \\
= y:= x
\]

\[
(x:= x+y. \ x:= x\times y) \parallel (y:= x-y. \ y:= x/y)
\]

versus

\[
(x:= x+y \parallel y:= x-y). \ (x:= x \times y \parallel y:= x/y)
\]

With time, independent composition is defined as

\[
P \parallel Q = \exists tP, tQ. \quad (\text{substitute } tP \text{ for } t' \text{ in } P ) \\
\wedge (\text{substitute } tQ \text{ for } t' \text{ in } Q ) \\
\wedge t' = \max tP \ tQ
\]
Laws of Independent Composition

\[(x := e \parallel y := f). \ P = (\text{for } x \text{ substitute } e \text{ and independently for } y \text{ substitute } f \text{ in } P)\]

\[P \parallel Q = Q \parallel P\] symmetry

\[P \parallel (Q \parallel R) = (P \parallel Q) \parallel R\] associativity

\[P \parallel ok = ok \parallel P = P\] identity

\[P \parallel Q \lor R = (P \parallel Q) \lor (P \parallel R)\] distributivity

\[P \parallel \text{if } b \text{ then } Q \text{ else } R\]

\[= \text{if } b \text{ then } (P \parallel Q) \text{ else } (P \parallel R)\] distributivity

\[\text{if } b \text{ then } (P \parallel Q) \text{ else } (R \parallel S)\]

\[= \text{if } b \text{ then } P \text{ else } R \parallel \text{if } b \text{ then } Q \text{ else } S\] distributivity
List Concurrency

\[ L_i := e \quad \Rightarrow \quad L_i' = e \land (\forall j \cdot j \neq i \Rightarrow L_j' = L_j) \land x' = x \land y' = y \land \ldots \]

partition within lists

Example: maximum item in a nonempty list

\[ \text{findmax } 0 \ (\# L) \ \text{ where} \]

\[ \text{findmax } = \lambda i, j \cdot i < j \Rightarrow L_i' = \text{MAX} \ L [i;..j] \]

\[ \text{findmax } i \ j \ \iff \ \begin{cases} \text{if } j - i = 1 \ \text{then ok} \\ \text{else} & ( ( \text{findmax } i \ (\text{div} \ (i+j) \ 2) \\ \text{||} \ \text{findmax} \ (\text{div} \ (i+j) \ 2) \ j) \\ L_i := \text{max} \ (L_i) \ (L \ (\text{div} \ (i+j) \ 2)) ) \end{cases} \]

recursive time = \( \text{ceil} \ (\log \ (j-i)) \)
Sequential to Parallel Transformation

\[ x := y. \quad x := x + 1. \quad z := y \]

\[ = \quad x := y. \quad (x := x + 1 \parallel z := y) \]

\[ = \quad (x := y. \quad x := x + 1) \parallel z := y \]

\[ \text{start} \rightarrow x := y \rightarrow x := x + 1 \rightarrow z := y \rightarrow \text{finish} \]
Whenever two programs occur in sequence, and neither assigns to a variable assigned in the other, and no variable assigned in the first appears in the second, they can be placed in parallel; a copy must be made of the initial value of any variable appearing in the first and assigned in the second.

Whenever two programs occur in sequence, and neither assigns to a variable appearing in the other, they can be placed in parallel without any copying of initial values.
Buffer

\[ produce = \ldots \cdot b := e \ldots \cdot \]
\[ consume = \ldots \cdot x := b \ldots \cdot \]
\[ control = produce. consume. control \]

\[ P \rightarrow C \rightarrow P \rightarrow C \rightarrow P \rightarrow C \rightarrow P \rightarrow C \rightarrow \]

\[ control = produce. newcontrol \]
\[ newcontrol = consume. produce. newcontrol \]
\[ newcontrol = (consume \parallel produce). newcontrol \]
produce = ..........p:= e.......... 

consume = ..........x:= c.......... 

control = produce. newcontrol 

newcontrol = c:= p. (consume || produce). newcontrol 

\[
\begin{array}{cccccccc}
P & P & P & P & P & P & P & P \\
B & B & B & B & B & B & B & B \\
C & C & C & C & C & C & C & C \\
\end{array}
\]

produce = ..........bw:= e.......... 

consume = ..........x:= br.......... 

control = produce. w:= w+1. consume. r:= r+1. control 

\[
\begin{array}{cccccccc}
P & W & P & W & P & W & P & W \\
C & R & C & R & C & R & C & R \\
\end{array}
\]

control = produce. w:= mod (w+1) n. 

consume. r:= mod (r+1) n. 

control
Insertion Sort

define

\[
\text{sort} \; = \; \lambda n \cdot \forall i, j: 0, \ldots, n \cdot i \leq j \Rightarrow L_i \leq L_j
\]

\[
\text{swap } i \; j \; = \; L_i := L_j \| L_j := L_i
\]

\[
\text{sort}' (\#L) \iff \text{sort } 0 \Rightarrow \text{sort}' (\#L)
\]

\[
\text{sort } 0 \Rightarrow \text{sort}' (\#L) \iff \text{for } n := 0; \ldots, \#L \text{ do sort } n \Rightarrow \text{sort}' (n+1)
\]

\[
\text{sort } n \Rightarrow \text{sort}' (n+1) \iff
\]

if \( n=0 \) then ok

else if \( L (n-1) \leq L n \) then ok

else (swap \( n-1 \) \( n \). sort \( n-1 \) \( \Rightarrow \) sort' \( n \))

\[
\begin{bmatrix}
L_0 & L_1 & L_2 & L_3 & L_4 \\
0 & 1 & 2 & 3 & 4 & 5
\end{bmatrix}
\]
If $\text{abs}(i-j) > 1$ then

$S_i$ and $S_j$ in parallel

$S_i$ and $C_j$ in parallel

$C_i$ and $C_j$ in parallel
Dining Philosophers

\[
\begin{align*}
\text{life} & = (P \ 0 \lor P \ 1 \lor P \ 2 \lor P \ 3 \lor P \ 4) \cdot \text{life} \\
P \ i & = \text{up} \ i \cdot \text{up}(i+1) \cdot \text{eat} \ i \cdot \text{dn} \ i \cdot \text{dn}(i+1) \\
\text{up} \ i & = \text{chopstick} \ i := \top \\
\text{dn} \ i & = \text{chopstick} \ i := \bot \\
\text{eat} \ i & = \ldots \text{chopstick} \ i \ldots \text{chopstick}(i+1) \ldots
\end{align*}
\]

If \( i \neq j \), \((\text{up} \ i \cdot \text{up} \ j)\) becomes \((\text{up} \ i \parallel \text{up} \ j)\).
If \( i \neq j \), \((\text{up} \ i \cdot \text{dn} \ j)\) becomes \((\text{up} \ i \parallel \text{dn} \ j)\).
If \( i \neq j \), \((\text{dn} \ i \cdot \text{up} \ j)\) becomes \((\text{dn} \ i \parallel \text{up} \ j)\).
If \( i \neq j \), \((\text{dn} \ i \cdot \text{dn} \ j)\) becomes \((\text{dn} \ i \parallel \text{dn} \ j)\).
If \( i \neq j \land i+1 \neq j \), \((\text{eat} \ i \cdot \text{up} \ j)\) becomes \((\text{eat} \ i \parallel \text{up} \ j)\).
If \( i \neq j \land i \neq j+1 \), \((\text{up} \ i \cdot \text{eat} \ j)\) becomes \((\text{up} \ i \parallel \text{eat} \ j)\).
If \( i \neq j \land i+1 \neq j \), \((\text{eat} \ i \cdot \text{dn} \ j)\) becomes \((\text{eat} \ i \parallel \text{dn} \ j)\).
If \( i \neq j \land i \neq j+1 \), \((\text{dn} \ i \cdot \text{eat} \ j)\) becomes \((\text{dn} \ i \parallel \text{eat} \ j)\).
If \( i \neq j \land i+1 \neq j \land i \neq j+1 \), \((\text{eat} \ i \cdot \text{eat} \ j)\) becomes \((\text{eat} \ i \parallel \text{eat} \ j)\).

\[
\begin{align*}
\text{life} & = P \ 0 \ || \ P \ 1 \ || \ P \ 2 \ || \ P \ 3 \ || \ P \ 4 \\
P \ i & = (\text{up} \ i \ || \text{up}(i+1)) \cdot \text{eat} \ i \cdot (\text{dn} \ i \ || \text{dn}(i+1)) \cdot P \ i
\end{align*}
\]