Message Passing

Johannes Åman Pohjola
CSE, UNSW
Term 2 2022
Where we are at

Last week, we saw *semaphores* and *monitors*, concluding our examination of shared variable concurrency.

For the rest of this course, our focus will be on *message passing*.
**Distributed Programming**

**distributed program:** processes can be distributed across machines → no shared memory (usually)
processes share *communication channels* for message passing

**languages:** Promela (synchronous and asynchronous MP), Java (RPC, RMI, ...)

**libraries:** sockets, message passing interface (MPI), parallel virtual machine (PVM) etc.
Message Passing

A *channel* is a typed FIFO queue between processes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Ben-Ari</th>
<th>Promela</th>
</tr>
</thead>
<tbody>
<tr>
<td>send a message</td>
<td>ch ⇐ x</td>
<td>ch ! x</td>
</tr>
<tr>
<td>receive a message</td>
<td>ch ⇒ y</td>
<td>ch ? y</td>
</tr>
</tbody>
</table>
## Message Passing

A *channel* is a typed FIFO queue between processes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Ben-Ari</th>
<th>Promela</th>
</tr>
</thead>
<tbody>
<tr>
<td>send a message</td>
<td>$\text{ch} \leftarrow x$</td>
<td>$\text{ch} ! x$</td>
</tr>
<tr>
<td>recieve a message</td>
<td>$\text{ch} \Rightarrow y$</td>
<td>$\text{ch} ? y$</td>
</tr>
</tbody>
</table>

### Synchronous channels

A *synchronous* channel has queue capacity 0. Both the send and the receive operation block until both are ready. When they are, they execute at the same time, and assign the value of $x$ to $y$. 
Message Passing

A *channel* is a typed FIFO queue between processes.

<table>
<thead>
<tr>
<th>Action</th>
<th>Ben-Ari</th>
<th>Promela</th>
</tr>
</thead>
<tbody>
<tr>
<td>send a message</td>
<td>( ch \leftarrow x )</td>
<td>( ch ! x )</td>
</tr>
<tr>
<td>receive a message</td>
<td>( ch \rightarrow y )</td>
<td>( ch ? y )</td>
</tr>
</tbody>
</table>

**Synchronous channels**

A *synchronous* channel has queue capacity 0. Both the send and the receive operation block until both are ready. When they are, they execute at the same time, and assign the value of \( x \) to \( y \).

**Asynchronous channels**

For *asynchronous* channels, send doesn’t block. It appends the value of \( x \) to the queue of \( ch \). Receive blocks, until \( ch \) contains a message. When it does, the oldest message is removed, and its content is stored in \( y \).
Taxonomy of Asynchronous Message Passing

Asynchronous channels may be...

**Reliable:** all messages sent will eventually arrive.

**Lossy:** messages may be lost in transit.

**FIFO:** messages will arrive in order.

**Unordered:** messages can arrive out-of-order.

**Error-detecting:** received messages aren’t garbled in transit (or if they are, we can tell).
Taxonomy of Asynchronous Message Passing

Asynchronous channels may be...

**Reliable:** all messages sent will eventually arrive.

**Lossy:** messages may be lost in transit.

**FIFO:** messages will arrive in order.

**Unordered:** messages can arrive out-of-order.

**Error-detecting:** received messages aren’t garbled in transit (or if they are, we can tell).

**Example**

TCP is reliable and FIFO. UDP is lossy and unordered, but error-detecting.
**Algorithm 2.1: Producer-consumer (channels)**

channel of integer ch

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
</table>
| integer x  
loop forever  
p1: $x \leftarrow$ produce  
p2: $ch \leftarrow x$ | integer y  
loop forever  
q1: $ch \Rightarrow y$  
q2: consume(y) |
Conway’s Problem

Example

**Input** on channel inC: a sequence of characters

**Output** on channel outC:
- The sequence of characters from inC, with runs of $2 \leq n \leq 9$ occurrences of the same character $c$ replaced by the $n$ and $c$
- a newline character after every $K$th character in the output.
Conway’s Problem

**Example**

**Input** on channel inC: a sequence of characters

**Output** on channel outC:
- The sequence of characters from inC, with runs of $2 \leq n \leq 9$ occurrences of the same character $c$ replaced by the $n$ and $c$
- a newline character after every $K$th character in the output.

Let’s use message-passing for separation of concerns:

```
| inC | compress | pipe | output | outC |
```
Algorithm 2.2: Conway’s problem

<table>
<thead>
<tr>
<th>compress</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant integer MAX ← 9</td>
<td>char c</td>
</tr>
<tr>
<td>constant integer K ← 4</td>
<td>integer m ← 0</td>
</tr>
<tr>
<td>channel of integer inC, pipe, outC</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>compress</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>inC ⇒ previous</td>
<td></td>
</tr>
</tbody>
</table>

| loop forever |
| p1: inC ⇒ c |
| p2: if (c = previous) and (n < MAX - 1) |
| p3: n ← n + 1 |
| else |
| p4: if n > 0 |
| p5: pipe ← i2c(n+1) |
| p6: n ← 0 |
| p7: pipe ← previous |
| p8: previous ← c |

| loop forever |
| q1: pipe ⇒ c |
| q2: outC ← c |
| q3: m ← m + 1 |
| q4: if m >= K |
| q5: outC ← newline |
| q6: m ← 0 |
| q7: |
| q8: |
Reminder: Matrix Multiplication

Example

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\times
\begin{pmatrix}
1 & 0 & 2 \\
0 & 1 & 2 \\
1 & 0 & 0
\end{pmatrix}
=
\begin{pmatrix}
4 & 2 & 6 \\
10 & 5 & 18 \\
16 & 8 & 30
\end{pmatrix}
\]
Reminder: Matrix Multiplication

Example

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\times
\begin{pmatrix}
1 & 0 & 2 \\
0 & 1 & 2 \\
1 & 0 & 0
\end{pmatrix}
= 
\begin{pmatrix}
4 & 2 & 6 \\
10 & 5 & 18 \\
16 & 8 & 30
\end{pmatrix}
\]

Let \( p, q, r \in \mathbb{N} \). Let \( A = (a_{i,j})_{1 \leq i \leq p} \in \mathbb{T}^{p \times q} \) and \( B = (b_{j,k})_{1 \leq j \leq q} \in \mathbb{T}^{q \times r} \) be two (compatible) matrices. Recall that the matrix \( C = (c_{i,k})_{1 \leq i \leq p} \in \mathbb{T}^{p \times r} \) is their product, \( A \times B \), iff, for all \( 1 \leq i \leq p \) and \( 1 \leq k \leq r \):

\[ c_{i,j} = \sum_{j=1}^{q} a_{i,j} b_{j,k} \]
Algorithms for Matrix Multiplication

The standard algorithm for matrix multiplication is:

for all rows \( i \) of \( A \) do:
  for all columns \( k \) of \( B \) do:
    set \( c_{i,k} \) to 0
    for all columns \( j \) of \( A \) do:
      add \( a_{i,j}b_{j,k} \) to \( c_{i,k} \)

Because of the three nested loops, its complexity is \( \mathcal{O}(p \cdot q \cdot r) \).
The standard algorithm for matrix multiplication is:

for all rows \( i \) of \( A \) do:
    for all columns \( k \) of \( B \) do:
        set \( c_{i,k} \) to 0
        for all columns \( j \) of \( A \) do:
            add \( a_{i,j} b_{j,k} \) to \( c_{i,k} \)

Because of the three nested loops, its complexity is \( O(p \cdot q \cdot r) \). In case both matrices are quadratic, i.e., \( p = q = r \), that’s \( O(p^3) \).
Process Array for Matrix Multiplication

- **Source**:
  - 1: 4,2,6
  - 2: 3,2,4
  - 3: 3,0,0
  - 4: 10,5,18
  - 5: 6,5,10
  - 6: 6,0,0
  - 7: 16,8,30
  - 8: 9,8,16
  - 9: 9,0,0

- **Result**:
  - 1: 4,2,6
  - 2: 3,2,4
  - 3: 3,0,0
  - 4: 10,5,18
  - 5: 6,5,10
  - 6: 6,0,0
  - 7: 16,8,30
  - 8: 9,8,16
  - 9: 9,0,0

- **Zero**:
  - Source 1
  - Source 2
  - Source 3

- **Sink**:
  - Result 1
  - Result 2
  - Result 3
  - Result 4
  - Result 5
  - Result 6
  - Result 7
  - Result 8
  - Result 9
Computation of One Element
Algorithm 2.3: Multiplier process with channels

<table>
<thead>
<tr>
<th>Loop forever</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
</tr>
<tr>
<td>p2:</td>
</tr>
<tr>
<td>p3:</td>
</tr>
<tr>
<td>p4:</td>
</tr>
<tr>
<td>p5:</td>
</tr>
</tbody>
</table>
Algorithm 2.4: Multiplier with channels and selective input

<table>
<thead>
<tr>
<th></th>
<th>integer FirstElement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>channel of integer North, East, South, West</td>
</tr>
<tr>
<td></td>
<td>integer Sum, integer SecondElement</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td></td>
</tr>
<tr>
<td>either</td>
<td></td>
</tr>
<tr>
<td>p1:</td>
<td>North ⇒ SecondElement</td>
</tr>
<tr>
<td>p2:</td>
<td>East ⇒ Sum</td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>p3:</td>
<td>East ⇒ Sum</td>
</tr>
<tr>
<td>p4:</td>
<td>North ⇒ SecondElement</td>
</tr>
<tr>
<td>p5:</td>
<td>South ⇐ SecondElement</td>
</tr>
<tr>
<td>p6:</td>
<td>Sum ⇐ Sum + FirstElement · SecondElement</td>
</tr>
<tr>
<td>p7:</td>
<td>West ⇐ Sum</td>
</tr>
</tbody>
</table>
Multiplier Process in Promela

```plaintext
proctype Multiplier(byte Coeff;
        chan North;
        chan East;
        chan South;
        chan West)
{
    byte Sum, X;
    for (i : 0..(SIZE-1)) {
        if :: North ? X -> East ? Sum;
        :: East ? Sum -> North ? X;
    fi;
    South ! X;
    Sum = Sum + X*Coeff;
    West ! Sum;
    }
}
Algorithm 2.5: Dining philosophers with channels

channel of boolean forks[5]

<table>
<thead>
<tr>
<th>philosopher i</th>
<th>fork i</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean dummy</td>
<td>boolean dummy</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: think</td>
<td>q1: forks[i] ⇐ true</td>
</tr>
<tr>
<td>p2: forks[i] ⇒ dummy</td>
<td>q2: forks[i] ⇒ dummy</td>
</tr>
<tr>
<td>p3: forks[i+1] ⇒ dummy</td>
<td>q3:</td>
</tr>
<tr>
<td>p4: eat</td>
<td>q4:</td>
</tr>
<tr>
<td>p5: forks[i] ⇐ true</td>
<td>q5:</td>
</tr>
<tr>
<td>p6: forks[i+1] ⇐ true</td>
<td>q6:</td>
</tr>
</tbody>
</table>

NB

The many shared channels make it possible to give forks directly to other philosophers, rather than putting them back on the table.
Synchronous Message Passing

Recall that, when message passing is synchronous, the exchange of a message requires *coordination* between sender and receiver (sometimes called a *handshaking* mechanism).

In other words, the sender is *blocked* until the receiver is ready to cooperate.
Synchronous Transition Diagrams

Definition

A *synchronous transition diagram* is a parallel composition $P_1 \parallel \ldots \parallel P_n$ of $n$ (sequential) transition diagrams $P_1, \ldots, P_n$ called *processes*. The processes $P_i$

- do not share variables
- communicate along channels $C, D, \ldots$ connecting processes by way of
  - *output* statements $C \leftarrow e$ for sending the value of expression $e$ along channel $C$
  - *input* statements $C \Rightarrow x$ for receiving a value along channel $C$ into variable $x$
For shared variable concurrency, labels $b; f$, where $b$ is a Boolean condition and $f$ is a state transformation sufficed.

**Example**

Now, we call such transitions *internal*. 
I/O Transitions

We extend this notation to message passing by allowing the guard to be combined with an input or an output statement:

\[ \ell \xrightarrow{b; C \Rightarrow x; f} \ell' \]

\[ \ell \xleftarrow{b; C \Leftarrow e; f} \ell' \]
Let $P = P_1 \parallel P_2$ be given as:

![Example Diagram]

Obviously, $\{\top\} P \{x = 1\}$, but how to prove it?
Some notation

For an $n$-tuple $x = \langle x_1, \ldots, x_i, \ldots, x_n \rangle$, we define

$$x[i \leftarrow e] = \langle x_1, \ldots, e, \ldots, x_n \rangle$$

$x[i \leftarrow e]$ is like $x$, except the $i$:th element is replaced with $e$.

Example

$\langle 1, 5, 7 \rangle[2 \leftarrow 3] = \langle 1, 3, 7 \rangle$
Definition

The *closed product* of $P_i = (L_i, T_i, s_i, t_i)$, for $1 \leq i \leq n$ (with disjoint local variable sets), is defined as $P = (L, T, s, t)$, where:

$$
L = L_1 \times \ldots \times L_n \quad s = \langle s_1, \ldots, s_n \rangle \quad t = \langle t_1, \ldots, t_n \rangle
$$

and $\ell \xrightarrow{a} \ell' \in T$ holds iff

1. $\ell' = \ell[i \to \ell'_i]$ and $\ell_i \xrightarrow{a} \ell'_i \in T_i$ is an internal transition, or
2. $\ell' = \ell[i \to \ell'_i][j \to \ell'_j]$ and $i \neq j$,

with $\ell_i \xrightarrow{b; C \Leftarrow e; f} \ell'_i \in T_i$ and $\ell_j \xrightarrow{b'; C \Rightarrow x; g} \ell'_j \in T_j$, and

$a = b \land b'; f \circ g \circ [x \leftarrow e]$
Example 1 cont’d

Observe that the closed product is just

\[
\langle s_1, s_2 \rangle \xleftarrow{x} 1 \rightarrow \langle t_1, t_2 \rangle
\]

so validity of \( \{\top\} \ P \{x = 1\} \) follows from

\[
\models \top \Rightarrow (x = 1) \circ [x \leftarrow 1]
\]

which is immediate.

(See the glossary of notation for the meaning of all these strange symbols.)
Verification

To show that the Hoare triple

$$\{\phi\} P_1 \parallel \ldots \parallel P_n \{\psi\}$$

is valid, it suffices to prove

$$\{\phi\} P \{\psi\}$$

where $P$ is the closed product of the $P_i$. 
Verification

To show that the Hoare triple

\[ \{ \phi \} P_1 \parallel \ldots \parallel P_n \{ \psi \} \]

is valid, it suffices to prove

\[ \{ \phi \} P \{ \psi \} \]

where \( P \) is the closed product of the \( P_i \).

There are no I/O transitions in \( P \), so Floyd’s method works.
Verification

To show that the Hoare triple

\[
\{ \phi \} \ P_1 \parallel \ldots \parallel P_n \ \{ \psi \}
\]

is valid, it suffices to prove

\[
\{ \phi \} \ P \ \{ \psi \}
\]

where \( P \) is the closed product of the \( P_i \).

There are no I/O transitions in \( P \), so Floyd’s method works.

Disadvantage

As with the product construction for shared-variable concurrency, the closed product is exponential in the number of processes.
Verification

To show that the Hoare triple

\[ \{ \phi \} \ P_1 \parallel \ldots \parallel P_n \{ \psi \} \]

is valid, it suffices to prove

\[ \{ \phi \} \ P \{ \psi \} \]

where \( P \) is the closed product of the \( P_i \).

There are no I/O transitions in \( P \), so Floyd’s method works.

Disadvantage

As with the product construction for shared-variable concurrency, the closed product is exponential in the number of processes.

Is there an Owicki-Gries equivalent for synchronous message passing?
A Simplistic Method

For each location \( \ell \) in some \( L_i \), find a local predicate \( Q_\ell \), only depending on \( P_i \)'s local variables.

1. Prove that, for all \( i \), the local verification conditions hold, i.e.,
\[
\models Q_\ell \land b \to Q_{\ell'} \circ f \text{ for each } \ell \xrightarrow{b;f} \ell' \in T_i.
\]

2. For all \( i \neq j \) and matching pairs of I/O transitions
\[
\ell_i \xrightarrow{b;C\leftarrow e;f} \ell'_i \in T_i \text{ and } \ell_j \xrightarrow{b';C\rightarrow x;g} \ell'_j \in T_j \text{ show that}
\]
\[
\models Q_{\ell_i} \land Q_{\ell_j} \land b \land b' \implies (Q_{\ell'_i} \land Q_{\ell'_j}) \circ f \circ g \circ [x \leftarrow e].
\]

3. Prove \( \models \phi \implies Q_{s_1} \land \ldots \land Q_{s_n} \) and \( \models Q_{t_1} \land \ldots \land Q_{t_n} \implies \psi \).
Proof of Example 1

There are no internal transitions. There's one matching pair.

\[ \top \implies (x = 1) \circ [x \leftarrow 1] \equiv 1 = 1 \equiv \top \]
Soundness & Incompleteness

The simplistic method is sound but not complete. It generates proof obligations for all *syntactically matching* I/O transition pairs, regardless of whether these pairs can actually be matched *semantically* (in an execution).
Example 2

Let $P = P_1 \parallel P_2$ be given as:

We cannot prove $\{\top\} P \{x = 2\}$ using the simplistic method. Proof obligations for the transition pairs $(T_1, T_4)$ and $(T_2, T_3)$ must be discharged. This leads to a contradiction: no assertion network can make the simplistic method work for this example.
Remedy 1: Adding Shared Auxiliary Variables

Use shared, write-only auxiliary variables to relate locations in different processes. Only output transitions need to be augmented with assignments to these shared auxiliary variables.

**Pro** easy

**Con** incomplete when channels are shared between more than two process.

**Con** re-introduces interference freedom tests for matching pairs $\ell_i \xrightarrow{b_i; C \leftarrow e; f_i} \ell'_i \in T_i$ and $\ell_j \xrightarrow{b_j; C \Rightarrow x; f_j} \ell'_j \in T_j$, and location $\ell_m$ of process $P_m$, $m \neq i, j$:

$$\models Q_{\ell_i} \land Q_{\ell_j} \land Q_{\ell_m} \land b_i \land b_j \implies Q_{\ell_m} \circ f_i \circ f_j \circ [x \leftarrow e]$$

[This method is due to Levin & Gries.]
Example 2 cont’d

\[ s_1 \xrightarrow{C \leftarrow 1} l_1 \xrightarrow{C \leftarrow 2} t_1 \]

\[ s_2 \xrightarrow{C \Rightarrow x} l_2 \xrightarrow{C \Rightarrow x} t_2 \]
Example 2 cont’d

\[s_1 \xrightarrow{C \leftarrow 1; \ k \leftarrow 1} \ell_1 \xrightarrow{C \leftarrow 2; \ k \leftarrow 2} t_1\]

\[s_2 \xrightarrow{C \Rightarrow x} \ell_2 \xrightarrow{C \Rightarrow x} t_2\]
Example 2 cont’d

\[
\begin{align*}
&\ s_1 \quad C \leftarrow 1; \; k \leftarrow 1 \quad \ell_1 \quad C \leftarrow 2; \; k \leftarrow 2 \quad t_1 \\
&\ k = 0 \quad k = 1 \quad k = 2
\end{align*}
\]

\[
\begin{align*}
&\ s_2 \quad C \Rightarrow x \quad \ell_2 \quad C \Rightarrow x \quad t_2 \\
&\ k = 0 \quad k = 1 \quad k = 2 \land x = 2
\end{align*}
\]
Levin & Gries-style Proof for Example 2

There are no internal transitions. Four matching I/O transition pairs exist, the same as in the simplistic method. Proof obligations:

1. \[ k = 0 \implies (k = 1) \circ [k \leftarrow 1] \circ [x \leftarrow 1] \]
2. \[ k = 0 \land k = 1 \implies (k = 1 \land k = 2 \land x = 2) \circ [k \leftarrow 1] \circ [x \leftarrow 1] \]
3. \[ k = 1 \land k = 0 \implies (k = 2 \land k = 1) \circ [k \leftarrow 2] \circ [x \leftarrow 2] \]
4. \[ k = 1 \implies (k = 2 \land x = 2) \circ [k \leftarrow 2] \circ [x \leftarrow 2] \]

No interference freedom proof obligations are generated in this example since there is no third process.
Levin & Gries-style Proof for Example 2 cont’d

Thanks to contradictory assumptions about $k$, (2) and (3) are vacuously true.

The right-hand-sides of the implications (1) and (4) simplify to $\top$, which discharges those proof obligations, e.g., for the RHS of (1):

$$(k = 1) \circ [k \leftarrow 1] \circ [x \leftarrow 1] \equiv 1 = 1$$

$$\equiv \top$$
Remedy 2: Local Auxiliary Variables + Invariant

Use *local, write only* auxiliary variables + a global *communication invariant* $I$ to relate values of local auxiliary variables in the various processes.

**Pro** no interference freedom tests

**Con** more complicated proof obligation for communication steps:

$$
\models Q_{\ell_i} \land Q_{\ell_j} \land b \land b' \land I \implies (Q_{\ell'_i} \land Q_{\ell'_j} \land I) \circ f \circ g \circ [x \leftarrow e]
$$

[This is the *AFR method*, named for Apt, Francez, and de Roever.]
Example 2 cont’d

\[ s_1 \xrightarrow{C \leftarrow 1} \ell_1 \xrightarrow{C \leftarrow 2} t_1 \]

\[ s_2 \xrightarrow{C \Rightarrow x} \ell_2 \xrightarrow{C \Rightarrow x} t_2 \]
Example 2 cont’d

\[ C \leftarrow 1; \quad k_1 \leftarrow 1 \]

\[ s_1 \quad \rightarrow \quad \ell_1 \]

\[ C \leftarrow 2; \quad k_1 \leftarrow 2 \]

\[ \ell_1 \quad \rightarrow \quad t_1 \]

\[ C \Rightarrow x; \quad k_2 \leftarrow 1 \]

\[ s_2 \quad \rightarrow \quad \ell_2 \]

\[ C \Rightarrow x; \quad k_2 \leftarrow 2 \]

\[ \ell_2 \quad \rightarrow \quad t_2 \]
Example 2 cont’d

\[
\begin{align*}
&s_1 \quad C \leftarrow 1; \ k_1 \leftarrow 1 \\
&\quad k_1 = 0
\end{align*}
\]

\[
\begin{align*}
&\ell_1 \quad C \leftarrow 2; \ k_1 \leftarrow 2 \\
&\quad k_1 = 1
\end{align*}
\]

\[
\begin{align*}
&t_1 \\
&\quad k_1 = 2
\end{align*}
\]

\[
\begin{align*}
&s_2 \quad C \Rightarrow x; \ k_2 \leftarrow 1 \\
&\quad k_2 = 0
\end{align*}
\]

\[
\begin{align*}
&\ell_2 \quad C \Rightarrow x; \ k_2 \leftarrow 2 \\
&\quad k_2 = 1
\end{align*}
\]

\[
\begin{align*}
&t_2 \\
&\quad k_2 = 2 \land x = 2
\end{align*}
\]
Example 2 cont’d

Define \( I \equiv (k_1 = k_2) \).
**AFR-style Proof for Example 2**

There are no internal transitions. Four matching pairs of I/O transitions exist, with these proof obligations:

\[ k_1 = 0 \land k_2 = 0 \land k_1 = k_2 \implies (k_1 = 1 \land k_2 = 1 \land k_1 = k_2) \circ [k_1 \leftarrow 1] \circ [k_2 \leftarrow 1] \circ [x \leftarrow 1] \]  

(5)

\[ k_1 = 0 \land k_2 = 1 \land k_1 = k_2 \implies (k_1 = 1 \land k_2 = 2 \land x = 2 \land k_1 = k_2) \circ [k_1 \leftarrow 1] \circ [k_2 \leftarrow 2] \circ [x \leftarrow 1] \]  

(6)

\[ k_1 = 1 \land k_2 = 0 \land k_1 = k_2 \implies (k_1 = 2 \land k_2 = 1 \land k_1 = k_2) \circ [k_1 \leftarrow 2] \circ [k_2 \leftarrow 1] \circ [x \leftarrow 2] \]  

(7)

\[ k_1 = 1 \land k_2 = 1 \land k_1 = k_2 \implies (k_1 = 2 \land k_2 = 2 \land x = 2 \land k_1 = k_2) \circ [k_1 \leftarrow 2] \circ [k_2 \leftarrow 2] \circ [x \leftarrow 2] \]  

(8)
AFR-style Proof for Example 2 cont’d

Thanks to the invariant $k_1 = k_2$, (6) and (7) are vacuously true. The right-hand-sides of the implications (5) and (8) simplify to $\top$, which discharges those proof obligations, e.g., for the RHS of (8):

\[
(k_1 = 2 \land k_2 = 2 \land x = 2 \land k_1 = k_2) \circ [k_1 \leftarrow 2] \circ [k_2 \leftarrow 2] \circ [x \leftarrow 2]
\]

\[
\equiv 2 = 2 \land 2 = 2 \land 2 = 2 \land 2 = 2
\]

\[
\equiv \top
\]
What Now?

Next lecture, we’ll be looking at proof methods for termination (convergence and deadlock freedom) in sequential, shared-variable concurrent, and message-passing concurrent settings.

After the break, we’ll see a compositional proof method for verification, proving properties for asynchronous communication, and, if time on Thursday, we’ll talk about process algebra.

Assignment 1 is out! Read the spec ASAP!
For each $\ell \in L_i$, the annotation $Q_\ell$ should only depend on $P_i$'s local variables, and shared write-only auxiliary variables. Shared variables should only be assigned to in output transitions.

1. Prove that, for all $i$, the local verification conditions hold, i.e.,

$$\models Q_\ell \land b \rightarrow Q_\ell' \circ f$$

for each $\ell \xrightarrow{b;f} \ell' \in T_i$.

2. For all $i \neq j$ and $\ell_i \xrightarrow{b;C \Leftarrow e;f} \ell'_i \in T_i$ and $\ell_j \xrightarrow{b';C \Rightarrow x;g} \ell'_j \in T_j$ show that

$$\models Q_{\ell_i} \land Q_{\ell_j} \land b \land b' \Rightarrow (Q_{\ell'_i} \land Q_{\ell'_j}) \circ f \circ g \circ [x \leftarrow e].$$

3. For all $i \neq j$ and $\ell_i \xrightarrow{b_i;C \Leftarrow e;f_i} \ell'_i \in T_i$ and $\ell_j \xrightarrow{b_j;C \Rightarrow x;f_j} \ell'_j \in T_j$, and location $\ell_m$ of process $P_m$, $m \neq i, j$:

$$\models Q_{\ell_i} \land Q_{\ell_j} \land Q_{\ell_m} \land b_i \land b_j \Rightarrow Q_{\ell_m} \circ f_i \circ f_j \circ [x \leftarrow e].$$
Levin & Gries in full, part 2

Let \( k_1, \ldots, k_m \) be all auxiliary variables used in the transition diagrams. We assume that \( \phi, \psi \) mentions none of these auxiliaries.

4 Prove

\[
\models \phi \implies \exists k_1, \ldots, k_m. Q_{s_1} \land \cdots \land Q_{s_n}
\]

and

\[
\models Q_{t_1} \land \cdots \land Q_{t_n} \implies \psi
\]

These four items suffice to prove \( \{ \phi \} P \{ \psi \} \), where \( P \) is the closed product of the \( P_i \):s.

NB

The existential quantification over the shared variables allows the final Hoare triple to make no mention of the auxiliary variables.
AFR in full, part 1

For each $\ell \in L_i$, the annotation $Q_\ell$ should only depend on $P_i$’s local variables, and local write-only auxiliary variables. Auxiliary variables should only be assigned to in I/O transitions. The communication invariant $I$ should only mention auxiliary variables.

1. Prove that, for all $i$, the local verification conditions hold, i.e.,
   \[ \models Q_\ell \land b \rightarrow Q_\ell' \circ f \text{ for each } \ell \xrightarrow{b;f} \ell' \in T_i. \]

2. For all $i \neq j$ and $\ell_i \xrightarrow{b_i;C\Leftarrow e;f_i} \ell'_i \in T_i$ and $\ell_j \xrightarrow{b_j;C\Rightarrow x;f_j} \ell'_j \in T_j$, and location $\ell_m$ of process $P_m$, $m \neq i, j$:
   \[ \models Q_{\ell_i} \land Q_{\ell_j} \land b \land b' \land I \implies (Q_{\ell'_i} \land Q_{\ell'_j} \land I) \circ f \circ g \circ [x \leftarrow e] \]
AFR in full, part 2

Let $k_1, \ldots, k_m$ be all auxiliary variables used in the transition diagrams. We assume that $\phi, \psi$ mentions none of these auxiliaries.

3 Prove

$$\models \phi \implies \exists k_1, \ldots, k_m. \ Q_{s_1} \land \ldots \land Q_{s_n} \land I$$

and

$$\models Q_{t_1} \land \ldots \land Q_{t_n} \land I \implies \psi$$

These three items suffice to prove $\{\phi\} \ P \ \{\psi\}$, where $P$ is the closed product of the $P_i$:
s.

NB

We could allow non-auxiliary variables in $I$, at the expense of making proof obligation 1 more involved.