Message Passing

Johannes Åman Pohjola
CSIRO's Data61 (and UNSW)
Term 2 2021
In the last lecture, we saw monitors and the readers and writers problem, concluding our examination of shared variable concurrency. For the rest of this course, our focus will be on message passing. It’s a useful concurrency abstraction on one computer, and the foundation for distributed programming. In this lecture, we will introduce message passing and discuss simple proof techniques for synchronous message passing.
Distributed Programming

**distributed program:** processes can be distributed across machines → they cannot use shared variables (usually; DSM exception)
processes do share *communication channels* for message passing
**languages:** Promela (synchronous and asynchronous MP), Java (RPC)
**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><code>ch ← x</code></td>
<td><code>ch ! x</code></td>
</tr>
<tr>
<td>recieve a message</td>
<td><code>ch → y</code></td>
<td><code>ch ? y</code></td>
</tr>
</tbody>
</table>
Message Passing

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

- **Ben-Ari**
  - send a message: `ch ≡ x`
  - receive a message: `ch ⇒ y`
- **Promela**
  - send a message: `ch ! x`
  - receive a message: `ch ? y`

**Synchronous channels**

A *synchronous* channel has queue capacity 0. Both the send and the receive operation block until they 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>Send a message</th>
<th>Recieve a message</th>
</tr>
</thead>
<tbody>
<tr>
<td>send a message</td>
<td>ch ↔ x</td>
</tr>
<tr>
<td>recieve a message</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 they 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 associated with channel ch. The receive operation 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).
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>
<tbody>
<tr>
<td>integer <code>x</code>&lt;br&gt;loop forever&lt;br&gt;p1: <code>x ← produce</code>&lt;br&gt;p2: <code>ch ← x</code></td>
<td>integer <code>y</code>&lt;br&gt;loop forever&lt;br&gt;q1: <code>ch ⇒ y</code>&lt;br&gt;q2: <code>consume(y)</code></td>
</tr>
</tbody>
</table>
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:
### 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></td>
</tr>
<tr>
<td>constant integer K ← 4</td>
<td></td>
</tr>
<tr>
<td>channel of integer inC, pipe, outC</td>
<td></td>
</tr>
<tr>
<td><strong>compress</strong></td>
<td><strong>output</strong></td>
</tr>
<tr>
<td>char c, previous ← 0</td>
<td>char c</td>
</tr>
<tr>
<td>integer n ← 0</td>
<td>integer m ← 0</td>
</tr>
<tr>
<td>inC ⇒ previous</td>
<td></td>
</tr>
<tr>
<td><strong>loop</strong> forever</td>
<td><strong>loop</strong> forever</td>
</tr>
<tr>
<td><strong>p1:</strong> inC ⇒ c</td>
<td>q1: pipe ⇒ c</td>
</tr>
<tr>
<td><strong>p2:</strong> if (c = previous) and (n &lt; MAX − 1)</td>
<td>q2: outC ← c</td>
</tr>
<tr>
<td></td>
<td>q3: m ← m + 1</td>
</tr>
<tr>
<td><strong>p3:</strong> n ← n + 1</td>
<td>q4: if m ≥ K</td>
</tr>
<tr>
<td>else</td>
<td>q5: outC ← newline</td>
</tr>
<tr>
<td><strong>p4:</strong> if n &gt; 0</td>
<td>q6: m ← 0</td>
</tr>
<tr>
<td><strong>p5:</strong> pipe ← i2c(n+1)</td>
<td></td>
</tr>
<tr>
<td><strong>p6:</strong> n ← 0</td>
<td>q7:</td>
</tr>
<tr>
<td><strong>p7:</strong> pipe ← previous</td>
<td></td>
</tr>
<tr>
<td><strong>p8:</strong> previous ← c</td>
<td>q8:</td>
</tr>
</tbody>
</table>
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

\[
C = (c_{i,k})_{1 \leq i \leq p} \in \mathbb{T}^{p \times s}
\]

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 $O(p \cdot q \cdot r)$. 
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 $O(p \cdot q \cdot r)$. In case both matrices are quadratic, i.e., $p = q = r$, that’s $O(p^3)$. 
Process Array for Matrix Multiplication

- Source 1: 2, 0, 1
- Source 2: 2, 1, 0
- Source 3: 0, 0, 1
- Result 1: 4, 2, 6
- Result 2: 3, 2, 4
- Result 3: 3, 0, 0
- Result 1: 10, 5, 18
- Result 2: 6, 5, 10
- Result 3: 6, 0, 0
- Result 1: 16, 8, 30
- Result 2: 9, 8, 16
- Result 3: 9, 0, 0
- Sink 1
- Sink 2
- Sink 3
- Zero
Computation of One Element
### Algorithm 2.3: Multiplier process with channels

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

<table>
<thead>
<tr>
<th>loop forever</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: North ⇒ SecondElement</td>
</tr>
<tr>
<td>p2: East ⇒ Sum</td>
</tr>
<tr>
<td>p3: Sum ← Sum + FirstElement · SecondElement</td>
</tr>
<tr>
<td>p4: South ← SecondElement</td>
</tr>
<tr>
<td>p5: West ← Sum</td>
</tr>
</tbody>
</table>
Algorithm 2.4: Multiplier with channels and selective input

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

```
loop forever
  either
    p1: North ⇒ SecondElement
    p2: East ⇒ Sum
  or
    p3: East ⇒ Sum
    p4: North ⇒ SecondElement
    p5: South ⇐ SecondElement
    p6: Sum ⇐ Sum + FirstElement · SecondElement
    p7: West ⇐ Sum
```
Multiplier Process in Promela

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><strong>p1:</strong> think</td>
<td><strong>q1:</strong> forks[i] ⇐ true</td>
</tr>
<tr>
<td><strong>p2:</strong> forks[i] ⇒ dummy</td>
<td><strong>q2:</strong> forks[i] ⇒ dummy</td>
</tr>
<tr>
<td><strong>p3:</strong> forks[i+1] ⇒ dummy</td>
<td><strong>q3:</strong></td>
</tr>
<tr>
<td><strong>p4:</strong> eat</td>
<td><strong>q4:</strong></td>
</tr>
<tr>
<td><strong>p5:</strong> forks[i] ⇐ true</td>
<td><strong>q5:</strong></td>
</tr>
<tr>
<td><strong>p6:</strong> forks[i+1] ⇐ true</td>
<td><strong>q6:</strong></td>
</tr>
</tbody>
</table>
Algorithm 2.6: 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 some (sequential) transition diagrams \( P_1, \ldots, P_n \) called processes.

The processes \( P_i \):

- do not share variables
- communicate along unidirectional channels \( C, D, \ldots \) connecting at most 2 different 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 \( x \)
Edges in (Sequential) Transition Diagrams

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:

\[ \ell b; C \Rightarrow x; f \ell' \]

\[ \ell b; C \Leftarrow e; f \ell' \]
Example 1

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

$$s_1 \xleftarrow{C \leftarrow 1} t_1 \parallel s_2 \xrightarrow{C \rightarrow x} t_2$$

Obviously, $\{\text{TRUE}\} \ P \ \{x = 1\}$, but how to prove it?
Semantics: Closed Product

Definition

Given \( P_i = (L_i, T_i, s_i, t_i) \), for \( 1 \leq i \leq n \), with disjoint local variable sets, define their \textit{closed product} as \( P = (L, T, s, t) \) such that:

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

and, \( \ell \xrightarrow{a} \ell' \in T \) iff, either

1. \( \ell = \langle \ell_1, \ldots, \ell_i, \ldots, \ell_n \rangle, \)
   \( \ell' = \langle \ell_1, \ldots, \ell'_i, \ldots, \ell_n \rangle, \)
   \( \ell_i \xrightarrow{a} \ell'_i \in T_i \) an internal transition, or

2. \( \ell = \langle \ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n \rangle, \)
   \( \ell' = \langle \ell_1, \ldots, \ell'_i, \ldots, \ell'_j, \ldots, \ell_n \rangle, \)
   \( i \neq j, \) 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
   \( a \) is \( b \land b'; f \circ g \circ [x \leftarrow e] \)
Example 1 cont’d

Observe that the closed product is just

\[
\langle s_1, s_2 \rangle \times x \leftarrow 1 \rightarrow \langle t_1, t_2 \rangle
\]

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

\[
\models \text{TRUE} \quad \Rightarrow \quad (x = 1) \circ [x \leftarrow 1]
\]

which is immediate.
See the glossary of notation for the meaning of all these strange symbols.
To show that \( \{ \phi \} P_1 \parallel \ldots \parallel P_n \{ \psi \} \) is valid, we could simply prove \( \{ \phi \} P \{ \psi \} \) for \( P \) being the closed product of the \( P_i \).
Verification

To show that $\{\phi\} P_1 \parallel \ldots \parallel P_n \{\psi\}$ is valid, we could simply prove $\{\phi\} P \{\psi\}$ for $P$ being the closed product of the $P_i$. This can be done using Floyd’s method, because there are no I/O transitions left in $P$. 

Disadvantage

As with the standard product construction for shared-variable concurrency, the closed product construction leads to a number of verification conditions exponential in the number of processes. Therefore, we are looking for an equivalent of the Owicki/Gries method for synchronous message passing.
To show that $\{\phi\} \ P_1 \parallel \ldots \parallel P_n \{\psi\}$ is valid, we could simply prove $\{\phi\} \ P \{\psi\}$ for $P$ being the closed product of the $P_i$. This can be done using Floyd’s method, because there are no I/O transitions left in $P$.

**Disadvantage**

As with the standard product construction for shared-variable concurrency, the closed product construction leads to a number of verification conditions exponential in the number of processes.
Verification

To show that $\{\phi\} P_1 \parallel \ldots \parallel P_n \{\psi\}$ is valid, we could simply prove $\{\phi\} P \{\psi\}$ for $P$ being the closed product of the $P_i$. This can be done using Floyd’s method, because there are no I/O transitions left in $P$.

Disadvantage

As with the standard product construction for shared-variable concurrency, the closed product construction leads to a number of verification conditions exponential in the number of processes.

Therefore, we are looking for an equivalent of the Owicki/Gries method for synchronous message passing.
A Simplistic Method

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

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

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

3. Prove \[ \models \phi \Rightarrow Q_{s_1} \land \ldots \land Q_{s_n} \text{ and } \models Q_{t_1} \land \ldots \land Q_{t_n} \Rightarrow \psi. \]
Proof of Example 1

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

\[
\text{TRUE} \implies (x = 1) \circ [x \leftarrow 1] \equiv 1 = 1 \equiv \text{TRUE}
\]
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 \( \{ \text{TRUE} \} \ P \{ x = 2 \} \) using the simplistic method. Proof obligations for the transition pairs \((T_1, T_4)\) and \((T_2, T_3)\) should not, but have to be, discharged, and lead to a contradiction, meaning that no inductive assertion network for applying the simplistic method to this example can be found.
Remedy 1: Adding Shared Auxiliary Variables

Use \textit{shared} auxiliary variables to relate locations in processes by expressing that certain combinations will not occur during execution. Only output transitions need to be augmented with assignments to these shared auxiliary variables.

\textbf{Pro} easy

\textbf{Con} re-introduces \textit{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

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

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

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

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

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

\[ \begin{align*}
k &= 0 \\
k &= 1 \\
k &= 2
\end{align*} \]

\[ s_2 \xrightarrow{C \Rightarrow x} l_2 \xrightarrow{C \Rightarrow x} t_2 \]

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

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

\[ \models k = 0 \implies (k = 1 \land x = 1) \circ [k \leftarrow 1] \circ [x \leftarrow 1] \]  
\[ (1) \]

\[ \models k = 0 \land k = 1 \land x = 1 \implies (k = 1 \land k = 2 \land x = 2) \circ [k \leftarrow 1] \circ [x \leftarrow 1] \]
\[ (2) \]

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

\[ \models k = 1 \land x = 1 \implies (k = 2 \land x = 2) \circ [k \leftarrow 2] \circ [x \leftarrow 2] \]
\[ (4) \]

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 contradicting propositions about the value of \( k \), (2) and (3) are vacuously true because their left-hand-sides are false. The right-hand-sides of the implications (1) and (4) simplify to \text{T}rue, which discharges those proof obligations, e.g., for the RHS of (1):

\[
(k = 1 \land x = 1) \circ [k \leftarrow 1] \circ [x \leftarrow 1] \equiv 1 = 1 \land 1 = 1
\equiv \text{True}
\]
Remedy 2: Local Auxiliary Variables + Invariant

Use only local 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:

\[
\begin{align*}
\models Q_{\ell_i} \wedge Q_{\ell_j} \wedge b \wedge b' \wedge I \quad \Rightarrow \quad (Q_{\ell_i'} \wedge Q_{\ell_j'} \wedge I) \circ f \circ g \circ [x \leftarrow e]
\end{align*}
\]

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

\[ s_1 \rightarrow C \leftarrow 1 \rightarrow \ell_1 \rightarrow C \leftarrow 2 \rightarrow t_1 \]

\[ s_2 \rightarrow C \Rightarrow x \rightarrow \ell_2 \rightarrow C \Rightarrow x \rightarrow t_2 \]
Example 2 cont’d

\[
\begin{align*}
S_1 & \quad C \leftarrow 1; \quad k_1 \leftarrow 1 \quad \rightarrow \quad \ell_1 \\
\ell_1 & \quad C \leftarrow 2; \quad k_1 \leftarrow 2 \quad \rightarrow \quad T_1
\end{align*}
\]

\[
\begin{align*}
S_2 & \quad C \Rightarrow x; \quad k_2 \leftarrow 1 \quad \rightarrow \quad \ell_2 \\
\ell_2 & \quad C \Rightarrow x; \quad k_2 \leftarrow 2 \quad \rightarrow \quad T_2
\end{align*}
\]
Example 2 cont’d

\[ C \leftarrow 1; k_1 \leftarrow 1 \quad \text{and} \quad k_1 = 0 \]

\[ \ell_1 \quad C \leftarrow 2; k_1 \leftarrow 2 \quad \text{and} \quad k_1 = 1 \]

\[ t_1 \quad k_1 = 2 \]

\[ s_2 \quad C \Rightarrow x; k_2 \leftarrow 1 \quad \text{and} \quad k_2 = 0 \]

\[ \ell_2 \quad C \Rightarrow x; k_2 \leftarrow 2 \quad \text{and} \quad k_2 = 1 \land x = 1 \]

\[ t_2 \quad k_2 = 2 \land x = 2 \]
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, the same as in the simplistic method. The proof obligations are:

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

(5)

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

(6)

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

(7)

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

(8)
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 True, which discharges those proof obligations, e.g., for the RHS of (8):

\[
(k_1 = 2 \land k_2 = 2 \land x = 2 \land k_1 = k_2) \circ [k_1 \leftarrow 2] \circ [k_2 \leftarrow 2] \circ [x \leftarrow 2] \\
\equiv 2 = 2 \land 2 = 2 \land 2 = 2 \land 2 = 2 \\
\equiv True
\]
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. Next week, we have a break! After the break, we’ll be looking at 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!